Le Programmeur

Développez des applications pour iPhone avec HTML, CSS et JavaScript



Lee S. Barney

PEARSON

LE PROGRAMMEUR

Développez des applications pour l'iPhone

avec HTML, CSS et JavaScript

Lee S. Barney

Traduit par Hervé Soulard,
avec la contribution de Julien Desrosiers

PEARSON

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Mise en pages : TyPAO

ISBN : 978-2-7440-4096-2
Copyright © 2009 Pearson Education France
Tous droits réservés

Titre original : *Developing Hybrid Applications for the iPhone : using HTML, CSS, and JavaScript to Build Dynamic Apps for the iPhone*

Traduit par Hervé Soulard,
avec la contribution de Julien Desrosiers

ISBN original : 978-0-321-60416-3
Copyright © 2009 Pearson Education, Inc.
All rights reserved

Édition originale publiée par Addison-Wesley

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher.

Sommaire

Préface 1

1. Développer avec Dashcode et Xcode 7

2. Modularité JavaScript 33

3. Interfaces utilisateur 57

4. GPS, accéléromètre et autres fonctions natives avec QuickConnectiPhone 91

5. GPS, accéléromètre et autres fonctions natives avec PhoneGap 115

6. Cartes Google 133

7. Bases de données 151

8. Données distantes 185

A. Introduction à JSON 205

B. Plan de développement pour QuickConnectFamily 213

C. Plan de développement pour PhoneGap 219

Index 223

Table des matières

Préface 1
Outils pour le développement d'applications hybrides 1
Comment utiliser ce livre 3
Ressources en ligne 5
Prérequis 5
Remerciements 6
À propos de l'auteur 6
Contacter l'auteur 6

1. Développer avec Dashcode et Xcode 7
Section 1 : utiliser Dashcode et le modèle QuickConnect 8
Section 2 : utiliser Xcode et le modèle QuickConnect 12
Section 3 : les bases d'Objective-C 16
Section 4 : structure Objective-C d'une application QuickConnectiPhone 19
Section 5 : structure Objective-C d'une application PhoneGap 23
Section 6 : embarquer du contenu web avec QuickConnectiPhone 25
Section 7 : embarquer du contenu web avec PhoneGap 29
En résumé 30

2. Modularité JavaScript 33
Section 1 : modularité 33
Section 2 : modularité avec le framework JavaScript QuickConnect 35
Section 3 : conception modulaire dans QuickConnectiPhone 44
Section 4 : implémentation des contrôleurs métier et d'affichage 49
Section 5 : implémentation d'un contrôleur d'erreur 53
Section 6 : étapes de création d'une fonctionnalité de l'application 54
En résumé 55

3. Interfaces utilisateur 57
Section 1 : guide de l'interface utilisateur d'Apple 57
Section 2 : interfaces fondées sur les listes et sur Navigateur 61
Section 3 : applications non fondées sur des listes 64
Section 4 : applications d'immersion 69
Section 5 : créer et utiliser des transformations CSS personnalisées 71

Section 6 : créer et utiliser un module de glisser-déposer, de redimensionnement et de rotation 78
En résumé 89

4. GPS, accéléromètre et autres fonctions natives avec QuickConnectiPhone 91
Section 1 : activation de l'appareil en JavaScript 92
Section 2 : activation de l'appareil en Objective-C 98
Section 3 : implémentation Objective-C de l'architecture de QuickConnectiPhone 107
En résumé 113

5. GPS, accéléromètre et autres fonctions natives avec PhoneGap 115
Section 1 : activation de l'appareil en JavaScript 115
Section 2 : activation de l'appareil en Objective-C 122
En résumé 130

6. Cartes Google 133
Section 1 : afficher une carte dans une application JavaScript QuickConnect 133
Section 2 : implémentation Objective-C du module de cartographie de QuickConnect 138
En résumé 149

7. Bases de données 151
Section 1 : application BrowserDBAccess ... 151
Section 2 : utilisation des bases de données SQLite avec WebView 153
Section 3 : utilisation de bases de données SQLite natives 158
Section 4 : utilisation de DataAccessObject avec les bases de données du moteur WebKit 161
Section 5 : utilisation de DataAccessObject avec les bases de données natives 172
En résumé 182

8. Données distantes 185
Section 1 : application browserAJAXAccess 186
Section 2 : utilisation de ServerAccessObject 188
Section 3 : ServerAccessObject 193
Section 4 : fonctions de contrôle de la sécurité 203
En résumé 204

A. Introduction à JSON 205
Section 1 : les fondamentaux 205
Section 2 : une API JavaScript pour JSON 208
En résumé 211

B. Plan de développement pour QuickConnectFamily 213

C. Plan de développement pour PhoneGap 219

Index 223

Préface

Cet ouvrage explique comment créer un nouveau type d'applications pour l'iPhone : les applications hybrides écrites en HTML, CSS et JavaScript. Ce sont des applications autonomes qui s'exécutent sur l'iPhone comme des applications normales, mais sans que les fichiers requis résident sur un serveur Internet.

La création d'applications hybrides pour l'iPhone permet de réduire le temps de développement et d'apprentissage, car il n'est plus nécessaire de se former à Objective-C ou de maîtriser les frameworks Cocoa.

Outils pour le développement d'applications hybrides

Dans ce livre, nous étudierons les deux paquetages logiciels JavaScript open-source les plus utilisés dans le développement d'applications pour l'iPhone et l'iPod Touch : QuickConnectiPhone et PhoneGap. Ils permettent de construire des applications JavaScript qui accèdent aux fonctionnalités natives de l'appareil, comme le vibreur, les informations de localisation GPS, l'accéléromètre, etc., sans écrire une seule ligne de code Objective-C ou Cocoa.

QuickConnectiPhone (**http://quickconnect.pbwiki.com**) expose le fonctionnement natif de l'appareil et propose un framework complet de haut niveau pour le développement. Il réduit énormément le temps de mise sur le marché d'une application, car le code glue qu'il faut normalement écrire en Objective-C, Cocoa et JavaScript est fourni par le framework. Mieux encore, aucun serveur distant n'est requis pour héberger les fichiers JavaScript, HTML et CSS.

Le second paquetage, PhoneGap (**http://phonegap.com**), expose un nombre moindre de comportements natifs et se présente sous la forme d'une bibliothèque, non d'un framework complet. En tant que bibliothèque, PhoneGap vous permet de construire l'application à votre manière. En revanche, un serveur distant est obligatoire pour héberger les fichiers[1].

De manière à faciliter l'apprentissage et la compréhension, les exemples décrits tout au long de cet ouvrage sont intéressants et importants.

Si votre objectif est de créer des applications installables, si vous avez les connaissances web requises et si vous souhaitez proposer des solutions dynamiques et convaincantes qui seront réellement utilisées, ce livre vous expliquera comment ces deux paquetages peuvent répondre à vos besoins.

Le Tableau P.1 compare les possibilités de chaque paquetage au moment de l'écriture de ces lignes.

Tableau P.1 : Comparaison des possibilités de QuickConnectiPhone et de PhoneGap

Fonctionnalités	*QuickConnectiPhone*	*PhoneGap*
GPS	Oui	Oui
Accéléromètre	Oui	Oui
Vibreur	Oui	Oui
Sons système	Oui	Oui
Réseau *ad hoc* (Bonjour)	Oui	Non
Réseau par câble de synchronisation	Oui	Non
Accès à une base de données par le navigateur	Oui	Non
Accès à une base de données intégrée	Oui	Non
Bibliothèque de glisser-déposer	Oui	Non
Enveloppe AJAX	Oui	Non
Enregistrement/lecture de fichiers audio	Oui	Non
Cartes Google embarquées	Oui	Non
Bibliothèque pour tableaux et graphiques	Oui	Non

1. N.d.T. : à partir de la version 0.7.3, PhoneGap enregistre les fichiers JavaScript, HTML et CSS sur l'appareil, non plus sur un serveur web.

Comment utiliser ce livre

Chaque chapitre est organisé en deux parties. La première explique comment utiliser la fonctionnalité de QuickConnectiPhone ou de PhoneGap qui permet de réaliser une tâche particulière, par exemple obtenir la géolocalisation de l'appareil. La seconde partie présente le code qui se cache derrière l'appel JavaScript utilisé, ainsi que son fonctionnement. Vous pouvez ainsi décider du niveau de détails du code JavaScript et Objective-C jusqu'où vous irez.

- Le Chapitre 1, "Développer avec Dashcode et Xcode", explique comment utiliser Dashcode et Xcode avec QuickConnectiPhone et PhoneGap pour créer rapidement des applications amusantes pour l'iPhone. Il fournit les bases de l'utilisation de Dashcode et montre comment déplacer votre application Dashcode dans Xcode pour sa compilation et son exécution sur des appareils.
- Le Chapitre 2, "Modularité JavaScript", montre comment réduire énormément le temps de mise sur le marché d'une application en exploitant la modularité du framework QuickConnectiPhone. Il explique comment exploiter les contrôleurs frontaux, les contrôleurs d'application et la réflexion JavaScript.
- Le Chapitre 3, "Interfaces utilisateur", facilite l'acceptation de vos applications par l'App Store d'Apple. Il décrit les meilleures pratiques du développement d'applications fonctionnelles pour l'iPhone. Les différents types d'applications généralement créées pour l'iPhone sont examinés, ainsi que les pièges à éviter.
- Le Chapitre 4, "GPS, accéléromètre et autres fonctions natives avec QuickConnectiPhone", montre comment obtenir des informations du GPS, de l'accéléromètre et de l'appareil. Il explique également comment déclencher le vibreur, et lire et enregistrer des fichiers audio. Le framework QuickConnectiPhone est utilisé pour accéder à ces fonctions de l'appareil et pour les employer. Toutes ces possibilités ajoutent à vos applications un côté plaisant.
- Le Chapitre 5, "GPS, accéléromètre et autres fonctions natives avec PhoneGap", montre comment obtenir des informations du GPS, de l'accéléromètre et de l'appareil. Il explique également comment déclencher le vibreur, et lire et enregistrer des fichiers audio. La bibliothèque PhoneGap est utilisée pour accéder à ces fonctions de l'appareil et pour les employer. Toutes ces possibilités ajoutent à vos applications un côté plaisant.
- Le Chapitre 6, "Cartes Google", décrit l'intégration d'une carte Google à une application en utilisant QuickConnectiPhone. Il s'agit de l'une des fonctionnalités les plus demandées, qui évite aux utilisateurs de passer par l'application de cartographie !

- Le Chapitre 7, "Bases de données", explique comment lire et enregistrer des données dans des bases SQLite intégrées à l'application développée avec le framework QuickConnectiPhone. Si vous devez accompagner votre nouvelle application d'une base de données contenant un jeu de données prédéfini, lisez ce chapitre.
- Le Chapitre 8, "Données distantes", montre comment accéder facilement à des données disponibles sur des serveurs et/ou des services distants depuis votre application en utilisant une enveloppe qui permet d'extraire des informations depuis n'importe quel lieu. Par exemple, vous pouvez récupérer des données à partir d'un blog et les fusionner dans un flux Twitter. Grâce au module d'accès aux données distantes de QuickConnectiPhone, rien n'est plus facile.

Ce livre comprend également trois annexes :

- L'Annexe A, "Introduction à JSON", est une courte introduction à JSON (*JavaScript Object Notation*). Ce format est très utilisé et constitue la solution la plus simple pour envoyer des données à leurs destinataires.
- L'Annexe B, "Plan de développement pour QuickConnectFamily", présente le futur de QuickConnectiPhone. Si vous prévoyez de développer des applications pour l'iPhone et d'autres plates-formes, comme les téléphones Android de Google, les téléphones de Nokia, les Blackberry et les ordinateurs de bureau sous Mac OS X, Linux et Windows, vous devez lire cette annexe.
- L'Annexe C, "Plan de développement pour PhoneGap", présente le futur de PhoneGap. Si vous envisagez de créer des applications pour l'iPhone et d'autres plates-formes, comme les téléphones Android de Google, les téléphones de Nokia, les Blackberry et les ordinateurs de bureau sous Mac OS X, Linux et Windows, vous devez lire cette annexe.

Code des exemples

Les fichiers des exemples de code sont disponibles depuis le site web Pearson (**http://www.pearson.fr**), en suivant le lien Codes sources sur la page dédiée à ce livre. Les dernières versions de ces exemples, en anglais, sont fournies dans le paquetage de QuickConnectiPhone et dans le modèle d'application PhoneGap. Dans cet ouvrage, les commentaires ont été traduits afin de faciliter la lecture, mais certains seront inévitablement en anglais car générés par les frameworks.

Ressources en ligne

Le développement de QuickConnectiPhone et de PhoneGap est en cours et les évolutions sont rapides. Pour connaître les nouvelles fonctions et les nouvelles possibilités, et pour en savoir plus, consultez les liens suivants.

QuickConnectiPhone

- téléchargement du framework et de plusieurs exemples (**http://sourceforge.net/projects/quickconnect/**) ;
- blog concernant le développement (**http://tetontech.wordpress.com**) ;
- wiki dédié au framework (**http://quickconnect.pbwiki.com/FrontPage**) ;
- groupe Google (**http://groups.google.com/group/quickconnectiPhone/**) ;
- service Twitter (**http://twitter.com/quickconnect**).

PhoneGap

- téléchargement du framework et de plusieurs exemples (**http://sourceforge.net/projects/phonegapinstall/**) ;
- site web de la bibliothèque (**http://www.phonegap.com/**) ;
- wiki dédié à la bibliothèque (**http://phonegap.pbwiki.com/**) ;
- groupe Google (**http://groups.google.com/group/phonegap**) ;
- service Twitter (**http://twitter.com/phonegap**).

Prérequis

Pour réellement profiter du contenu de cet ouvrage, vous devez posséder des connaissances de base en HTML, CSS et JavaScript. Si vous avez déjà créé des pages web à l'aide de ces technologies, vous êtes prêt à créer des applications pour l'iPhone. Si vous avez besoin d'aide avec Objective-C, que ce soit pour QuickConnectiPhone ou PhoneGap, elle vous sera fournie. Ce livre ne constitue pas une introduction à Objective-C ni à son utilisation dans le développement d'applications pour l'iPhone.

Vous devez télécharger et installer les outils Xcode d'Apple disponibles sur le site web du développeur pour l'iPhone (**http://developer.apple.com/iphone**). Ils nécessitent Mac OS X 10.5 ou ultérieur et une machine Intel.

Bien que cela ne soit pas obligatoire, il est préférable de disposer d'un iPhone ou d'un iPod Touch afin de tester et d'exécuter les applications sur ces appareils.

Remerciements

Je souhaite remercier tout particulièrement Daniel Barney pour avoir travaillé sur le code d'intégration des cartes Google. Merci également à mes collègues du département des technologies de l'information de l'université Brigham Young, Idaho, pour leur attention et leurs suggestions.

À propos de l'auteur

Lee S. Barney (Rexburg, Idaho) est professeur dans le département des technologies de l'information au Business and Communication College de l'université Brigham Young, dans l'Idaho. Il a travaillé comme directeur des technologies de l'information pour @HomeSoftware, une société qui développe des applications web mobiles de planification pour le marché des soins médicaux à domicile. Avant cela, il a été pendant plus de sept ans programmeur, ingénieur logiciel senior, directeur du service de la fiabilité des chefs de projet pour AutoSimulations, Inc., le principal fournisseur de logiciels de planification pour l'industrie des semi-conducteurs. Il est l'auteur du livre *Oracle Database AJAX & PHP Web Application Development*.

Contacter l'auteur

Pour contacter l'auteur par courrier électronique, utilisez l'adresse **quickconnect-family@gmail.com**. Pour d'autres méthodes de contact, utilisez les liens Twitter, wiki et groupe Google donnés précédemment.



1

Développer avec Dashcode et Xcode

Utilisés ensemble, Dashcode et Xcode offrent la puissance et la simplicité d'utilisation nécessaires à la création d'applications hybrides uniques et passionnantes pour l'iPhone. Puisque ces deux outils fournissent des modèles personnalisés adaptés aux applications hybrides pour l'iPhone, vous n'avez pas besoin de créer votre propre enveloppe en Objective-C. Les trois premières sections de ce chapitre expliquent comment utiliser les modèles d'application existants pour Dashcode et Xcode. Grâce à ces modèles, vous pouvez créer rapidement des applications hybrides pour l'iPhone. Les quatre dernières sections introduisent les bases d'Objective-C et la manière de structurer une application iPhone en Objective-C dans les deux outils les plus employés – QuickConnectiPhone et PhoneGap.

Section 1 : utiliser Dashcode et le modèle QuickConnect

Dans les applications hybrides pour l'iPhone, une grande partie de l'interface utilisateur et des interactions est créée avec HTML, JavaScript et CSS. Par conséquent, le développement et le débogage se font principalement dans Dashcode. Les possibilités et la facilité d'emploi de l'outil de construction d'interfaces par glisser-déposer fourni par Dashcode sont uniques. Dashcode est utilisé pour créer une grande partie de l'application et sert également à la tester en employant le simulateur d'iPhone et les outils de débogage intégrés.

Puisque les applications hybrides pour l'iPhone ont beaucoup de code en commun, la création d'un modèle contenant ce code permet d'éviter son écriture ou son importation à chaque début d'un nouveau projet. Nous reviendrons sur ce code commun au Chapitre 2.

Pour télécharger QuickConnectiPhone, allez sur la page **http://sourceforge.net/projects/quickconnect**. Le paquetage obtenu inclut un modèle Dashcode qui vous aidera à créer des applications hybrides pour l'iPhone. Le programme d'installation de QuickConnectFamily ajoute ce modèle à Dashcode. Malheureusement, au moment de l'écriture de ces lignes, les auteurs de PhoneGap ne proposent pas de modèle Dashcode.

Après avoir exécuté le programme d'installation de QuickConnectFamily et avoir lancé Dashcode, le modèle QuickConnectiPhone est proposé à la fin de la liste affichée dans la rubrique Dashboard Widget. En double-cliquant sur l'icône `QuickConnectiPhone`, vous arrivez directement à l'écran principal de Dashcode, avec une interface utilisateur vierge (voir Figure 1.1).

Pour comprendre les fichiers inclus dans le framework et les utiliser facilement, nous allons créer une première interface utilisateur simple avec Dashcode et la déployer sur l'iPhone avec Xcode. Elle sera constituée uniquement d'un bouton et d'une zone de texte. Lors d'un clic sur le bouton, la zone de texte affichera : "Vous l'avez fait !"

Applications hybrides et boîte d'alerte

Les développeurs qui ont l'habitude de coder en JavaScript utilisent souvent une boîte d'alerte pour déboguer l'application ou afficher des messages à l'utilisateur. La fonction JavaScript `alert` est en réalité un appel à du code natif dans le navigateur, non une possibilité du moteur JavaScript.

Ce fonctionnement n'est pas mis en œuvre dans les applications QuickConnectiPhone car l'utilisation des boîtes de dialogue est contraire aux standards établis par Apple pour l'interface utilisateur des applications iPhone. Pour le débogage, vous pouvez vous servir du débogueur de Dashcode. Si vous déplacez votre application dans Xcode, vous pouvez employer la fonction `debug` pour afficher des messages dans la console de Xcode.

PhoneGap propose une boîte d'alerte, mais pas la fonction de débogage de Xcode.

Pour afficher des informations importantes, ajoutez-les dans un élément HTML `<div>`, ou autre, quel que soit l'outil que vous utilisez.

N'oubliez pas : vous devez être attentif, sans être alarmiste.

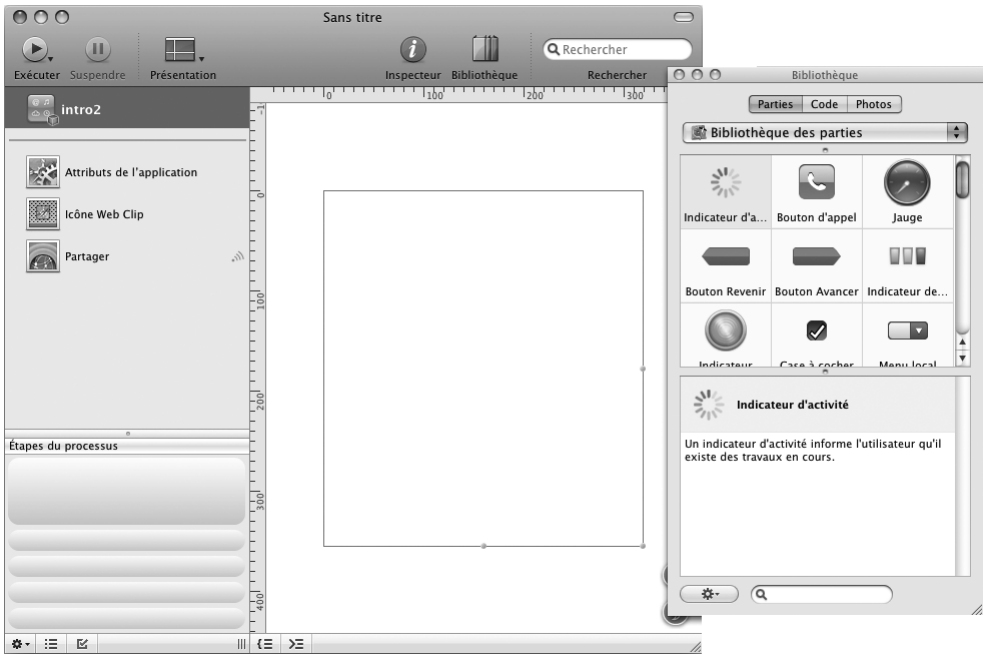


Figure 1.1
Le modèle QuickConnectiPhone est utilisé dans Dashcode. Le contenu de la bibliothèque standard est affiché.

Avant de créer l'interface utilisateur, vérifiez que la fenêtre Bibliothèque est ouverte. Ensuite, recherchez l'élément `Texte` dans la bibliothèque des parties et faites-le glisser sur l'écran vierge de l'application. Une nouvelle zone de texte, contenant le mot "Texte", s'affiche en haut de l'interface. Par défaut, la taille de cette zone de texte est fixée à 100 %. Dashcode a inséré dynamiquement une balise HTML `<div>` dans le fichier *index.html* de l'application, ainsi que du code JavaScript pour la compléter par le texte, la couleur d'arrière-plan et les autres caractéristiques que vous choisissez.

Pour notre exemple, nous souhaitons fixer l'identifiant de la balise de texte à `display` et effacer son contenu. Pour cela, nous allons employer l'inspecteur des éléments de l'interface.

Cliquez sur l'icône `Inspecteur` dans la barre supérieure de Dashcode de manière à activer la boîte de dialogue correspondante. Sélectionnez l'onglet `Attributs` (de couleur rouge et blanc) dans le coin supérieur gauche de l'inspecteur, fixez le champ `Identifiant` à `display` et effacez le contenu du champ `Étiquette`.

Ajoutez un bouton poussoir à l'interface, en faisant glisser et en déposant la partie correspondante sous la zone de texte. L'inspecteur affiche à présent les informations concernant ce bouton, non plus celles du champ de texte. Ouvrez l'onglet `Comportements` en cliquant sur le cube bleu dans le coin supérieur droit de l'inspecteur. Il permet de définir les fonctions JavaScript qui serviront de gestionnaires pour les types d'événements listés. Vous remarquerez que plusieurs des événements standard de la souris sont absents. Ils ont été remplacés par `ongesturestart`, `ongesturechange` et `ongestureend`. Saisissez `changeText` dans la colonne `Gestionnaires` de l'événement `onclick`. Cette opération ajoute une fonction `changeText` dans le fichier *main.js* et affiche son contenu à l'écran afin que vous puissiez saisir le code exécuté lorsque l'événement `onclick` est déclenché. Pour notre exemple simple, placez le code suivant dans la fonction `changeText` :

```
document.getElementById('display').innerHTML = "Vous l'avez fait !";
```

Notre application peut à présent être exécutée dans le simulateur d'iPhone[1]. Cliquez sur l'icône `Exécuter` dans le coin supérieur gauche de Dashcode. Le simulateur est alors démarré et l'application y est exécutée (voir Figure 1.2).

Puisque l'application est déboguée et terminée, vous pouvez déplacer le code dans Xcode en vue de son déploiement sous forme d'une application installable.

Tout d'abord, vous devez utiliser Dashcode pour déployer l'application actuelle. En effet, le code est caché dans le projet Dashcode et contient des directives que seul Dashcode peut comprendre. Cliquez sur l'icône `Partager` dans la partie gauche de Dashcode afin d'afficher l'écran de déploiement. Vous pourrez ainsi enregistrer l'intégralité des fichiers HTML, CSS et JavaScript sur le disque de manière à les inclure dans votre application. Dans le champ `Chemin`, saisissez un nom pour le nouveau répertoire dans lequel seront placés ces fichiers. Ils peuvent à présent être importés dans Xcode. La Figure 1.3 montre l'écran de déploiement.

Pour de plus amples informations concernant les fichiers JavaScript inclus dans ce modèle et leur utilisation pour simplifier la création d'une application, consultez le Chapitre 2.

1. N.d.T. : par défaut, le simulateur d'iPhone est normalement configuré pour la langue anglaise. Pour le passer en français, utilisez l'application Réglages.



Figure 1.2
L'application en cours d'exécution dans le simulateur d'iPhone de Dashcode.



Figure 1.3
L'écran de déploiement affiche l'application terminée en cours de déploiement vers le répertoire Exemple1_Chapitre1.

Section 2 : utiliser Xcode et le modèle QuickConnect

Puisque vous avez exécuté le programme d'installation de QuickConnectFamily, le modèle Xcode pour les applications QuickConnectiPhone a été installé. Nous allons l'utiliser pour créer le projet Xcode de notre application hybride QuickConnectiPhone. Cette section décrit les différentes étapes de la procédure. Le wiki QuickConnectFamily propose une vidéo qui décrit cette procédure (**http://quickconnect.pbwiki.com/Moving-Dashcode-projects-to-Xcode**).

Commencez par sélectionner `File > New Project`, puis `iPhone OS > Applications`. Double-cliquez sur l'icône `QuickConnect iPhone Application` et nommez le projet (le répertoire correspondant est créé sur le disque dur). Xcode crée un projet qui comprend les fichiers Objective-C nécessaires à l'exécution de l'application JavaScript directement sur l'appareil, sans disposer d'un accès réseau ou d'un accès Internet. Dans le groupe `Resources` de l'application, vous trouverez un ensemble de fichiers HTML, CSS et JavaScript.

L'un de ces fichiers se nomme *index.html*. Il contient tout le code HTML, CSS et JavaScript d'un exemple d'application prête à fonctionner. La Figure 1.4 montre l'exécution de cet exemple sur le simulateur sous forme d'une application installée. Avant de l'essayer, vous devez préciser à Xcode d'utiliser le simulateur de l'iPhone. Pour cela, ouvrez le menu `Project > Set Active SDK`, puis choisissez un simulateur dans la liste.

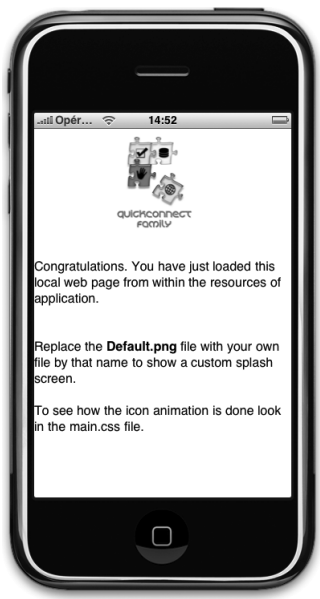


Figure 1.4
L'application QuickConnect par défaut.

Pour inclure dans ce projet les fichiers créés précédemment dans Dashcode, commencez par supprimer les fichiers suivants à partir du groupe `Resources` :

- *index.html* ;
- *main.css* ;
- *main.js* ;
- les éventuels fichiers du sous-groupe `Parts` ;
- les éventuels fichiers du sous-groupe `Images`.

Ensuite, importez les fichiers *index.html*, *main.css* et *main.js* de l'exemple précédent. Pour cela, maintenez enfoncée la touche Contrôle et cliquez du bouton droit sur le groupe `Resources`, puis sélectionnez `Add > Existing Files`. Allez dans le répertoire dans lequel vous avez déployé l'application Dashcode et sélectionnez *index.html*, *main.css* et *main.js*. Vous pouvez copier les fichiers dans le projet Xcode ou les utiliser à partir de leur emplacement actuel. Pour cet exemple, cochez la case `Copy items into destination group's folder (if needed)`.

Copier ou ne pas copier, telle est la question

C'est à vous de décider si vous devez copier les fichiers existants ou laisser Xcode utiliser des références vers ces fichiers. Comment prendre votre décision ? Chaque méthode présente des avantages.

Si vous copiez les fichiers, le répertoire du projet est complet et peut être passé à d'autres développeurs sans qu'ils aient besoin de reproduire la structure de répertoires de la machine d'où proviennent les fichiers d'origine.

Avec les références, vous pouvez retourner dans Dashcode pour apporter des modifications et exporter ensuite le projet de manière à actualiser les fichiers. Vous n'avez pas besoin de les importer à nouveau dans Xcode.

Cliquez ensuite du bouton droit sur le groupe `Parts` (s'il n'existe pas sur le disque dur, il est affiché en rouge et vous devrez le créer avant l'importation) et importez les fichiers qui se trouvent dans le dossier *Parts*. Répétez cette opération pour le groupe `Images` et le dossier *Images*. L'application est quasiment prête à être exécutée.

Puisque des fichiers ont été ajoutés au groupe `Resources`, il faut indiquer à Xcode qu'il doit les inclure dans les ressources de l'application. Ouvrez la rubrique `Targets`, puis développez votre application et `Copy Bundle Resources`. Vous voyez alors les fichiers de ressources requis par l'application. Sélectionnez les fichiers, non les groupes, que vous

venez d'ajouter dans votre projet et faites-les glisser dans la liste `Copy Bundle Resources`. Ensuite, développez la rubrique `Compile Sources` et supprimez tous les fichiers JavaScript, car ils ne pourront évidemment pas être compilés. Pour cela, maintenez enfoncée la touche Ctrl, cliquez du bouton droit sur chacun d'eux et sélectionnez `Delete`. Les fichiers sont supprimés de la liste des fichiers à compiler, mais ils ne sont pas retirés du projet ou du disque.

Puisque Dashcode utilise des répertoires et que Xcode utilise des groupes, vous devez apporter deux autres modifications avant de pouvoir exécuter l'application. La première concerne la section `<head>` du fichier *index.html*. Puisque les fichiers JavaScript et les autres fichiers référencés sont placés dans le répertoire des ressources de l'application finale, les références aux répertoires *Parts* et *QCiPhone* doivent être supprimées. Par exemple, avant la suppression des références, voici l'aspect d'une balise `<script>` :

```
<script type="text/JavaScript" src="Parts/utilities.js" charset="utf-8"></script>
```

Elle doit devenir la suivante :

```
<script type="text/JavaScript" src="utilities.js" charset="utf-8"></script>
```

Puisque des images sont utilisées pour les boutons et autres éléments créés dans Dashcode, vous devez également retrouver les instances de la chaîne `Images/` dans l'ensemble du projet et les remplacer par une chaîne vide. Cette opération est très facile en ouvrant le menu `Edit`, en choisissant `Find > Find in Project` et en recherchant `Images/`. La Figure 1.5 montre les résultats de la recherche dans notre exemple, avant la modification du fichier *PushButton.js*.

Vous pouvez à présent installer et exécuter votre application en sélectionnant l'icône `Build and Go`, qui se trouve dans la barre supérieure de l'application Xcode. Si vous recevez le message d'erreur "No provisioned iPhone OS device is connected", vous pouvez installer et exécuter l'application dans le simulateur à la place de votre appareil. Pour cela, cliquez sur `Succeeded` dans le coin inférieur droit de la fenêtre Xcode, ouvrez la liste `Device | Debug` et sélectionnez une version du simulateur. Notez que vous pouvez également choisir `Release` ou `Debug` dans cette liste déroulante. Cette boîte de dialogue est fréquemment utilisée au cours des développements pour effectuer ce type de modification. La Figure 1.6 montre l'application installée et en cours d'exécution dans le simulateur.

Félicitations, vous venez de terminer votre première application hybride pour l'iPhone.

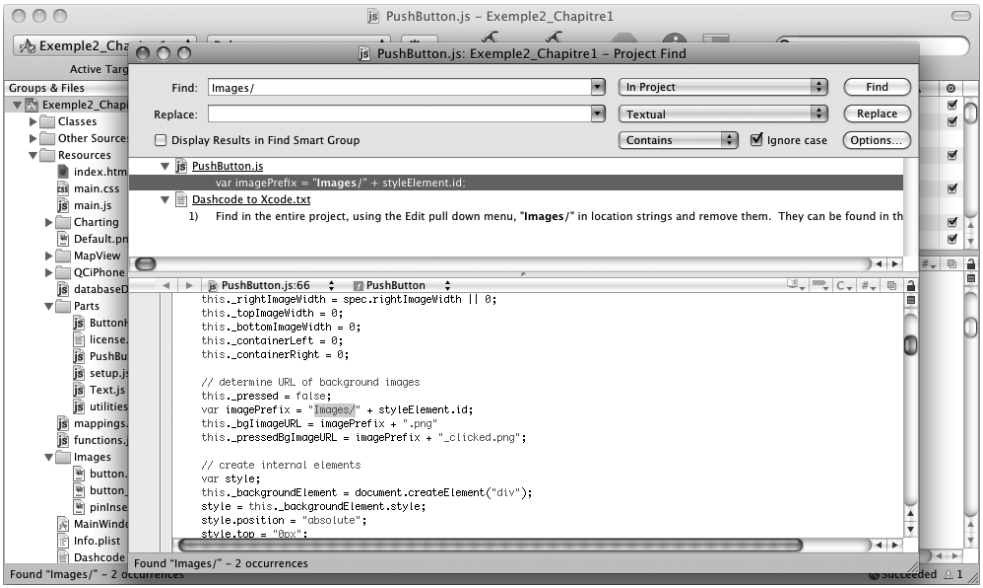


Figure 1.5
L'écran de recherche montre les résultats de la recherche de la chaîne Images/ sur l'ensemble du projet.

Figure 1.6
L'application Exemple2_Chapitre1 est installée et s'exécute sur le simulateur d'iPhone.



Approvisionnement

L'approvisionnement (*provisioning*) est un processus en plusieurs étapes que vous, ou votre représentant, devez exécuter pour que vous puissiez installer et exécuter votre application sur un iPhone.

Pour préparer votre iPhone, vous devez être membre de l'ADC (*Apple Developer Connection*) et être abonné au Program Portal. Si vous faites partie d'une équipe, l'approvisionnement a sans doute déjà été effectué pour vous. Dans ce cas, il vous suffit simplement d'envoyer les informations d'approvisionnement à votre iPhone.

Le site ADC détaille la procédure d'approvisionnement. Assurez-vous de réaliser toutes les étapes indiquées car toute erreur risque de se solder par un échec qui vous empêchera de tester vos applications sur votre appareil.

Section 3 : les bases d'Objective-C

Cette section ne constitue pas un didacticiel détaillé sur Objective-C, pas plus qu'une présentation complète sur la manière d'employer ce langage pour développer des applications pour l'iPhone. Elle se contente d'expliquer comment les classes Objective-C utilisées dans des modèles interagissent et se comportent, afin que vous puissiez exploiter ces informations dans les applications hybrides pour l'iPhone. Elle suppose que vous ayez une certaine connaissance des objets, méthodes et attributs. Si vous souhaitez en savoir plus sur le framework JavaScript ou si vous n'êtes pas intéressé par le code Objective-C, vous pouvez sauter la suite de ce chapitre et aller directement au Chapitre 2. Pour de plus amples informations concernant le développement en Objective-C pour l'iPhone, consultez l'ouvrage *The iPhone Developer's Cookbook: Building Applications with the iPhone SDK*, d'Erica Sadun.

Objective-C est un langage intéressant. Les personnes qui possèdent une expérience dans d'autres langages, comme JavaScript, PHP, Java ou Perl, risquent de le trouver intimidant et incompréhensible au premier abord. Toutefois, il mérite d'être étudié de plus près et pas uniquement parce qu'il s'agit du langage "natif" de l'iPhone.

Objective-C est une variante orientée objet de C. Vous pouvez employer tous les aspects puissants mais dangereux de la programmation en C/C++, comme l'arithmétique de pointeurs, et bénéficier de mécanismes qui facilitent le travail, comme la gestion automatique de la mémoire. Dans un langage orienté objet, la manière d'instancier un objet est la première chose à connaître. Si une classe nommée `Mammifere` est disponible dans le code

source et si elle possède deux attributs, `couleurPoils` et `tauxButyreux`, il est possible de l'instancier en JavaScript de la manière suivante :

```
var unMammifere = new Mammifere("brun", 0.15);
```

Vous pourriez penser que cette méthode est normale et attendre des autres langages un comportement semblable. Dans ce cas, vous risquez de trouver étrange l'instanciation des objets en Objective-C. Voici l'instanciation équivalant à la précédente en Objective-C :

```
Mammifere *unMammifere = [[Mammifere alloc] initAvecCouleur: @"brun" etTauxBuryteux:
➥0.15];
```

Certaines parties sont compréhensibles, d'autres, beaucoup moins. Si vous y réfléchissez, `alloc` a un sens car c'est ainsi que de l'espace en mémoire RAM est alloué à l'objet `Mammifere`. Même `initAvecCouleur` et `etTauxBuryteux` ont un sens en tant que mutateurs ou passeurs des deux paramètres requis. Toutefois, que se passe-t-il réellement et que signifient les crochets ?

Pour toutes les interactions avec les objets et les autres éléments qui ne sont pas nécessairement des objets dans d'autres langages, Objective-C utilise le passage de messages. Étudions le code suivant :

```
[Mammifere alloc]
```

Précédemment, nous avons supposé que ce fragment de code allouait de l'espace en RAM pour un objet de type `Mammifere`. C'est effectivement le cas. Les crochets autour de `Mammifere` et d'`alloc` indiquent que l'objet applicatif qui représente la classe `Mammifere` reçoit le message `alloc`. Autrement dit, ce bout de code doit se lire "passer un message `alloc` à l'objet de classe `Mammifere`". Le passage du message `alloc` à l'objet de classe `Mammifere` conduit au retour d'un pointeur sur un nouvel objet `Mammifere`.

Pointeurs

Les pointeurs sont intéressants. Toutefois, de nombreux développeurs en ont peur car ils ne les comprennent pas ou ne les connaissent pas.

Pour mieux les expliquer, prenons l'analogie suivante. Imaginez une foule immense dans laquelle se trouvent Anne et Jean. Ces deux personnes se connaissent et Anne sait où Jean se trouve dans la foule. Vous abordez Anne et lui demandez où est Jean. Anne pointe son doigt vers Jean et répond "le voici".

À ce moment-là, Anne est un pointeur sur Jean. Si vous considérez un pointeur comme quelque chose qui sait où un objet se trouve en mémoire, vous avez tout compris.

Ce nouvel objet `Mammifere` instancié peut recevoir des messages. L'extrait de code précédent contient un autre message pour ce nouvel objet `Mammifere`.

Ce nouveau message combine `initAvecCouleur` et `etTauxBuryteux`. Nous savons que ces deux parties représentent un message car elles sont, avec le nouvel objet `Mammifere`, entourées de crochets qui, rappelons-le, signifient un passage de message. Les multiples parties d'un message sont séparées par des espaces.

Par ailleurs, les différentes parties du message et les valeurs correspondantes sont liées par le caractère deux-points (`:`). Un seul paramètre peut être associé à chaque partie du message. Le message passé retourne un pointeur sur le nouvel objet `Mammifere` alloué afin qu'il puisse être enregistré localement en vue de son utilisation ultérieure. En Objective-C, ces indicateurs de messages, qu'il s'agisse d'un message à une ou à plusieurs parties, sont appelés sélecteurs car ils désignent les méthodes de l'objet sélectionnées par le compilateur et exécutées.

Revenez au projet Xcode `Exemple2_Chapitre1` créé à la Section 2. Dans le fichier *Exemple2_Chapitre1AppDelegate.m*, examinez la méthode `applicationDidFinishLaunching` générée par le modèle. Ne vous occupez pas du fonctionnement du code, simplement du passage de message :

```
1 - (void)applicationDidFinishLaunching:(UIApplication *)application {
2     // Cette ligne aide au débogage. Vous pouvez voir exactement où sont placées
      ➥vos vues.
3     // Si vous voyez du rouge, la fenêtre est vide. Sinon, utilisez le noir.
4     //window.backgroundColor = [UIColor redColor];
5
6
7     QuickConnectViewController *aBrowserViewController=
      ➥[[QuickConnectViewController alloc] init];
8
9     // Ajouter la vue CreateViewController à window en tant que vue secondaire.
10    [window addSubview:aBrowserViewController.view];
11
12    [window makeKeyAndVisible];
13 }
```

La ligne 7 doit vous sembler familière. Elle n'implique aucun mammifère, mais elle utilise les messages `alloc` et `init` que vous avez rencontrés précédemment. Dans ce cas, un objet `QuickConnectViewController` est alloué et initialisé. Son objet de classe reçoit le message `alloc` et retourne un pointeur sur le nouvel objet `QuickConnectViewController` alloué. Celui-ci, au travers de son pointeur, reçoit le message `init`.

Ce message réalise une opération semblable à celle du message multipartie `initAvecCouleur:etTauxBuryteux` de `Mammifere`, mais il est beaucoup plus simple. Il s'agit d'un

message en une partie, sans aucun paramètre. Plus loin dans ce chapitre, vous verrez comment créer des méthodes d'initialisation et d'autres méthodes exécutées par les objets lorsqu'ils recevront un message.

La ligne 10 envoie un message à `window`. Ce message `addSubView` possède un paramètre qui correspond à l'attribut `view` contenu dans l'objet `aBrowserViewController`.

Vous savez à présent comment instancier un objet, enregistrer localement un pointeur sur le nouvel objet, comment accéder aux attributs d'un objet et comment passer aux objets des messages avec ou sans paramètre. Vous disposez donc des bases d'Objective-C nécessaires pour comprendre le code des modèles QuickConnectiPhone et PhoneGap. Nous allons à présent voir comment les applications Objective-C sont assemblées.

Section 4 : structure Objective-C d'une application QuickConnectiPhone

Bien que cette section présente du code issu du modèle d'application QuickConnectiPhone, une approche équivalente est employée par PhoneGap et toutes les autres mises en œuvre des applications hybrides. Vous pouvez utiliser l'une de ces implémentations ou, en les étudiant, créer votre propre version.

Imaginez que vous disposiez d'un grand nombre de parts dans une entreprise prospère. Imaginez qu'une réunion des actionnaires ait lieu pour élire le président du conseil d'administration, mais que vous ne puissiez pas y participer en raison de vos vacances à Tahiti. Comment pouvez-vous néanmoins voter ?

Si vous donnez pouvoir à une autre personne pour voter à votre place, elle devient votre mandataire. En tant que mandataire, elle est pleinement autorisée à agir pour votre compte lors de la réunion. Votre mandataire peut donc être appelé votre délégué. Ce délégué vous considère comme le mandant car vous êtes l'actionnaire réel. La Figure 1.7 illustre ces relations. Les applications Objective-C pour l'iPhone se fondent sur ces relations mandant-délégué entre des objets, dont l'un représente le mandant et l'autre, le délégué.

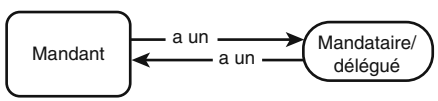


Figure 1.7
Une représentation graphique de la relation mandant-délégué.

Les relations de type mandant-délégué sont très répandues dans les applications Objective-C pour l'iPhone. Voici celles qui nous intéressent principalement :

- `UIApplication/UIApplicationDelegate` ;

- `UIWebView/UIWebViewDelegate` ;
- `UIAccelerometer/UIAccelerometerDelegate`.

À ce stade, vous devez comprendre que l'implémentation des méthodes du protocole pour ces délégués indique à l'application, à la vue ou à l'accéléromètre que vous souhaitez que la prise en charge d'événements spécifiques se fasse par le délégué à la place de la méthode. Chaque méthode du protocole est associée à un événement.

Protocoles

Un protocole est un ensemble de méthodes qui peuvent être ajoutées à une classe afin qu'elle réponde à certains messages.

En ayant ces concepts de mandant-délégué à l'esprit, examinons une classe qui joue le rôle de délégué. Le fichier d'en-tête de la classe `Exemple2_Chapitre1AppDelegate` a été généré par le modèle QuickConnectiPhone lorsque que vous avez créé l'application `Exemple2_Chapitre1` à la Section 2. Il est donné ci-après.

Les fichiers d'en-tête Objective-C, ceux qui se terminent par *.h*, déclarent des classes. Examinez celui de `Exemple2_Chapitre1AppDelegate`, mais sans vous occuper du fichier d'implémentation :

```
1 //  Exemple2_Chapitre1AppDelegate.h
2 #import <UIKit/UIKit.h>
3 #import "QuickConnectViewController.h"
4
5 @interface  Exemple2_Chapitre1AppDelegate : NSObject <UIApplicationDelegate> {
6     IBOutlet UIWindow *window;
7     QuickConnectViewController *browserViewController;
8 }
9
10 @property (nonatomic, retain) UIWindow *window;
11 @property (nonatomic, retain) QuickConnectViewController *browser
   ➥ViewController;
12
13 @end
```

Examinez la ligne 5. Si vous connaissez Java, ne vous laissez pas induire en erreur par l'indicateur `@interface`. Il ne signifie pas que cette classe équivaille à une interface Java. Il signifie que ce fichier contient la définition de l'interface de la classe. Ce fichier d'en-tête déclare les attributs de la classe `Exemple2_Chapitre1AppDelegate`, la manière d'y

accéder et les méthodes qui doivent être mises en œuvre dans le fichier d'implémentation. Cette classe ne possède aucune méthode en propre.

S'il ne s'agit pas d'une déclaration d'interface à la manière de Java, quel est donc le rôle de la ligne 5 ? Elle déclare le nom de la classe, `Exemple2_Chapitre1AppDelegate`, et utilise le caractère deux-points pour indiquer qu'elle dérive de la classe `NSObject`. La classe est donc un `NSObject` et peut accepter tout message défini par `NSObject`. Si vous examinez la classe `NSObject` dans la documentation de l'API (accessible au travers du menu d'aide de Xcode), vous pouvez constater qu'elle possède une méthode `description`. Par conséquent, puisque `Exemple2_Chapitre1AppDelegate` hérite de `NSObject`, elle possède également une méthode `description`.

Après la déclaration d'héritage de `NSObject`, vous voyez `<UIApplicationDelegate>`. Cela indique à la classe `Exemple2_Chapitre1AppDelegate` qu'elle se comporte comme un délégué de votre application et vous permet de mettre en œuvre les méthodes des messages du protocole `UIApplicationDelegate` dans le fichier d'implémentation de `Exemple2_Chapitre1AppDelegate`. L'une des méthodes de ce protocole se nomme `applicationDidFinishLaunching`.

Cette méthode est invoquée lorsque le chargement de l'application est terminé et qu'elle est prête à être exécutée. La méthode permet de personnaliser l'application ou de demander des informations supplémentaires à l'utilisateur.

Dans le code suivant, la ligne 13 contient la définition de `applicationDidFinish-Launching` donnée par QuickConnectiPhone dans le fichier d'implémentation. Elle commence par un signe moins (`-`) qui indique qu'il s'agit d'une méthode d'objet. `(void)` signifie que la méthode ne retourne aucune valeur, tandis que `:(UIApplication *)application` indique qu'elle attend un paramètre de type `UIApplication`.

```
 1 //
 2 //  Exemple2_Chapitre1AppDelegate.m
 3 //  Exemple2_Chapitre1
 4 //
 5
 6 #import "Exemple2_Chapitre1AppDelegate.h"
 7 @implementation Exemple2_Chapitre1AppDelegate
 8
 9 @synthesize window;
10 @synthesize browserViewController;
11
12
13 - (void)applicationDidFinishLaunching:(UIApplication *)application {
14     // Cette ligne aide au débogage. Vous pouvez voir exactement où sont placées
       ➥vos vues.
```

```
15      // Si vous voyez du rouge, la fenêtre est vide. Sinon, utilisez le noir.
16      //window.backgroundColor = [UIColor redColor];
17
18
19      QuickConnectViewController *aBrowserViewController=
        ➥[[QuickConnectViewController alloc] init];
20
21      // Ajouter la vue CreateViewController à window en tant que vue secondaire.
22      [window addSubview:aBrowserViewController.view];
23
24      [window makeKeyAndVisible];
25  }
```

Puisqu'elle fait partie de la classe déléguée de votre application, cette méthode `applicationDidFinishLaunching` est invoquée automatiquement lorsque le chargement de l'application est terminé. C'est pourquoi elle peut être utilisée pour instancier d'autres éléments nécessaires à l'application. Dans cet exemple, vous pouvez voir à la ligne 19 l'allocation et l'initialisation d'une autre classe, `QuickConnectViewController`, ajoutée à l'application par le modèle.

Les applications pour l'iPhone utilisent des vues, et n'importe quel objet `UIWindow` ou `UIView` peut contenir des objets `UIView`. Par conséquent, il est possible d'avoir des vues imbriquées. Cependant, cette conception est déconseillée dans les applications pour l'iPhone. À la place de cette approche hiérarchique, la plupart des développeurs choisissent d'échanger une vue secondaire par une autre à un niveau aussi élevé que possible, selon les besoins de l'utilisateur.

Le remplacement des vues secondaires réduit la complexité de la structure des vues de l'application. Par chance, le modèle utilisé pour créer l'application a placé le nombre de vues imbriquées adéquat pour que le contenu web puisse être affiché. En réalité, comme nous le verrons plus loin, il a inséré une vue secondaire web dans la vue qui a été ajoutée à l'objet `window`.

Un attribut de la classe `QuickConnectViewController` correspond à l'objet de vue qui affiche le contenu dans la fenêtre de l'application. Cet attribut doit être ajouté à la fenêtre principale en tant que vue secondaire, une opération effectuée à la ligne 22.

En plus de posséder la vue du contenu, la classe `QuickConnectViewController` joue également le rôle de délégué pour la localisation GPS, l'accéléromètre, la vue web et d'autres types d'événements.

Section 5 : structure Objective-C d'une application PhoneGap

En tant qu'applications pour l'iPhone, les applications PhoneGap respectent également la même structure mandant-délégué que les applications QuickConnectiPhone (pour de plus amples informations, consultez la Section 4). La classe déléguée que vous devez comprendre se nomme `GlassAppDelegate`. Tout comme la classe `Exemple2_Chapitre1AppDelegate` examinée à la Section 4, elle possède un fichier de définition, *GlassAppDelegate.h*, et un fichier d'implémentation, *GlassAppDelegate.m*.

La classe `GlassAppDelegate` des applications PhoneGap n'est pas seulement un délégué de l'application, mais également un délégué pour tous les types de comportements. Les fichiers *.h* et *.m* sont donc beaucoup plus complexes.

Dans le code suivant, vous constatez que la classe `GlassAppDelegate` est un délégué pour l'affichage `WebView`, le gestionnaire de localisation GPS, l'accéléromètre et autres. En effet, ces délégués sont présents sous forme d'une liste dans la déclaration de l'interface, qui débute à la ligne 16.

```
1  #import <UIKit/UIKit.h>
2  #import <CoreLocation/CoreLocation.h>
3  #import <UIKit/UINavigationController.h>
4
5  #import "Vibrate.h"
6  #import "Location.h"
7  #import "Device.h"
8  #import "Sound.h"
9  #import "Contacts.h"
10
11
12 @class GlassViewController;
13 @class Sound;
14 @class Contacts;
15
16 @interface GlassAppDelegate : NSObject <
17 UIApplicationDelegate,
18 UIWebViewDelegate,
19 CLLocationManagerDelegate,
20 UIAccelerometerDelegate,
21 UIImagePickerControllerDelegate,
22 UIPickerViewDelegate,
23 UINavigationControllerDelegate
24 >
25 {
```

```
26
27
28    IBOutlet UIWindow *window;
29    IBOutlet GlassViewController *viewController;
30    IBOutlet UIWebView *webView;
31    IBOutlet UIImageView *imageView;
32    IBOutlet UIActivityIndicatorView *activityView;
33
34    CLLocationManager *locationManager;
35    CLLocation *lastKnownLocation;
36
37    UIImagePickerController *imagePickerController;
38
39    NSURLConnection *callBackConnection;
40    Sound *sound;
41    Contacts *contacts;
42    NSURL* appURL;
43 }
44
45 @property (nonatomic, retain) CLLocation *lastKnownLocation;
46 @property (nonatomic, retain) UIWindow *window;
47 @property (nonatomic, retain) GlassViewController *viewController;
48 @property (nonatomic, retain) UIImagePickerController
49    *imagePickerController;
50
51 - (void) imagePickerController:(UIImagePickerController *)picker
52    didFinishPickingImage:(UIImage *)image2 editingInfo:(NSDictionary
53    *)editingInfo;
54 - (void) imagePickerControllerDidCancel:(UIImagePickerController
55    *)picker;
56 @end
```

Bien que la classe `GlassAppDelegate` soit plus complexe, elle est comparable à la classe `Exemple2_Chapitre1AppDelegate` de la section précédente. Elle joue le rôle de délégué pour l'application et d'autres types d'événements, alors que l'implémentation QuickConnectiPhone utilise la classe `QuickConnectViewController` comme délégué pour tous les événements autres que ceux du délégué de l'application.

La méthode `applicationDidFinishLaunching` est comparable à celle de la classe `Exemple2_Chapitre1AppDelegate`. Pour que ce soit plus clair, seule une partie du code source de la méthode `applicationDidFinishLaunching` de PhoneGap est donnée ci-après. Le code restant sera étudié en détail à la Section 8 et au Chapitre 7.

```
1  -(void)applicationDidFinishLaunching:
     ...
2      webView.delegate = self;
```

Openmirrors.com

```
    ...
3     [window addSubview:viewController.view];
    ...
4 }
```

La ligne 2 est intéressante. À l'instar de la version de `Exemple2_Chapitre1AppDelegate` étudiée à la Section 4, elle affecte un objet `UIWebView` comme vue secondaire de la fenêtre principale. Autrement dit, l'objet `UIWebView` est utilisé comme zone d'affichage pour l'application.

Puisque vous connaissez à présent l'implémentation de la principale méthode déléguée de l'application dans QuickConnectiPhone et PhoneGap, vous êtes prêt à comprendre l'utilisation de la classe `UIWebView` pour afficher et exécuter une application JavaScript.

Section 6 : embarquer du contenu web avec QuickConnectiPhone

Pour afficher du contenu web, par exemple des applications JavaScript ou de simples pages web, dans votre application, vous devez utiliser la classe `UIWebView`. Toutes les mises en œuvre des applications hybrides, que ce soit avec QuickConnectiPhone ou PhoneGap, emploient cette classe. Si vous souhaitez vous amuser avec les polices de caractères dans une application, par exemple plusieurs polices, dans différentes tailles et couleurs, vous devez utiliser `UIWebView`, à moins que vous ne souhaitiez dessiner le texte vous-même. La classe `UIWebView` est simple d'utilisation car elle sait comment interpréter le contenu HTML et CSS, ainsi que le code JavaScript. Cela permet de créer facilement des présentations textuelles, ou d'autres types, complexes.

`UIWebView` est en réalité une enveloppe autour du moteur de rendu WebKit utilisé dans le navigateur Safari, dans Adobe Air, dans Android, dans les téléphones Nokia et dans plusieurs autres applications, notamment celles livrées avec Mac OS X, comme Mail. Dashcode est également un grand utilisateur du moteur WebKit.

Nous l'avons mentionné dans les deux sections précédentes, pour qu'une vue web soit incluse dans une application l'objet `UIWebView` doit être ajouté comme vue secondaire d'une autre vue de l'application. Pour cela, la méthode `loadView` de la classe `QuickConnectViewController` est invoquée.

La méthode `loadView` contient différents éléments qui permettent d'exprimer des comportements dans une application JavaScript. Par exemple, elle fournit le code qui redimensionne l'interface utilisateur de l'application de manière à l'adapter à la taille de l'écran.

Cette possibilité est désactivée par défaut, car l'interface utilisateur doit être initialement conçue à la taille adéquate.

La partie intéressante de `loadView` permet l'affichage de l'interface conçue dans Dashcode précédemment dans ce chapitre. L'extrait de code suivant montre comment l'iPhone insère ce contenu dans l'application. Il commence par calculer la taille et le point d'origine pour l'affichage de l'objet `UIWebView`. Pour cela, il obtient la taille et l'emplacement du cadre d'affichage de l'application.

La variable `webFrame`, dont le type est une structure `CGRect`, contient ces informations obtenues par l'envoi du message `applicationFrame` à l'écran principal de l'application. La structure `CGRect` est constituée de deux éléments : un `CGPoint`, nommé `origin`, qui représente les coordonnées X et Y du point supérieur gauche, et un `CGSize`, qui représente la taille du rectangle à exprimer sous forme d'une hauteur et d'une largeur :

```
CGRect webFrame = [[UIScreen mainScreen] applicationFrame];
webFrame.origin.y -= 20.0;
```

Les coordonnées X et Y, la largeur et la hauteur d'un `CGRect` sont des nombres réels utilisés pour enregistrer un nombre de pixels. La seconde ligne du code précédent montre comment changer la position verticale courante enregistrée dans la variable `webFrame`. Elle décale l'origine vers le haut de vingt pixels. Cette opération est nécessaire pour recouvrir un espace laissé vide dans la vue en raison de l'absence d'une barre d'outils en haut de la fenêtre d'affichage.

Cette barre d'outils est visible dans de nombreuses applications standard, comme l'application Réglages utilisée pour configurer l'iPhone. Elle a été retirée des modèles afin d'augmenter la place disponible sur l'écran pour l'application. Si vous souhaitez disposer des boutons Suivant et Précédent proposés par cette barre d'outils, vous devez la créer dans votre application à l'aide de Dashcode.

Après avoir enregistré dans la variable `webFrame` l'emplacement et la taille souhaités pour l'affichage du contenu web, elle est utilisée pour initialiser un objet `UIWebView` nommé `aWebView`. Les lignes 1 et 2 du code suivant montrent cette opération. Notez qu'elle ressemble à l'allocation de `QuickConnectViewController` examinée précédemment dans ce chapitre. Les principales différences sont l'envoi du message `alloc` à la classe `UIWebView` et l'envoi du message `initWithFrame` à l'objet `UIWebView` qui vient d'être alloué, avec le passage de la structure `webFrame` qui a été créée et modifiée dans l'extrait de code précédent. L'objet `aWebView` est positionné et dimensionné conformément aux valeurs contenues dans `webFrame`.

```
1 UIWebView *aWebView = [[UIWebView alloc]
2                                initWithFrame:webFrame];
```

Openmirrors.com

```
3 self.webView = aWebView;
4 aWebView.autoresizesSubviews = YES;
5 aWebView.autoresizingMask=(UIViewAutoresizingFlexibleHeight
6                             | UIViewAutoresizingFlexibleWidth);
7 // Fixer le délégué du WebView à lui-même.
8 [aWebView setDelegate:self];
```

Le nouvel objet `UIWebView` est enregistré dans l'attribut `webView` de `QuickConnectViewController` par le code de la ligne 3 afin qu'il soit possible d'y accéder ultérieurement depuis d'autres méthodes de `QuickConnectViewController`. Ce point est essentiel pour l'utilisation de l'accéléromètre, de la localisation GPS et des autres possibilités décrites au Chapitre 4.

Les lignes 5 à 6 illustrent la capacité de l'objet `aWebView` à se redessiner lui-même. Si vous le pouvez, évitez l'ajout de vues secondaires. La ligne 4 stipule que, si `aWebView` change de taille, les vues secondaires doivent également être redimensionnées. La syntaxe employée précise que, si la largeur de `aWebView` change en raison d'une rotation, les vues secondaires qu'il contient doivent également changer de largeur d'un facteur identique.

Les lignes 5 et 6 indiquent que la largeur et la hauteur de `aWebView` seront également modifiées. Lorsque l'iPhone est basculé, il est fréquent de passer la vue courante en mode paysage, ou de l'en sortir, et de la redimensionner pour qu'elle corresponde aux nouvelles largeur et hauteur de l'appareil. Si les lignes 5 et 6 étaient retirées ou placées en commentaires, l'application basculerait toujours, mais la largeur et la hauteur de `aWebView` ne seraient pas affectées. Une grande zone vide apparaîtrait alors à droite de l'application en mode paysage. Il est rare de trouver des applications qui basculent sans se redimensionner.

La ligne 8 envoie à `aWebView` un message pour lui indiquer que l'objet `QuickConnectViewController` courant, connu sous le nom `self`, joue le rôle de délégué de l'objet `aWebView`. Cela permet d'implémenter plusieurs méthodes facultatives de `UIWebViewDelegate` dans la classe `QuickConnectViewController`. Le Tableau 1.1 recense ces méthodes.

Si vous en avez besoin, vous pouvez ajouter chacune de ces méthodes facultatives à la classe `QuickConnectViewController`. Le modèle a déjà ajouté `webView:shouldStartLoadWithRequest`, `webView:DidStartLoad`, `webView:DidFinishLoad` et `webView:didFailLoadWithError`.

`aWebView` étant prêt, il est temps à présent d'indiquer le contenu qui doit être chargé et de déclencher ce chargement. Pour cela, l'emplacement du fichier *index.html*, qui fait partie des ressources de l'application, doit être déterminé. Heureusement, comme le montrent les lignes 3 et 4, la classe `NSBundle` qui représente l'application sur le disque dispose d'une méthode nommée `pathForResource:ofType`.

Tableau 1.1 : L'API de UIWebView

Signature de la méthode	*Invocation*	*Paramètres*
`-(BOOL)webView:(UIWebView *) webView shouldStartLoad-WithRequest:(NSURLRequest *) request navigationType: (UIWebViewNavigationType) navigationType`	Juste avant que la vue ne commence à charger le contenu.	`webView` – la vue qui va charger le contenu. `request` – l'emplacement du contenu à charger. `navigationType` – le type d'action utilisateur qui déclenche le changement de la page. options de `UIWebViewNavigationType` – `LinkClicked`, `FormSubmitted`, `BackForward`, `Reload`, `FormResubmitted` et `Other`.
`- (void)webViewDidStart-Load:(UIWebView *) webView`	Après que la vue a commencé le chargement du contenu.	`webView` – la vue qui charge le contenu.
`- (void)webViewDidFinish-Load:(UIWebView *) webView`	Après que la vue a terminé avec succès le chargement du contenu.	`webView` – la vue qui charge le contenu.
`- (void)webView:(UIWebView *) webView didFailLoadWith-Error:(NSError *) error`	Si la vue n'a pas réussi à charger le contenu.	`webView` – la vue qui tente de charger le contenu. `error` – un objet qui représente l'erreur générée.

La méthode `pathForResource:ofType` prend deux chaînes de caractères en argument. La première correspond au nom du fichier (la chaîne `"index"`) et la seconde correspond à l'extension du fichier (la chaîne `"html"`). Cet appel génère le chemin complet du fichier sur votre machine et l'enregistre dans la variable locale `filePathString`. Ce chemin est ensuite utilisé pour créer un objet qui représente une URL vers le fichier, puis un objet `aRequest` de type `NSURLRequest` qui représente l'élément à charger (voir les lignes 7 et 8).

```
1 // Déterminer le chemin du fichier index.html dans le
2 // répertoire Resources.
3 NSString *filePathString = [[NSBundle mainBundle]
4             pathForResource:@"index" ofType:@"html"];
5 // Construire l'URL et la requête pour le fichier index.html.
6 NSURL *aURL = [NSURL fileURLWithPath:filePathString];
```

Openmirrors.com

```
 7 NSURLRequest *aRequest = [NSURLRequest
 8                             requestWithURL:aURL];
 9 // Charger le fichier index.html dans la vue web.
10 [aWebView loadRequest:aRequest];
11 // Ajouter la vue web à la vue de contenu.
12 [contentView addSubview:aWebView];
```

À la ligne 6, l'objet `NSURL` reçoit le message `fileURLWithPath`. Puisqu'un fichier est chargé directement depuis le disque, ce message est bien adapté. Cela suffit pour les applications hybrides QuickConnectiPhone, mais, si vous utilisez une autre implémentation et chargez une page directement depuis le Web, le message doit être `URLWithString`, avec en paramètre une URL complète du type `http://www.byui.edu`.

Après avoir créé l'objet `NSURLRequest`, le chargement réel de l'URL est déclenché par l'envoi du message `loadRequest` à l'objet `aWebView` de type `UIWebView`. L'objet `NSURLRequest`, représenté par la variable `aRequest`, est passé comme seul paramètre de ce message.

Après le chargement de la requête, `aWebView` est ajouté à la vue principale de contenu en lui envoyant le message `addSubview` avec l'objet `UIWebView` en paramètre. Si cet appel n'est pas effectué, la page est chargée et pleinement active, mais elle n'est pas affichée.

Section 7 : embarquer du contenu web avec PhoneGap

Contrairement à QuickConnectiPhone, PhoneGap définit l'emplacement d'un fichier HTML dans la méthode déléguée `applicationDidFinishLaunching` présentée à la Section 5. Néanmoins, une grande partie de la procédure d'affichage du contenu web dans l'application reste identique.

Tout comme dans le cas de QuickConnectiPhone décrit à la section précédente, PhoneGap doit obtenir un chemin vers un fichier dans le paquetage de distribution de l'application. Cette fois-ci, le fichier se nomme *url.txt* à la place du fichier *index.html* de QuickConnect. Cette opération est réalisée aux lignes 8 à 12 du code ci-après.

Tout d'abord, et comme à la section précédente, l'objet `NSBundle` qui représente l'application sur le disque est créé. Le message `pathForResource` lui est ensuite envoyé avec les valeurs `url` et `txt` en paramètres. Si le chargement de ce fichier réussit, la chaîne de caractères contenue dans le fichier *url.txt* est affectée à la variable locale `theURLString` (lignes 10 à 12).

```
1  NSString * htmlFileName;
2  NSString * urlFileName;
3  htmlFileName = @"index";
```

```
 4 urlFileName = @"url";
 5 NSString * urlPathString;
 6 NSBundle * thisBundle = [NSBundle bundleForClass:
 7                                [self class]];
 8 if (urlPathString = [thisBundle
 9     pathForResource:urlFileName ofType:@"txt"]) {
10     NSString * theURLString =
11         [NSString stringWithContentsOfFile:
12             urlPathString];
13     appURL = [NSURL URLWithString:theURLString];
14     [appURL retain];
15     NSURLRequest * aRequest =
16         [NSURLRequest requestWithURL:appURL];
17     [webView loadRequest:aRequest];
18 }
```

La ligne 13 convertit la chaîne lue depuis le fichier *url.txt* en un objet NSURL qui sert à créer une requête. Nous l'avons vu à la section précédente, cette requête est passée en paramètre à `webView` en utilisant le message `loadRequest`.

Par ces deux implémentations du même comportement, vous pouvez constater que, malgré leurs légères différences, elles sont quasiment identiques. Toutes les mises en œuvre d'applications hybrides utilisent l'approche suivante :

- obtenir une chaîne d'URL ;
- créer un `NSURL` à partir de la chaîne ;
- créer un `NSURLRequest` à partir du `NSURL` ;
- utiliser le message `loadRequest` de `UIWebView` avec le `NSURLRequest` en paramètre.

Si vous décidez d'écrire votre propre implémentation, vous devez respecter cette procédure.

En résumé

Pour créer des applications hybrides pour l'iPhone, vous avez besoin d'une petite enveloppe Objective-C pour l'application HTML, CSS et JavaScript. Dashcode est un outil puissant qui permet de créer rapidement et facilement une application JavaScript dynamique, que vous pouvez embarquer en utilisant cette enveloppe. Les modèles d'application QuickConnectiPhone pour Dashcode et Xcode, ainsi que le modèle PhoneGap pour Xcode, accélèrent la création d'une application en incluant dans votre projet le code répétitif employé dans toutes les applications hybrides. Comme le montrent les Chapitres 3, 4 et 6 à 8, les modèles Xcode apportent le code Objective-C et JavaScript dont vous avez

Openmirrors.com

besoin pour écrire des applications hybrides qui exploitent les possibilités suivantes en JavaScript :

- PhoneGap
  - données de l'accéléromètre ;
  - données de localisation GPS ;
  - vibreur de l'appareil.
- QuickConnectiPhone
  - données de l'accéléromètre ;
  - données de localisation GPS ;
  - vibreur de l'appareil ;
  - sons système personnalisés ;
  - enregistrement et lecture audio ;
  - affichage des dates standard et sélecteurs de date et d'heure ;
  - accès aux bases de données SQLite livrées avec l'application et à celles de `UIWebView` lors de l'exécution de l'application.

Grâce aux modèles Dashcode et Xcode, vous pouvez créer des applications pour l'iPhone plus rapidement que jamais.

Openmirrors.com



2

Modularité JavaScript

Lorsque l'on mentionne JavaScript, deux caractéristiques viennent habituellement à l'esprit : compatibilité internavigateurs et complexité. Ce chapitre explique comment éviter la complexité dans les applications hybrides pour l'iPhone et apporte le code source qui permet de mettre en œuvre facilement et rapidement un comportement complexe, sans remettre en cause la flexibilité. Avec les applications hybrides pour l'iPhone, la compatibilité internavigateurs n'est pas un problème car seul le moteur WebKit de Safari est utilisé. Il est ainsi beaucoup plus facile d'écrire des applications JavaScript intéressantes et amusantes.

Section 1 : modularité

Le concept de modularité existe depuis longtemps, que ce soit dans l'informatique ou dans d'autres secteurs industriels. L'essence de la modularité se trouve dans la phrase "construire à partir de pièces interchangeables". Si les pièces sont des modules réellement interchangeables, elles doivent être capables de prendre la place d'une autre sans que cela nécessite, ou presque, une modification des éléments qui interagissent avec elles. Dans le domaine du logiciel, il s'agit généralement d'une API commune qui ne change pas.

L'industrie du divertissement aurait quelques soucis si chaque film était produit sur un support différent, car un système de lecture différent serait alors nécessaire pour chaque film. Si un fabricant d'automobiles ne standardisait pas la liaison entre le moteur et la boîte de vitesses, chaque combinaison moteur-boîte devrait être réalisée à la main. Les coûts monteraient en flèche et la qualité en pâtirait. Dans l'industrie du logiciel, les tentatives pour créer du code modulaire réutilisable ont été nombreuses. Aujourd'hui, elles prennent la forme de frameworks.

Définition d'un module

Pour qu'un module existe, il doit présenter deux caractéristiques : cohésion forte et couplage faible.

Une cohésion forte signifie que le module a un rôle clairement défini et qu'il fait le nécessaire pour le jouer. Il existe pour remplir un objectif, comme gérer une activité, et il agit en ce sens.

Un couplage faible signifie que le module ne dépend pas d'une connaissance du fonctionnement interne d'autres modules et qu'aucun autre module ne connaît le sien. Pour y parvenir, il faut créer et utiliser une interface solide.

Lorsque ces deux caractéristiques sont là, un module est né.

L'étude des frameworks est intéressante. En général, un compromis a été fait entre la facilité d'utilisation du framework et sa flexibilité. Si le développeur du framework n'est pas attentif, il peut obtenir un framework avec lequel la mise en œuvre des choses non importantes est facile, tandis que celle des besoins de l'ingénieur ou du programmeur est difficile.

Souvent, pour que le framework soit simple d'emploi et flexible, son extensibilité est sacrifiée, ce qui est le cas de Ruby on Rails. Ce framework est vraiment génial, mais il s'adapte mal à un environnement d'entreprise sans passer par la mise en place d'un matériel en cluster. Sa facilité d'utilisation est donc réduite et les coûts augmentent. Alors, comment un framework peut-il être extensible, facile à utiliser et flexible ? La réponse se trouve dans une modularité parfaitement appliquée et étudiée.

Bien qu'ils ne soient pas toujours enseignés ou révélés, certains types de modules permettent de faciliter le développement de logiciels. Ces modules, quelque peu secrets, sont connus sous les termes contrôleurs frontaux et contrôleurs d'application.

Les exemples de ce chapitre montrent comment créer et utiliser ces modules et comment ils permettent de faciliter et d'accélérer le développement d'une application.

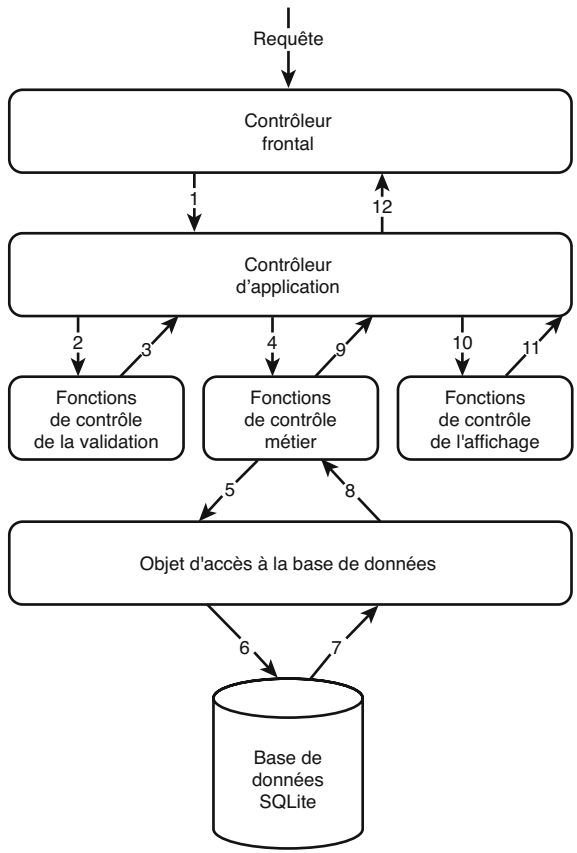
Openmirrors.com

Section 2 : modularité avec le framework JavaScript QuickConnect

Dans les modèles Dashcode et Xcode, le framework JavaScript est conçu pour minimiser l'utilisation du processeur et de la mémoire, tout en restant facile à employer. Puisqu'il est conçu de manière hautement modulaire, chaque composant réalise une chose, la fait bien et la fait rapidement.

La conception se fonde sur un paradigme de commande-réponse. Lorsque vous envoyez une commande, les modules exécutent les fonctions nécessaires qui lui sont associées. La Figure 2.1 illustre ce flux de traitement dans une application dont la conception se fonde sur cette approche. Le traitement commence à l'étape 1 et se poursuit tout au long du framework en respectant la numérotation des flèches.

Figure 2.1
Le flux de traitement associé à une seule commande.



Les seuls éléments du flux qui ne sont pas déjà créés sont les différentes fonctions de contrôle propres au fonctionnement de l'application. Des exemples de ces modules applicatifs spécifiques seront donnés tout au long de cet ouvrage.

L'examen des données saisies par l'utilisateur se fait par l'intermédiaire des fonctions de contrôle de la validation (ValCF, *Validation Control Function*). Les fonctions de contrôle métier (BCF, *Business Control Function*) servent à obtenir des données à partir d'une base de données, d'un serveur web ou d'une autre source, ou à enregistrer des données. Les fonctions de contrôle de l'affichage (VCF, *View Control Function*) sont employées pour mettre à jour la vue présentée à l'utilisateur.

Prenons un exemple. Supposons que vous souhaitiez collecter des informations sur l'utilisateur actuel à partir d'un formulaire qui comprend un bouton Envoyer. Ces informations sont ensuite placées dans une base de données SQLite et l'utilisateur est informé du succès de cet enregistrement. Pour cela, vous devez créer trois fonctions de contrôle : une ValCF pour garantir que les données saisies répondent aux standards définis par l'application, une BCF pour enregistrer les informations dans la base de données et une VCF pour déclencher l'affichage des messages de succès.

Toutes les fonctionnalités ne sont pas nécessairement associées aux trois types de fonctions de contrôle. Par exemple, une application de jeu n'a pas forcément besoin d'une ValCF chaque fois que l'utilisateur déclenche un comportement.

La Figure 2.1 montre que ces différentes fonctions de contrôle n'ont pas besoin de communiquer l'une avec l'autre. Les modules du framework sont conçus pour un tel fonctionnement. Vous devez simplement écrire les fonctions de contrôle et les associer à des commandes. Grâce à cette conception, chaque fonction de contrôle comprend seulement quelques lignes de code et est immédiatement opérationnelle.

Puisque la conception est modulaire, vous pouvez facilement appliquer le concept de division du travail. Si vous répartissez la création de ces fonctions de contrôle au sein d'une équipe, que ce soit en fonction de la commande ou de leur type, les développements peuvent se faire rapidement en parallèle. Pour de plus amples informations concernant ces fonctions de contrôle et leur création, consultez les Sections 4 et 5.

Lorsqu'elles sont bien écrites, les fonctions de contrôle peuvent être utilisées pour plusieurs commandes. Par exemple, vous pourriez définir plusieurs commandes pour actualiser la même partie de l'écran. Avec une telle conception, il est possible d'associer une VCF qui actualise cette partie de la vue pour toutes les commandes.

Le Tableau 2.1 montre que le contrôleur frontal de l'application représente la passerelle au travers de laquelle toutes les requêtes d'exécution doivent transiter. En obligeant toutes les

Openmirrors.com

requêtes à passer par ce contrôleur, il devient beaucoup plus facile de prédéfinir l'ordre des exécutions dans l'application.

Tableau 2.1 : L'API du contrôleur frontal

Méthode	*Valeur de retour*	*Paramètres*
`handleRequest(aCmd, paramArray)`	`void`	`aCmd` – une chaîne unique qui représente le comportement à traiter, par exemple "displayBlogEntries". `paramArray` – un paramètre facultatif constitué d'un tableau de variables qui peuvent être requises pour le traitement.

Le contrôleur frontal est comparable à l'enceinte qui entoure une ville fortifiée : il n'existe qu'une porte d'entrée et qu'une porte de sortie. En limitant les points d'accès possibles à la ville, la défense est plus facile et les citoyens peuvent vivre en meilleure sécurité. L'ajout d'un contrôleur frontal à votre application permet de la sécuriser plus facilement.

Dans le framework QuickConnectiPhone, le contrôleur frontal est mis en œuvre par la fonction `handleRequest`. Elle est définie dans le fichier *QuickConnect.js*, qui, dans Xcode, fait partie du groupe `QCiPhone` des ressources d'application et, dans Dashcode, se trouve dans le dossier *QCiPhone*. Si vous examinez le code, vous verrez comment les fonctions de sécurité et d'ordre d'exécution sont mises en œuvre.

Lors de son invocation, la fonction `handleRequest` reçoit une commande et un tableau de paramètres. La commande est obligatoire, contrairement au tableau de paramètres.

Le code suivant, qui correspond aux lignes 17 à 20 du fichier *functions.js* de l'application simpleCalc, illustre l'invocation de la fonction `handleRequest` en réponse à une action de l'utilisateur. Dans ce cas, l'utilisateur a cliqué sur le bouton qui représente l'addition (voir Figure 2.2).

```
function add(event) {
    handleRequest('math',new Array('+'));
}
```

La commande `math` est passée en premier paramètre et un tableau contenant uniquement le caractère `+` est passé en second paramètre. Dans ce cas, l'utilisation d'un tableau en second paramètre peut sembler inutile, mais la conception impose ce paramètre et cette solution est beaucoup plus flexible, comme vous le verrez plus loin.

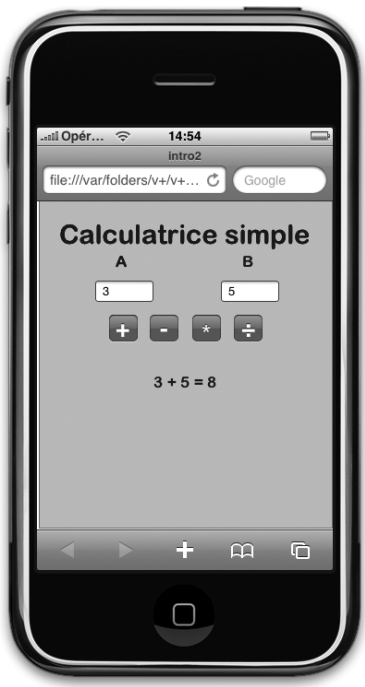


Figure 2.2
L'application simpleCalc après un appui sur le bouton d'addition.

La fonction `add` correspond au gestionnaire `onclick` d'un bouton. Lors d'un clic sur le bouton correspondant, toutes les ValCF, BCF et VCF associées à la commande `math` sont exécutées, avec le tableau de paramètres en argument. Vous remarquerez que les gestionnaires `subtract`, `multiply` et `divide` emploient la même commande que la fonction `add`, mais passent un caractère différent dans le tableau.

Dans notre exemple, l'application réutilise le même code de ValCF, BCF et VCF pour chaque fonctionnalité. Il serait possible d'utiliser une commande différente dans chaque gestionnaire avec différentes BCF et de réutiliser ensuite les mêmes ValCF et VCF, mais les différentes BCF requises pour chaque type d'opérations arithmétiques seraient très semblables. Nous avons donc choisi de créer une seule BCF.

La Figure 2.3 illustre le flux de commande dans l'application `simpleCalc`, après que l'utilisateur a cliqué sur l'un des boutons d'opération arithmétique. Dans cet exemple, deux ValCF sont exécutées pour déterminer si le traitement peut se poursuivre. N'oubliez pas que la conception modulaire montrée ici fait respecter l'ordre des appels de fonctions.

Openmirrors.com

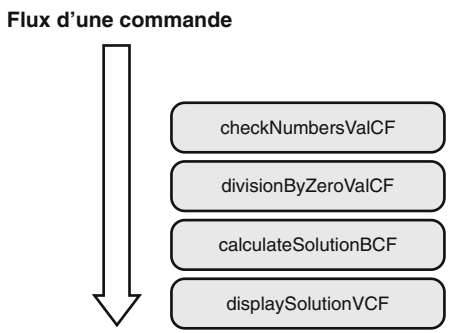


Figure 2.3
Ordre d'exécution des fonctions de contrôle associées à la commande math.

La première ValCF vérifie que les deux valeurs saisies par l'utilisateur sont des nombres. La seconde, `divisionByZeroValCF`, vérifie qu'une division par zéro ne peut pas se produire.

Après la validation, la fonction `calculateSolutionsBCF` est invoquée. Cette BCF effectue l'opération arithmétique demandée par l'utilisateur. La fonction `displaySolutionVCF` affiche ensuite le résultat (voir Figure 2.2).

En revanche, si la requête ne franchit pas l'une des ValCF, la conception modulaire présentée dispose de fonctions de contrôle pour gérer cette situation.

Les fonctions de contrôle des erreurs (ECF, *Error Control Function*) sont des fonctions de contrôle utilisées pour traiter les cas d'erreur. Si l'une des ValCF échoue, la fonction `entryECF` est appelée (voir Figure 2.4). Elle signale à l'utilisateur une erreur dans les valeurs qu'il a saisies.

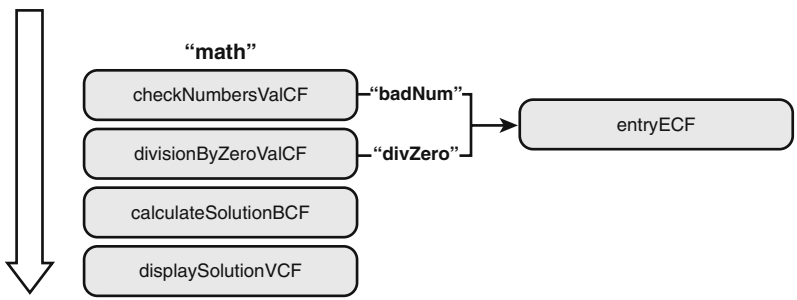


Figure 2.4
La conception du flux de la commande math.

Comment la commande `math` est-elle associée aux quatre fonctions de contrôle qui doivent être exécutées ? QuickConnectiPhone fournit pour cela quatre fonctions utilitaires (voir Tableau 2.2). Chacune d'elles associe une commande à une fonction de contrôle.

Tableau 2.2 : L'API des fonctions d'association

Méthode	*Valeur de retour*	*Paramètres*
`MapCommandToValCF(command, validationControlFunction)`	`void`	`command` – une chaîne unique qui représente le comportement à traiter, par exemple `math`. `validationControlFunction` – une fonction de validation à exécuter lorsque la commande est reçue par `handleRequest`.
`MapCommandToBCF(command, businessControlFunction)`	`void`	`command` – une chaîne unique qui représente le comportement à traiter, par exemple `math`. `businessControlFunction` – une fonction métier à exécuter lorsque la commande est reçue par `handleRequest`.
`MapCommandToVCF(command, viewControlFunction)`	`void`	`command` – une chaîne unique qui représente le comportement à traiter, par exemple `math`. `viewControlFunction` – une fonction d'affichage à exécuter lorsque la commande est reçue par `handleRequest`.
`MapCommandToECF(command, errorControlFunction)`	`void`	`command` – une chaîne unique qui représente le comportement à traiter, par exemple `divZero`. `errorControlFunction` – une fonction d'erreur à exécuter lorsque la commande est reçue par `handleRequest`.

Le code suivant, qui correspond aux lignes 24 à 27 du fichier *mappings.js*, montre l'association de la commande `math` aux fonctions de contrôle, ainsi que celle des erreurs `badNum` et `divZero`.

```
// Associer une commande à plusieurs fonctions.
mapCommandToValCF('math',checkNumbersValCF);
mapCommandToValCF('math',divisionByZeroValCF);
mapCommandToBCF('math', calculateSolutionBCF);
mapCommandToVCF('math', displaySolutionVCF);

// Associer plusieurs commandes à une fonction.
mapCommandToECF('badNum', entryECF);
mapCommandToECF('divZero', entryECF);
```

Openmirrors.com

Un contrôleur frontal bien conçu ne doit être écrit qu'une seule fois et doit pouvoir être réutilisé dans plusieurs applications. La conception particulière du contrôleur frontal et du contrôleur d'application présentée dans ce chapitre offre également une solution simple pour concevoir les comportements spécifiques de vos applications.

Puisque cette conception garantit que les fonctions de contrôle de la validation sont exécutées en premier et sont suivies de toutes les BCF et VCF, l'organisation de la conception au niveau des fonctions de l'application devient plus facile (voir Figure 2.4).

Cette conception générale du framework garantit également que les fonctions de contrôle sont exécutées dans l'ordre dans lequel elles sont associées aux commandes. Dans le code précédent, la fonction `checkNumbersValCF` est toujours invoquée avant `divisionByZero-ValCF`.

Contrôleurs d'application

Un contrôleur d'application, fondé sur le pattern de contrôleur d'application standard, est utilisé pour associer des commandes à des fonctionnalités précises. Ainsi, les implémentations du pattern impliquent généralement une mappe dont les clés correspondent aux commandes et les valeurs, aux fonctionnalités cibles.

Avec la conception décrite dans ce chapitre, les valeurs associées aux clés de la mappe sont des listes de fonctions. Cela permet au contrôleur d'exécuter plusieurs fonctions dans l'ordre indiqué et d'augmenter la modularité et la réutilisabilité des cibles de la fonction de contrôle.

Un contrôleur d'application bien conçu permet d'obtenir une application évolutive, car une fonctionnalité peut être ajoutée sans récrire le fonctionnement interne de l'application.

La mise en œuvre du contrôleur d'application de QuickConnectiPhone en est un exemple.

Puisque vous savez à présent comment les commandes sont associées aux fonctions de contrôle, il est temps d'examiner la création de ces dernières. La fonction `checkNumbers-ValCF` est un exemple relativement classique de ValCF. Elle se focalise sur une tâche, c'est-à-dire vérifier uniquement que les valeurs saisies par l'utilisateur sont des nombres. Si la validation échoue, elle invoque un contrôleur d'application, `dispatchToECF`, pour traiter l'erreur :

```
function checkNumbersValCF(parameters) {
    // Vérifier que a et b sont des nombres.
    var a = document.getElementById('a').value;
```

```
    var b = document.getElementById('b').value;
    if(isNaN(a) || isNaN(b)) {
        dispatchToECF('badNum','Saisissez uniquement des nombres.');
        return false;
    }
    return true;
}
```

Les ValCF retournent `true` si la validation est positive, sinon `false`. Cela permet d'arrêter immédiatement le traitement en cas d'échec et, par voie de conséquence, d'augmenter la sécurité de l'application.

Précédemment, la fonction `calculateSolutionBCF` a été associée en tant que BCF à la commande `math`. Comme la plupart des BCF, elle obtient des données sur lesquelles elle opère. Dans ce cas, elle prend ses données dans l'interface utilisateur. Dans d'autres cas, une BCF peut prendre des valeurs dans une base de données ou, en utilisant AJAX, à partir d'un serveur sur Internet. Le code suivant montre l'implémentation de cette BCF :

```
function calculateSolutionBCF(parameters){
    var a = document.getElementById('a').value;
    var b = document.getElementById('b').value;

    if(a == ''){
        a = 0;
    }
    if(b == ''){
        b = 0;
    }

    // Évaluer le résultat de l'opération.
    var expression = a+parameters[0]+b;
    var result = eval(expression);
    return new Array(a, b, result);
}
```

À la place de la valeur booléenne des ValCF, les BCF retournent un tableau d'informations. Il contient toute information que vous choisissez d'inclure. Dans notre cas, les deux valeurs saisies par l'utilisateur et le résultat calculé sont retournées, car ces données seront nécessaires pour générer les informations affichées à l'utilisateur (voir Figure 2.2).

Le calcul réel du résultat se fait avec la fonction JavaScript `eval`. Cette fonction tente d'exécuter une chaîne qui contient un code JavaScript valide. Dans cet exemple, si

Openmirrors.com

l'utilisateur saisit les valeurs `3` et `5` et clique sur le bouton d'addition, la variable `expression` contient la chaîne `"3+5"`. Lorsque cette chaîne est passée à `eval`, cette fonction retourne `8`.

Faites attention lorsque vous utilisez la fonction `eval`, car elle exécute la chaîne, quelle qu'elle soit. Dans l'application `simpleCalc`, cela ne pose pas de problème, car la ValCF vérifie que les valeurs `a` et `b` sont bien des nombres. Si les éléments utilisés dans un appel à `eval` n'étaient pas validés, des utilisateurs pourraient découvrir le fonctionnement interne des applications et avoir des agissements néfastes, comme une attaque par insertion SQL si la BCF accède à une base de données.

`displaySolutionVCF` est un exemple simple de VCF. Comme les autres fonctions de contrôle, elle réalise une seule chose : actualiser le `<div>` qui contient la chaîne représentant l'opération arithmétique effectuée et le résultat calculé.

```
function displaySolutionVCF(data, parameters){
    var result = data[0][0]+' '+parameters[0] +
            ' '+data[0][1]+' = '+data[0][2];
    document.getElementById('display').innerHTML = result;
}
```

En général, les VCF actualisent le contenu HTML de la page principale. Par exemple, une modification majeure de la vue consiste à changer l'intégralité de l'interface graphique. Notre exemple effectue une petite modification en affichant le calcul arithmétique effectué et en ne touchant pas au reste de l'interface graphique.

Les ECF peuvent être simples ou complexes. Elles peuvent modifier des données enregistrées ou être aussi simples que la fonction `entryECF` suivante, qui reçoit une chaîne de caractères en paramètre et l'affiche à l'utilisateur.

```
function entryECF(message){
    document.getElementById('display').innerHTML = message;
}
```

En raison de sa simplicité, cette ECF peut être utilisée par les deux ValCF de notre application en cas d'échec de la validation.

Avec une bonne implémentation de la conception contrôleur frontal-contrôleur d'application, vous pouvez focaliser la conception et l'implémentation de l'application sur le comportement souhaité, non sur les communications et les passages de données entre les fonctions. La section suivante décrit la mise en œuvre d'un contrôleur frontal et de contrôleurs d'application.

Section 3 : conception modulaire dans QuickConnectiPhone

Comme la plupart des autres parties de l'implémentation de la conception décrite dans ce chapitre, la fonction `handleRequest` est courte. Elle et la méthode pour laquelle elle sert de façade occupent vingt-trois lignes. Les lignes qui nous intéressent sont présentées en gras (7 à 21).

Puisque `handleRequest` implémente le contrôleur frontal de la conception examinée à la section précédente, elle est responsable de l'appel des quatre contrôleurs d'application. Chaque contrôleur d'application prend en charge un type de fonction de contrôle. Comme le montre la ligne 7, `dispatchToValCF` est le premier contrôleur d'application invoqué. Le Tableau 2.3 décrit chaque fonction contrôleur d'application.

Tableau 2.3 : L'API du contrôleur d'application

Méthode	*Valeur de retour*	*Paramètres*
`dispatchToValCF (aCmd, paramArray)`	Booléen (`true` en cas de succès, `false` en cas d'échec).	`aCmd` – une chaîne unique qui représente le comportement à traiter, par exemple `displayBlogEntries`. `paramArray` – un paramètre facultatif constitué d'un tableau de variables requises pour le traitement.
`dispatchToBCF(aCmd, paramArray, callBackData)`	Les données prêtes à être affichées ou `stop`. Si `stop` est retourné, le traitement de la commande indiquée s'arrête.	`aCmd` – une chaîne unique qui représente le comportement à traiter, par exemple `displayBlogEntries`. `paramArray` – un paramètre facultatif constitué d'un tableau de variables requises pour le traitement. `callBackData` – un paramètre facultatif généré par le framework si des appels asynchrones sont effectués depuis une BCF associée à la commande.

Openmirrors.com

Tableau 2.3 : L'API du contrôleur d'application *(suite)*

Méthode	*Valeur de retour*	*Paramètres*
`dispatchToVCF(aCmd, data, paramArray)`	`void`	`aCmd` – une chaîne unique qui représente le comportement à traiter, par exemple `displayBlogEntries`. `data` – un tableau constitué de toutes les données générées par les appels aux BCF. `paramArray` – un paramètre facultatif constitué d'un tableau de variables requises pour le traitement.
`dispatchToECF(aCmd, errorMessage)`	`void`	`aCmd` – une chaîne unique qui représente le comportement à traiter, par exemple `databaseDown`. `errorMessage` – un message d'avertissement, journalisé pour le développeur ou affiché à l'utilisateur, qui doit être suffisamment descriptif et utile pour le type d'utilisateur auquel il est destiné.

`dispatchToValCF` appelle toutes les ValCF qui ont été associées à la commande indiquée par la variable `aCmd`. Si les fonctions de validation réussissent, `dispatchToValCF` retourne la valeur booléenne `true` ; dans le cas où l'une des ValCF échoue, elle retourne `false`.

Puisque l'appel à `dispatchToValCF` est placé dans une instruction `if`, la poursuite du traitement de la requête est conditionnée par le retour de la valeur `true`. Si ce n'est pas le cas, la requête entre dans la routine de traitement des erreurs, mais seulement si des ECF ont été associées à la commande principale. Dans l'application `simpleCalc`, une telle association n'existe pas.

```
1  function handleRequest(aCmd, paramArray){
2      requestHandler(aCmd, paramArray, null);
3  }
4
5  function requestHandler(aCmd, paramArray, callBackData){
6      if(aCmd != null){
7          if(dispatchToValCF(aCmd, paramArray)){
8              try{
9                  var data = dispatchToBCF(aCmd,
```

```
10                      paramArray, callBackData);
11                  if(data != 'stop'){
12                      dispatchToVCF(aCmd, data, paramArray);
13                  }
14              }
15              catch(err){
16                  logError(err);
17              }
18          }
19          else{
20              dispatchToECF(aCmd, 'échec de la validation');
21          }
22      }
23  }
```

Après que la requête a passé la validation, le module du contrôleur frontal appelle le contrôleur d'application suivant, `dispatchToBCF`. Cette fonction contrôleur invoque les BCF associées à la commande de manière à obtenir ou à enregistrer des données.

Si une BCF retourne des données autres que `stop`, le contrôleur frontal appelle les VCF associées à la commande. Pour cela, il invoque le troisième contrôleur d'application, `dispatchToVCF`. Pour de plus amples informations concernant les BCF et les VCF, consultez la Section 4.

L'objectif du module du contrôleur frontal mentionné précédemment est d'offrir une solution rapide et simple pour garantir un flux de calcul stable et sécurisé dans l'application. Lorsqu'un flux est employé de manière cohérente, il est beaucoup plus facile de créer et de déboguer les applications. Par conséquent, lorsque vous utilisez la version QuickConnectiPhone de la conception, il est nécessaire d'invoquer la fonction `handleRequest` si vous voulez que l'application fasse quelque chose.

Chaque fonction de contrôle de l'application décrite au Tableau 2.3 joue un rôle distinct et permet la poursuite ou l'arrêt du traitement de la commande en fonction des décisions que vous prenez dans vos ValCF et vos BCF. Elles correspondent à la notion de module car elles sont réutilisables, faiblement couplées et fortement cohésives.

La fonction `dispatchToValCF` est une façade. Du point de vue comportement, elle est identique à la fonction `dispatchToSCF`, mais elles manipulent toutes deux des jeux de données différents. Les ValCF servent à valider la saisie de l'utilisateur. Les fonctions de contrôle de la sécurité (SCF, *Security Control Function*) permettent de garantir que les données obtenues à partir de sources distantes, comme des sites web, ne contiennent pas du code malveillant. En raison de leur objectif semblable, il est préférable de centraliser le code réel et d'utiliser des fonctions de façade pour invoquer la fonction `check` sous-jacente.

Openmirrors.com

Le code suivant, tiré du fichier *QuickConnect.js*, présente la fonction de façade `dispatchToValCF` et la fonction `check` sous-jacente. Vous le constatez, la fonction `dispatchToValCF` invoque `check` en lui passant ses deux paramètres, plus un autre. Ce paramètre supplémentaire est une chaîne qui décrit le type de mappe que la fonction `check` va utiliser. Cette mappe, ou tableau associatif, contient toutes les associations entre les commandes et un tableau de ValCF. Pour créer et associer des commandes à des ValCF, consultez la Section 2 de ce chapitre.

```
function dispatchToValCF(validationCommand, paramArray){
    return check(validationCommand, 'validation', paramArray);
}

/*
 * Cette fonction ne doit pas être appelée directement par le programmeur.
 */
function check(command, type, data){
    var retVal = true;
    /*
     * Exécuter toutes les fonctions par défaut définies qui s'appliquent à toutes
      ➥les commandes.
     */
    var map = securityMap;
    if(type == 'validation'){
        map = validationMap;
    }
    var defaultFuncs = map['default'];
    if(defaultFuncs){
        var numFuncs = defaultFuncs.length;
        for(var i = 0; i < numFuncs; i++){
            retVal = defaultFuncs[i](command, data);
            if(retVal == false){
                break;
            }
        }
    }
    /*
     * Si les fonctions par défaut ont réussi, exécuter celles spécifiques à la
     ➥commande.
     */
    if(retVal == true){
        commandFuncs = map[command];

```

```
33          if(commandFuncs){
34              var numFuncs = commandFuncs.length;
35              for(var i = 0; i < numFuncs; i++){
36                  retVal = commandFuncs[i](data);
37                  if(retVal == false){
38                      break;
39                  }
40              }
41          }
42      }
43      return retVal;
44  }
```

À la ligne 17, le code tente d'obtenir un tableau des fonctions de contrôle associées à la commande `default`. Autrement dit, vous pouvez créer une VCF qui, en l'associant à `default`, sera appelée pour chaque commande envoyée à `handleRequest`.

Une ValCF par défaut classique s'assure qu'une association pour la commande passée existe, que ce soit dans la mappe des BCF, des VCF ou les deux. Si la commande n'existe pas dans ces fonctions, il n'y a aucune raison de poursuivre le traitement de la commande. Par conséquent, elle stoppe alors les contrôles de validité suivants et retourne `false`.

En retournant `false`, elle conduit également la fonction `dispatchToValCF` à retourner `false` immédiatement, ce qui conduit le contrôleur frontal à arrêter le traitement. Cette ValCF détecte les commandes erronées sans association avant qu'elles ne soient sources de problèmes ultérieurs dans l'application.

S'il n'existe aucune ValCF par défaut ou si la commande a passé toutes celles associées à `default`, la fonction `check` se poursuit en obtenant la liste des ValCF associées à la commande spécifique fournie en argument et les exécute dans l'ordre.

Comme expliqué à la Section 2, ce contrôle préalable des saisies de tout type fait partie des objectifs de la fonction `dispatchToValCF` et des ValCF que vous créez. Il permet également de séparer le code de validation et le code d'exécution. Grâce à cette séparation, ces deux parties du code sont parfaitement identifiées et la maintenance de l'application est simplifiée. Il facilite également la création du logiciel en simplifiant le processus de conception (voir Section 2).

Openmirrors.com

Section 4 : implémentation des contrôleurs métier et d'affichage

De ces deux contrôleurs d'application, le plus complexe est le contrôleur métier, tandis que le plus simple est le contrôleur d'affichage simple. La fonction `dispatchToBCF` appelle toutes les BCF associées à une commande, même si une ou plusieurs des BCF effectuent des appels asynchrones. La fonction `dispatchToVCF` est plus simple car elle ressemble fortement à la fonction `dispatchToValCF` et les VCF ne sont jamais asynchrones. Bien que le comportement de ces fonctions soit comparable, leur implémentation est totalement différente.

Nous l'avons expliqué à la Section 2, la fonction de contrôle métier `dispatchToBCF` est invoquée uniquement si les ValCF que vous avez définies et associées indiquent que le traitement peut se poursuivre. Bien que cette fonction se fonde sur des idées équivalentes, elle est bien différente de la méthode `checkValidation`.

La fonction `dispatchToBCF` est constituée de deux parties principales. La première correspond aux lignes 18 à 40 et concerne le tableau `callBackData`. Ce tableau représente les données accumulées au cours d'un appel asynchrone. Si la BCF n'effectue aucun appel asynchrone, ce tableau est vide. Dans le cas contraire, par exemple avec un appel AJAX ou un appel pour consulter une base de données, ce tableau contient les données nécessaires à l'appel des autres BCF associées à la commande. Les lignes 25 et 31 montrent que ce tableau `callBackData` contient les résultats retournés par toutes les BCF invoquées avant l'appel asynchrone et les données générées par celui-ci.

Définition d'asynchrone

En informatique, nous avons coutume de raisonner de manière synchrone. Autrement dit, les choses se font une à la fois, dans un ordre défini. Lors de l'appel à une fonction, on s'attend à ce que chaque étape de la fonction soit exécutée dans l'ordre. Lorsque toutes les étapes sont terminées, la fonction retourne ou non une valeur.

Un comportement asynchrone est bien différent. Il ressemble plus à une partie de football. Dans ce jeu, chaque joueur s'occupe de la fonction attribuée à son poste, quelle que soit l'activité des autres joueurs. Tout autre comportement serait stupide. Imaginez un match dans lequel tous les joueurs attendraient que les autres joueurs aient fini leur action avant de commencer à bouger...

En informatique, un comportement asynchrone signifie qu'une fonction peut jouer son rôle, mais que le traitement continue sans attendre le retour de la fonction.

La ligne 34 ajoute au tableau `results` les données générées par l'appel asynchrone, dans le cas où aucun appel de BCF précédent n'a été effectué. Ainsi, après l'exécution de cette ligne, le reste du code voit le tableau `results` comme si aucun appel asynchrone n'avait eu lieu.

La ligne 37 obtient l'indice de la BCF qui a effectué l'appel asynchrone. Ainsi, le reste du code de la fonction `dispatchToBCF` sait combien de BCF associées à la commande ont déjà été exécutées. Les appels asynchrones constituent une "rupture" dans l'exécution des BCF associées à la commande. La fonction `dispatchToBCF` ne sait pas quelles BCF ont déjà été exécutées, jusqu'à ce qu'un indicateur soit inclus dans les données générées par l'appel asynchrone.

Au Chapitre 7, cela est mis en place par les méthodes `getData`, `setData`, `getNativeData` et `setNativeData` de la classe `DataAccessObject`. Si vous créez votre propre framework qui autorise les appels asynchrones, vous devez créer un code semblable au suivant :

```
 1 function dispatchToBCF(aCmd, paramArray, callBackData){
 2
 3     if(event == null){
 4         event = window.event;
 5     }
 6     if(event != null){
 7         stopDefault(event);
 8     }
 9     window.curCmd = aCmd;
10     if(paramArray){
11         window.globalParamArray = paramArray;
12     }
13     else{
14         window.globalParamArray = new Array();
15     }
16     var results = new Array();
17     window.numFuncsCalled = 0;
18     if(callBackData){
19         if(callBackData[0]){
20             if(callBackData[1]){
21                 var accumulatedDataFromCallback =
22                                         callBackData[1][3];
23                 if(accumulatedDataFromCallback &&
24                         accumulatedDataFromCallback.length > 0){
25                     results = accumulatedDataFromCallback;
26                 }
27             }
```

Openmirrors.com

```
28              if(results.length == 0){
29                  // results doit toujours être un tableau.
30                  // Les crochets ([]) s'en assurent.
31                  results = [callBackData[0]];
32              }
33              else{
34                  results.push(callBackData[0]);
35              }
36          }
37          if(callBackData[1]){
38              window.numFuncsCalled = callBackData[1][1];
39          }
40      }
41      var stop = false;
42      if(aCmd){
43          var commandList = businessMap[aCmd];
44          callFunc = function(data){
45              if(data){
46                  results.append(data);
47                  window.globalBCFResults = results;
48              }
49              if(window.numFuncsCalled < commandList.length){
50                  var funcToCall =
51                          commandList[window.numFuncsCalled];
52                  window.numFuncsCalled++;
53                  var result = null;
54                  try{
55                      result = funcToCall(paramArray,
56                              results);
57                  }
58                  catch(err){
59                      dispatchToECF('runFailure',
60                          err.message);
61                  }
62
63                  if(result != null){
64                      results[results.length] = result;
65                      callFunc();
66                  }
67                  else{
68                      stop = true;
69                  }
70              }
71          }
```

```
72          if(commandList && commandList.length > 0){
73              callFunc();
74          }
75      }
76      if(stop){
77          return 'stop';
78      }
79      return results;
80  }
```

Les lignes 41 à 80 de la fonction de contrôle `dispatchToBCF` illustrent la manière de réaliser trois choses :

- créer et utiliser des fonctions JavaScript anonymes ;
- employer la récursivité en JavaScript ;
- appeler les BCF associées à une commande.

Une fonction anonyme est une fonction qui est créée "à la volée" à l'intérieur d'une autre fonction. Dans le code précédent, la fonction `callFunc`, aux lignes 44 à 71, en est un exemple. Cette fonction n'existe pas en dehors de `dispatchToBCF`. Comme n'importe quelle fonction anonyme, elle est strictement limitée à la portée de la fonction dans laquelle elle est déclarée. Par ailleurs, les variables déclarées dans la fonction conteneur avant la déclaration de la fonction anonyme sont dans la portée et peuvent être utilisées dans la fonction anonyme, même si elles ne sont pas passées en paramètres.

Les lignes 43 et 44 en sont un exemple. La variable `commandList` est définie en dehors de la fonction `callFunc`, elle ne lui est pas passée en paramètre, mais elle est employée dans cette fonction. Elle est utilisée aux lignes 50 et 51 pour obtenir la prochaine BCF à exécuter. Les lignes 55 et 56 exécutent cette BCF et enregistrent les résultats de l'invocation.

Un exemple de récursivité se trouve à la ligne 65, où la fonction `callFunc` s'appelle elle-même. Cela se produit à la fin de la fonction, uniquement si le résultat de l'appel à la BCF n'est pas `null`. Ce type de récursivité est appelé récursivité finale car la vérification se trouve à la fin de la fonction. Si elle se trouvait au début de la fonction, il s'agirait d'une récursivité initiale. La cascade d'appels récursifs est déclenchée par l'appel à `callFunc` à la ligne 73.

Récursivité

La récursivité correspond à l'appel d'une fonction par elle-même.

Openmirrors.com

Lorsque la fonction `dispatchToVCF` est invoquée, une liste des fonctions de contrôle associées à la commande est obtenue (voir le code ci-après). Contrairement à la fonction de contrôle de la validation, `dispatchToVCF` reçoit en argument un tableau contenant les éléments qui correspondent aux résultats des appels à chacune des BCF.

À l'instar de la fonction `dispatchToBCF`, si l'une des VCF retourne `stop`, aucune autre n'est exécutée. Cela permet aux programmeurs de terminer l'exécution en faisant un appel à `dispatchToECF`.

```
function dispatchToVCF(aCmd, data, paramArray){
    if(aCmd){
        var vcfFuncList = viewMap[aCmd];
        if(vcfFuncList == null){
            vcfFuncList = new Array();
        }
        var numFuncs = vcfFuncList.length;
        for(var i = 0; i < numFuncs; i++){
            try{
                retVal = vcfFuncList[i](data, paramArray);
            }
            catch(err){
                debug(errorMessage(err));
            }
            if(retVal && retVal == 'stop'){
                break;
            }
        }
    }
}
```

Après qu'une VCF a été obtenue, elle est invoquée avec les données résultantes de la BCF et les paramètres d'origine, `paramArray`, envoyés par l'application à la fonction `handleRequest`. Cela permet de passer des informations aux BCF et aux VCF invoquées par les contrôleurs d'application métier et d'affichage.

Section 5 : implémentation d'un contrôleur d'erreur

Contrairement aux autres contrôleurs d'application examinés aux Sections 3 et 4, la mise en œuvre du contrôleur d'erreur dans le framework QuickConnectiPhone n'autorise l'association que d'une seule ECF à une commande. Autrement dit, chaque ECF doit gérer intégralement l'erreur, ce qui inclut modifier des données enregistrées, actualiser la vue pour avertir les utilisateurs, etc.

Voici un exemple d'implémentation simple d'un contrôleur d'erreur :

```
function dispatchToECF(errorCommand, errorMessage){
    var errorFunc = errorMap[errorCommand];
    if(errorFunc){
        return errorFunc(errorMessage);
    }
}
```

Cette version obtient simplement l'ECF à exécuter et lui passe le message d'erreur. Pour activer une telle gestion des erreurs, il faut effectuer un appel direct à `dispatchToECF`, sans passer par un appel à `handleRequest`. La ligne de code suivante est extraite de la fonction `checkNumbersValCF` :

```
dispatchToECF('badNum', 'Saisissez uniquement des nombres.');
```

Cet appel à `dispatchToECF` contient le message qui doit être affiché afin d'informer l'utilisateur que seuls les nombres sont acceptés.

Section 6 : étapes de création d'une fonctionnalité de l'application

Les Sections 3 à 5 ont expliqué ce qui se passe en coulisse lorsque vous utilisez l'implémentation des contrôleurs frontaux et d'application proposée par le framework QuickConnectiPhone. Mais quelles sont les étapes à suivre pour mettre en œuvre une fonctionnalité de l'application ? Les voici, dans l'ordre :

1. Créer les BCF pour obtenir ou enregistrer les données (voir Section 4).
2. Créer les VCF pour modifier l'interface utilisateur (voir Section 4).
3. Créer les ValCF nécessaires à la validation de la saisie de l'utilisateur (voir Section 3).
4. Créer les ECF nécessaires à la gestion des conditions d'erreur possibles (voir Section 5).
5. Associer toutes les fonctions de contrôle à une commande en utilisant les fonctions `mapCommandTo*` correspondantes.

Après avoir terminé ces cinq étapes, la nouvelle fonctionnalité est intégrée à l'application.

Openmirrors.com

En résumé

Lorsqu'elle est appliquée correctement, la modularité facilite l'écriture de code, simplifie sa maintenance et réduit sa taille. En gardant cela à l'esprit, ainsi que la rapidité d'exécution, l'implémentation apportée par le framework QuickConnectiPhone à chaque application créée à l'aide de ses modèles facilite énormément votre travail. Vous pouvez vous concentrer sur votre objectif, c'est-à-dire créer et fournir une fonctionnalité à vos utilisateurs. Le framework s'occupe de toutes les communications et des contrôles internes.

En écrivant des ValCF, des BCF, des VCF et des ECF, vous pouvez aisément focaliser votre travail de manière à augmenter la productivité. Dans le même temps, vous augmentez également la qualité et la sécurité de votre code.

Openmirrors.com



3

Interfaces utilisateur

L'iPhone permet de proposer des méthodes d'interaction uniques avec l'utilisateur. Sur ce téléphone, les anciennes méthodes de conception des interfaces ne suffisent plus. Votre application doit offrir les interactions et les éléments d'interfaces que les utilisateurs de l'iPhone réclament. Ce chapitre explique comment procéder. Il s'intéresse également au guide d'interface utilisateur employé par Apple pour décider de l'ajout des applications dans l'App Store. Un module de redimensionnement, de rotation et de glisser-déposer est également décrit afin que vous appreniez à traiter les événements liés au toucher et aux gestes en JavaScript. Ces nouveaux types d'événements JavaScript sont essentiels à la conception des interfaces utilisateur pour l'iPhone.

Section 1 : guide de l'interface utilisateur d'Apple

Pour que tous les utilisateurs de l'iPhone partagent la même expérience, Apple a rédigé un guide qui stipule les règles à suivre lors de la conception des interfaces utilisateur des applications pour l'iPhone. Cette section introduit le HIG (*Human Interface Guide*) pour l'iPhone de manière concise et simple. Sa version intégrale peut être consultée à l'adresse **http://developer.apple.com/iphone/library/documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html**.

Le HIG pour l'iPhone trouve ses origines dans les guides créés précédemment pour OS X et le développement web pour l'iPhone. Ce nouveau guide se fonde sur les points forts des précédents et ajoute de nouveaux aspects spécifiques aux applications pour l'iPhone.

La place disponible sur l'écran de l'iPhone est tellement limitée que l'affichage d'un texte à l'utilisateur peut vite devenir pénible. Pour les applications dont la fonction principale n'est pas de présenter des données textuelles, il est préférable d'employer des images et des icônes pour communiquer des idées et offrir des fonctionnalités à utilisateur.

L'abandon des informations textuelles se trouve au cœur de l'iPhone. Cet appareil ne présente pas de menus déroulants ou de rubans, et vos applications ne doivent pas en utiliser. Si l'application est bien conçue, son utilisation doit se révéler intuitive. Si elle n'est pas intuitive, l'application n'est pas conçue pour l'iPhone. Si elle est bien conçue, elle n'a besoin d'aucun guide de l'utilisateur.

Les menus déroulants ont proliféré car les applications n'ont plus un seul usage mais plusieurs. Si vous examinez les applications bureautiques existantes, il est facile de voir comment elles sont parties d'applications initialement simples, à l'usage bien défini, pour devenir des monstres de fonctionnalités.

Les logiciels de traitement de textes ne sont plus vraiment des traitements de textes. Ils permettent de créer des mises en page sophistiquées, sont capables d'inclure des données provenant d'applications totalement différentes et offrent même un environnement de développement. Si cette enchère de fonctionnalités a permis de conserver la viabilité des applications, celles-ci sont devenues volumineuses et lourdes.

Chaque application pour l'iPhone doit avoir un et un seul objectif ou fonction. Il doit être facile à identifier et facile à maîtriser par l'utilisateur. En respectant les standards d'Apple, vos applications seront plus facilement compréhensibles et apporteront à l'utilisateur une expérience plus plaisante.

Le contrôle de l'application est un élément vital dans toute conception. L'utilisateur doit disposer du plus grand contrôle possible. Autrement dit, l'application ne doit pas le contraindre à des comportements précis. La limitation des options est vue d'un mauvais œil dans les applications pour l'iPhone. Pour arriver à ce résultat, les vues d'une application doivent être organisées à plat. Les profondes hiérarchies de vues augmentent la charge de travail de l'appareil et sont déconseillées.

Puisque les interactions avec l'iPhone se font par des gestes, comme le toucher et le toucher multiple, l'application doit également les prendre en charge. Chaque zone de toucher doit être dimensionnée de manière appropriée pour que l'utilisateur puisse la sélectionner. Normalement, la largeur et la hauteur d'une zone de toucher doivent être égales à 34 pixels. Si elles sont plus petites, par exemple pour gagner de la place sur

Openmirrors.com

l'écran, l'utilisateur risque d'avoir des difficultés à sélectionner les éléments et n'en sera que plus mécontent.

Même si le balayement (*swipe*) et le pincement (*pinch*) sont des comportements pris en charge, les utilisateurs peuvent avoir du mal à savoir qu'ils existent. Si vous souhaitez les inclure, ajoutez des indicateurs visuels qui informent de leur disponibilité. Par exemple, dans une application de type livre électronique, ces indicateurs peuvent prendre la forme de coins de page pliés.

Le glisser-déposer est généralement déconseillé par le HIG, car la même interaction de l'utilisateur sert habituellement au défilement ou au déplacement panoramique. Toutefois, il est facile de trouver des applications réussies qui utilisent cette fonctionnalité. Apple a même écrit en JavaScript un exemple excessivement complexe de ce comportement. La seconde partie de ce chapitre explique comment mettre en œuvre le glisser-déposer, le pincement pour le redimensionnement et la rotation des éléments de l'interface.

Le Tableau 3.1 récapitule les différents types de gestes reconnus et les comportements standard associés. En respectant ces fonctionnements, la phase d'apprentissage de l'application est réduite. En revanche, si vous redéfinissez les comportements associés, elle devient plus longue.

Tableau 3.1 : Gestes reconnus par l'iPhone et comportements standard associés

Geste	*Comportement*
Toucher (*tap*)	Sélection d'un élément de l'interface utilisateur.
Glissement (*drag*)	Défilement ou déplacement panoramique pour un autre affichage.
Défilement (*flick*)	Défilement ou déplacement panoramique rapide. Une fois le geste terminé, le comportement doit se poursuivre.
Balayement (*swipe*)	Révélation des composants cachés, comme des boutons de suppression d'une ligne d'une table ou des vues supplémentaires.
Double-toucher (*double tap*)	Centrage, puis zoom avant ou arrière.
Pincement vers l'extérieur (*pinch open*)	Zoom avant.
Pincement vers l'intérieur (*pinch close*)	Zoom arrière.
Toucher et maintien	Affichage d'une vue agrandie.

L'iPhone est un appareil réellement extraordinaire, mais la saisie d'un texte peut être pénible et lente comparée à l'utilisation d'un clavier standard. Autant que possible, essayez d'utiliser des sélecteurs de données à la place d'une saisie directe. Dans les applications

hybrides, les sélecteurs apparaissent lorsque l'utilisateur active une balise HTML `<option>`. Grâce aux sélecteurs, les utilisateurs maîtrisent plus rapidement votre application et leur niveau de frustration diminue. Le Chapitre 4 explique comment inclure des sélecteurs de date et d'heure dans les applications hybrides.

Le document de standardisation des applications pour l'iPhone stipule que les cases à cocher et les boutons radio doivent être évités. À la place, il est préférable d'utiliser des interrupteurs. L'application Réglages fournie avec chaque iPhone et iPod Touch les utilise énormément. La Figure 3.1 illustre les options de Safari que vous pouvez modifier dans l'application Réglages. Les développeurs d'applications hybrides ont quelques difficultés à éviter les boutons radio et les cases à cocher.

Figure 3.1
Les interrupteurs utilisés pour les réglages de Safari.



L'outil Dashcode utilisé pour développer l'interface graphique des applications hybrides ne dispose pas d'une widget de type interrupteur, mais il est facile de la créer. Elle est constituée d'une boîte avec une bordure interne, deux éléments de texte, 1 et 0, et un bouton.

En utilisant les événements de geste sur le bouton, il est facile de le faire glisser à droite ou à gauche de manière à révéler l'état de l'interrupteur. La fonction de rappel `ongestureend` définie pour le bouton permet de détecter et d'enregistrer son état allumé (1) ou éteint (0). Servez-vous des boutons de réglage de Safari illustrés à la Figure 3.1 pour créer les vôtres.

En respectant ces règles de base pour la conception, votre application satisfait aux exigences d'Apple pour sa distribution sur l'App Store et son fonctionnement correspond aux

Openmirrors.com

attentes de l'utilisateur. Le non-respect de ces concepts risque de conduire au rejet de votre application par Apple et par ses utilisateurs potentiels.

Section 2 : interfaces fondées sur les listes et sur Navigateur

Certaines interfaces utilisateur de base pour l'iPhone se fondent sur les listes pour organiser l'affichage. L'application HistoryExample en est un exemple. Dans Dashcode, cette interface est créée de deux manières. La plus rapide consiste à utiliser la partie `Navigateur`.

La partie `Navigateur`, qui se trouve dans la bibliothèque des parties, peut être déposée directement sur l'application pour créer une pile de vues indépendantes. Les vues constituent l'unité d'affichage principal avec laquelle les utilisateurs interagissent ; elles sont souvent appelées, de manière erronée, écran. Lorsqu'une partie `Navigateur` est ajoutée à une application, deux vues sont ajoutées par défaut, mais pas le code de basculement d'une vue à l'autre. Une barre d'en-tête comprenant un bouton de navigation est également insérée automatiquement. La Figure 3.2 présente l'outil Dashcode après l'ajout de la partie `Navigateur`.

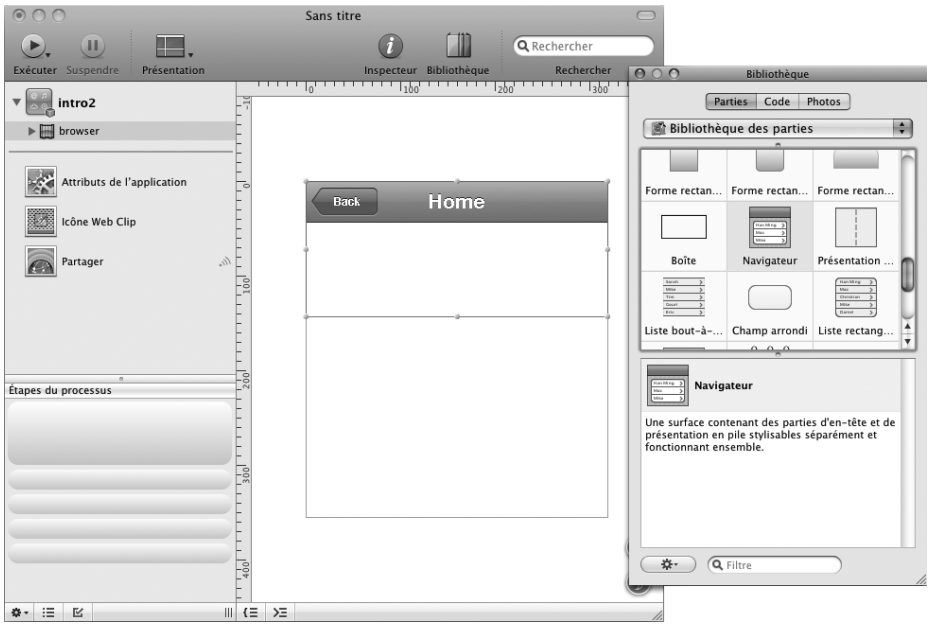


Figure 3.2
Ajout d'une partie Navigateur dans Dashcode.

Il est inutile de modifier le bouton ajouté automatiquement dans l'en-tête, car la partie `Navigateur` ajuste le texte affiché pour qu'il corresponde à l'intitulé de l'en-tête de la vue précédente (si elle existe). Lorsque vous créez une application, vous devez modifier le texte de l'en-tête, non celui du bouton, pour refléter le nom de l'application ou toute autre information appropriée.

Deux vues avec des noms par défaut sont insérées dans le projet lors de l'ajout d'une partie `Navigateur`. En sélectionnant l'onglet `Attributs` dans l'inspecteur, vous pouvez voir ces vues et les renommer. La Figure 3.3 présente l'application HistoryExample après avoir nommé les vues `mainView` et `presidentsView`. Vous remarquerez les options + et – dans la liste `sous-présentations`. Ces boutons vous permettent d'ajouter d'autres vues et de retirer celles dont vous n'avez pas besoin.



Figure 3.3
L'application HistoryExample avec des vues et deux listes rectangulaires arrondies ajoutées à la vue principale.

Toutes les vues de l'application sont gérées depuis la liste `sous-présentations`. Vous remarquerez que, même si la navigation dans l'application correspond à `mainView > ContinentsView > SouthAmericanView`, toutes les vues sont directement des enfants de `stackLayout`. C'est ainsi que les applications de ce type doivent être développées.

Dans les listes rectangulaires arrondies et les listes bout-à-bout, seul le premier élément de la liste peut être sélectionné dans l'écran Dashcode de conception de l'interface graphique.

Openmirrors.com

Ce premier élément sert de modèle pour les autres éléments de la liste. Toute modification de la couleur, du contenu ou de la taille de cet élément est appliquée à tous les autres éléments de la liste. Cela inclut également les gestionnaires d'événements affectés. Tous les éléments des listes présentes dans l'application HistoryExample partagent le même gestionnaire d'événements `onclick`, la fonction `changeView`.

Le code de la fonction `changeView`, qui se trouve dans le fichier *main.js*, est donné ci-après. Elle utilise les étiquettes et les valeurs saisies dans l'écran des attributs de la vue pour déterminer la nouvelle vue à afficher. Pour examiner cette fonction, sélectionnez la vue, non le premier élément d'étiquette. Dans votre application, c'est là où vous ajoutez les éléments statiques de la liste (voir Figure 3.4).

```
function changeView(event)
{
    var subViewName = event.target.object.value;
    var displayName = event.target.innerText;
    var browser = document.getElementById('browser').object;
    browser.goForward(subViewName+'View', displayName);
}
```



Figure 3.4
Affichage des attributs de la liste people de la vue principale, avec deux éléments statiques ajoutés.

En utilisant ces valeurs et en donnant le nom approprié aux vues, la méthode `goForward` de l'objet `browser` permet de passer à la vue suivante. Cette méthode prend deux arguments. Le premier est le nom de la vue vers laquelle basculer, le second, le texte à afficher dans l'en-tête de la vue. Si vous essayez de basculer vers une vue et si l'en-tête est modifié sans que la vue change, cela signifie que le nom de la vue que vous tentez d'afficher ne correspond pas à celui d'une vue de votre application.

Lorsque vous créez des applications fondées sur des listes et des vues, vous devez faire attention à l'organisation des données. Si les informations sont placées dans un trop grand nombre de vues, l'utilisateur trouvera la navigation exagérément compliquée. En réalité, si vous n'y prenez pas garde, vous pourriez réinventer la navigation de type DOS des années 1970 et 1980.

Les applications à base de listes et de vues ont de nombreuses utilisations, mais elles ne sont pas toujours les plus attrayantes visuellement. Parfois, le changement de vues doit se faire à partir d'un élément autre qu'une liste.

Section 3 : applications non fondées sur des listes

Bien que les utilisations des applications fondées sur les listes et les vues soient nombreuses, rien n'oblige toutes les applications fondées sur les vues à utiliser des listes pour accéder aux informations. D'autres indices visuels peuvent indiquer que toucher un élément permet d'afficher des informations supplémentaires. L'application PictureExample fournie avec le framework QuickConnectiPhone en est un exemple.

En indiquant sur l'écran qu'un élément peut être touché, l'utilisateur a tendance à toucher tous les éléments pour savoir s'ils sont actifs. Si vous retenez cette approche, faites attention à ne pas perturber vos utilisateurs par ces indices ou par les éléments qu'ils doivent toucher pour contrôler votre application.

Le code suivant présente deux différences par rapport au code précédent. Tout d'abord, chacune des images a besoin d'un gestionnaire `onclick` ou `ontouchstart` pour déclencher le passage à une vue secondaire. Dans cet exemple, une seule fonction, `goSub`, est utilisée pour gérer les images touchables.

```
function goSub(event)
{
    var stackLayout = document.getElementById('stackLayout').object;
    stackLayout.setCurrentView(event.target.id+'View', false);
}
```

Openmirrors.com

En affectant à chaque image un `id` semblable au nom de la vue qu'il représente, il est facile d'employer une méthode pour changer de vue. L'exemple précédent le montre, l'objet `stackLayout` offre une méthode `setCurrentView` qui prend deux arguments : l'identifiant de la vue vers laquelle basculer et un indicateur. Si l'indicateur vaut `true`, cela signifie que le changement de vue correspond à un retour en arrière. L'identifiant et l'indicateur permettent au programmeur de contrôler les transitions d'une vue à une autre.

Contrairement à l'application HistoryExample constituée de listes, de vues et d'un en-tête de navigation préconstruits, l'application PictureExample ne comprend aucune barre de navigation automatique. C'est pourquoi le programmeur doit écrire le code de retour d'une vue secondaire à la vue principale.

Les images de retour (marquées par Back) dans les vues secondaires sont faites pour revenir à la vue principale. Puisqu'il ne serait pas approprié que l'effet visuel associé à une transition en arrière soit identique à celui d'une transition en avant, la valeur du second paramètre de la fonction `setCurrentView` est fixée à `true`. Il indique une transition visuelle en arrière.

Le code de `goMain` présenté ci-après est associé au bouton `Retour` en tant que gestionnaire `onclick`. Puisque le second argument vaut `true`, la transition associée à `mainView` se fait dans un sens opposé au comportement standard. Nous pouvons ainsi faire en sorte que l'utilisateur pense qu'une action précédente est annulée.

```
function goMain(event)
{
    // Fixez la vue courante d'un StackLayout.
    var stackLayout = document.getElementById('stackLayout').object;
    stackLayout.setCurrentView('mainView', true);
}
```

Par rapport aux applications fondées sur une partie `Navigateur`, l'animation du changement de vues offre d'autres possibilités. Elle constitue un autre élément d'information donné à l'utilisateur. Lorsque l'animation utilisée pour passer à une vue n'est pas identique à celle employée pour d'autres vues, l'utilisateur sait que cette vue a quelque chose de différent. La Figure 3.5 présente la liste des vues secondaires et la fonction `goMain` dans Dashcode.



Figure 3.5
L'onglet des attributs de `stackLayout` *montre les options de la transition.*

Comme le montre le Tableau 3.2, il est possible d'utiliser plusieurs types de transitions dans une application, mais ce n'est pas judicieux.

Tableau 3.2 : Transitions par défaut disponibles dans Dashcode

Type de transition	*Comportement*
Défiler	Une transition bidimensionnelle dans laquelle la nouvelle vue à afficher arrive dans la zone visible pendant que l'ancienne vue la quitte.
Dissolution	Une transition bidimensionnelle dans laquelle la vue à afficher devient plus opaque tandis que l'ancienne vue devient plus transparente. Puisqu'elles se superposent, l'ancienne vue semble se transformer progressivement en la nouvelle vue.

Openmirrors.com

Tableau 3.2 : Transitions par défaut disponibles dans Dashcode *(suite)*

Type de transition	*Comportement*
Diapositive	Une transition bidimensionnelle semblable à Défiler. Dans ce cas, l'ancienne vue reste à sa place pendant que la nouvelle apparaît et la recouvre, pour donner à l'utilisateur l'impression d'une pile de vues. Dans ce cas, une transition "en arrière" correspond à entrer dans les détails des vues de l'application, tandis qu'une transition "en avant" correspond à un retour vers le début de la pile.
Fondu	Une transition bidimensionnelle semblable à Dissolution. Dans ce cas, l'ancienne vue reste opaque afin que les deux vues soient visibles à la fin de la transition. Lorsque cette transition se fait "en arrière", la nouvelle vue devient opaque et la vue d'origine est intégralement affichée. Cette transition sert à ajouter une nouvelle information ou fonctionnalité à une vue existante, car les deux vues peuvent accepter les événements de toucher.
Rotation	Une transition tridimensionnelle qui provoque une rotation le long de l'axe Y de l'appareil par rapport au centre de l'ancienne et de la nouvelle vue. L'utilisateur a l'impression que l'ancienne vue se trouve au premier plan de l'application et que la nouvelle se trouve à l'arrière-plan. Cette transition n'est généralement pas utilisée dans les applications qui comprennent uniquement deux vues.
Cube	Une transition tridimensionnelle dans laquelle l'utilisateur voit toutes les vues sur les côtés d'un cube, qui pivote vers l'avant ou l'arrière.
Échanger	Une transition tridimensionnelle dans laquelle l'ancienne vue semble glisser depuis un côté et passer sous la nouvelle vue. Au cours de la transition, l'arrière-plan de la nouvelle vue est transparent. À la fin de la transition, cet arrière-plan devient opaque.
Faire pivoter	Une transition tridimensionnelle dans laquelle l'ancienne et la nouvelle vue pivotent autour de l'axe Y de l'appareil sur un des bords. L'effet obtenu est comparable à une porte pivotante.

Si l'on en croit l'interface de Dashcode, il semble que les seules directions possibles pour ces transitions soient de droite à gauche et de gauche à droite, mais ce n'est pas le cas. Des mouvements de haut en bas et de bas en haut sont également possibles.

Le code suivant est extrait du fichier *setup.js* de l'application HistoryExample ; il a été généré par Dashcode. Vous le constatez, les types de transitions et leur direction sont indiqués pour toutes les vues. Si vous souhaitez les modifier, vous devez tout d'abord désactiver la génération de code en allant dans le menu Présentation > Arrêter le générateur de code.

```
var dashcodePartSpecs = {
    ...
```

```
    "stackLayout": { "creationFunction": "CreateStackLayout",
"subviewsTransitions": [{ "direction": "right-left", "duration": "", "timing":
"ease-in-out", "type": "push" }, { "direction": "right-left","duration": "",
"timing": "ease-in-out", "type": "push" }, { "direction": "right-left",
"duration": "", "timing": "ease-in-out", "type": "push" }, { "direction":
"right-left", "duration": "", "timing": "ease-in-out", "type": "push" }, {
"direction": "right-left", "duration": "", "timing": "ease-in-out", "type":
"push" }] }
    ...
 };
```

Puisque Dashcode génère une grande partie du code à votre place et remplace régulièrement le contenu du fichier *setup.js*, vous ne devez pas modifier celui-ci tant que l'application n'est pas terminée. Il est plus facile d'effectuer ces modifications après que le fichier a été placé dans le modèle QuickConnectiPhone pour Xcode car Dashcode n'entre plus en jeu et ne peut donc pas écraser les modifications que vous effectuez.

Le code précédent contient la déclaration de quatre objets JavaScript. Chacun commence par le caractère `{`, se termine par `}` et contient les attributs `direction`, `duration`, `timing` et `type`. L'un de ces objets, le deuxième, est présenté en gras afin de le distinguer des autres.

Chacun de ces objets anonymes définit le comportement du passage d'une vue à une autre. L'objet présenté en gras déclare une transition de type Défiler (voir Tableau 3.2). Elle pousse la nouvelle vue de la gauche vers la droite avec une accélération au début de la transition et un ralentissement à la fin (synchronisation `ease-in-out`).

Les autres options de synchronisation sont `ease-in`, `ease-out` et par défaut. La synchronisation par défaut, dans laquelle la vitesse est constante, est utilisée lorsque l'attribut `timing` de l'objet de définition est absent.

Nous l'avons mentionné précédemment, il existe d'autres options pour l'orientation de la transition : de haut en bas (`top-bottom`) et de bas en haut (`bottom-top`). Elles sont valides uniquement pour les transitions de type Diapositive et Défiler.

Si vous choisissez de modifier les déclarations de ces objets, sachez que cela peut provoquer des problèmes dans une application plus complexe. Il semble qu'Apple ait interdit certains choix dans les options de transition car ils provoquent des dysfonctionnements dans le moteur WebKit de Safari et dans l'objet `UIWebView` des applications hybrides.

Openmirrors.com

Section 4 : applications d'immersion

Les applications d'immersion sont en rupture totale avec l'utilisation des vues pour regrouper des informations et des contrôles. Les jeux en sont la concrétisation la plus fréquente, mais certaines applications médicales pour l'iPhone suivent également cette voie. L'idée est que les interactions de l'utilisateur avec l'application soient naturelles, fluides et, si possible, dans une seule vue.

Si les jeux utilisent cette approche sous sa forme extrême, elle peut également être employée d'autres manières. C'est notamment le cas dans les applications d'imagerie médicale, où les possibilités tactiles de l'iPhone changent énormément la manière dont les médecins interagissent avec les images.

Il n'y a aucune raison pour que des développeurs novateurs n'appliquent pas cette solution à des applications d'entreprise ou scientifiques. Souvent, de mauvaises décisions métier sont prises car l'affichage des éléments d'information connexes et complexes avec les graphiques actuels est difficile. Pour examiner les données de manière différente, il faut utiliser une autre forme d'affichage.

Si des données peuvent être présentées sous forme non linéaire sur l'intégrité de l'écran, des informations complémentaires peuvent leur être superposées afin d'identifier des relations. Une version simple de cette méthode est employée dans les applications cartographiques pour non seulement afficher une route menant d'un lieu à un autre, mais également indiquer la densité du trafic sur cette route et les routes voisines. Les deux éléments d'information sont superposés afin que l'utilisateur puisse en déduire un motif utile.

Cet ouvrage ne prétend pas fournir une solution pour la manipulation des données en vue de leur affichage. Il suggère simplement que cette possibilité existe et qu'elle est employée. Dans cette section, nous prendrons l'exemple d'un jeu.

Le jeu DollarStash est une variante de l'application web Leaves d'Apple. Dans cette application, des images de feuilles sont ajoutées sur la page et tombent progressivement vers le bas de l'écran en tourbillonnant. Pour en faire un jeu, nous remplaçons les feuilles par des images de billets.

Lorsque l'utilisateur touche un billet, la quantité d'argent sur son compte est incrémentée. Si un billet disparaît totalement avant que l'utilisateur ne le touche, le solde du compte est décrémenté. Des groupes de billets sont affichés par vagues en haut de l'écran. Chaque vague contient un billet de plus que la précédente. Lorsque le solde du joueur tombe sous zéro, la partie est terminée. La Figure 3.6 montre le jeu en cours d'exécution. Même s'il a été développé rapidement, il révèle les limites des applications de ce type dans un environnement hybride.



Figure 3.6
Le jeu DollarStash en cours d'exécution.

Comme d'autres applications, `UIWebView` utilise le moteur WebKit de Safari pour afficher le contenu à l'écran. Ce moteur, et d'autres comme lui, a énormément évolué ces dernières années. Toutefois, lorsque le processeur est fortement sollicité, quelques événements de l'interface utilisateur, comme les clics et les touchers, sont ignorés. Si vous jouez au jeu DollarStash sur votre appareil, non le simulateur, vous le constaterez rapidement.

Plus le nombre de billets augmente, plus le nombre de touchers ignorés par le moteur augmente. En soumettant l'utilisateur à un tel désagrément, vous ne respectez pas une règle fondamentale de la conception de l'interface utilisateur mentionnée dans la première section de ce chapitre : la réponse aux actions de l'utilisateur doit être rapide.

Pour les portions fortement interactives des applications de ce type, il est préférable d'utiliser Objective-C. Cela ne signifie pas que les applications ne puissent pas utiliser `UIWebView` pour la mise en page de textes et un affichage simple d'images. Cela signifie que, jusqu'à ce que la puissance des processeurs des iPhone et des iPod Touch augmente significativement, il ne faut pas utiliser les transformations et les animations CSS (*Cascading Style Sheet*) natives pour la création de jeux complexes.

En connaissant les limites de l'appareil, il est possible d'opter pour une meilleure conception. Si les jeux gourmands en ressources processeur ne sont pas vraiment viables dans des applications hybrides, l'utilisation des transformations et des animations CSS pour réaliser le glisser-déposer, le redimensionnement et la rotation d'un seul élément de l'interface à la fois est tout à fait envisageable.

Openmirrors.com

Section 5 : créer et utiliser des transformations CSS personnalisées

Cette section explique comment créer des possibilités de glisser-déposer, de redimensionnement et de rotation en se fondant sur les nouvelles transformations CSS intégrées au moteur WebKit. Pour de plus amples informations concernant les transitions, les transformations et les animations CSS, consultez le guide des effets visuels avec Safari à l'adresse **http://developer.apple.com/safari/library/documentation/InternetWeb/Conceptual/SafariVisualEffectsProgGuide/Introduction/Introduction.html**.

Plusieurs mises en œuvre JavaScript du glisser-déposer sont disponibles en téléchargement. Si elles sont parfaitement adaptées à une utilisation multinavigateur sur les machines de bureau, elles sont inefficaces sur les iPhone et les iPod Touch, car elles demandent des ressources processeur trop importantes. Une bonne alternative consiste à employer les transformations CSS.

WebKit, le moteur utilisé par Safari et la classe `UIWebView` dans les applications hybrides, est capable de prendre en charge les transitions définies en CSS. Elles bénéficient en effet d'une accélération matérielle, ce qui les rend plus efficaces que des modifications programmées en JavaScript comme dans les autres bibliothèques.

Prenons comme exemple simple le décalage vers le bas de la position d'une balise HTML `<div>`. Dans une version JavaScript classique, il faut modifier l'attribut `top` du style de la balise. Supposons qu'une classe CSS soit affectée au `<div>` et que la valeur de l'attribut `top` soit égale à 50 pixels. Un décalage vers le bas de 50 pixels supplémentaires s'obtient en fixant ce même attribut à 100 pixels :

```
unDiv.style.top = '100px';
```

Cette déclaration est interprétée comme une commande JavaScript et elle est exécutée par le moteur à la même vitesse et en utilisant les mêmes ressources processeur que n'importe quelle autre commande JavaScript. La transformation CSS équivalente fonctionne différemment.

En utilisant les transformations CSS pour obtenir ce décalage de 50 pixels par rapport à la position d'origine déclarée, il faut également employer une déclaration du style pour le `<div>`. Toutefois, dans ce cas, nous utilisons un attribut totalement différent.

`webkitTransform` fait partie des nouveaux attributs ajoutés aux classes CSS et, par conséquent, à l'attribut `style` de l'objet JavaScript `Element`. En lui donnant la bonne valeur, des fonctions natives bénéficiant d'une accélération matérielle sont invoquées. Puisqu'il n'est

pas interprété comme du JavaScript, toute modification de l'attribut CSS défini est exécutée beaucoup plus rapidement que le code JavaScript équivalent de l'exemple précédent.

La transformation requise ici ne demande qu'une seule ligne de code :

```
unDiv.style.webkitTransform = 'translateY(50px)';
```

À première vue, l'attribut de transformation ressemble à un pointeur de fonction, comme `onclick`, `ontouch` et d'autres gestionnaires d'événements, mais ce n'est pas le cas.

La principale différence entre `webkitTransform` et les gestionnaires réside dans le fait qu'aucune fonction JavaScript déclarée en ligne ou comme une fonction de `Window` n'est affectée. À la place, une chaîne décrivant la fonction standard à invoquer et ses paramètres est utilisée. Cette chaîne est ensuite analysée par le moteur WebKit dans une partie de son code distincte de celle qui interprète les instructions JavaScript.

Notez également que le décalage indiqué est relatif à l'emplacement d'origine du `<div>`. Pour décaler un `Element` de 50 pixels supplémentaires vers le bas, nous passons à `translateY` le paramètre `50px`. Il est également important de comprendre que la définition de l'emplacement initial n'a pas changé. La valeur d'origine de l'attribut `top` du `<div>` est toujours égale à `50px`. Seul l'emplacement auquel il est affiché a changé. Si, après le déplacement, vous consultez l'attribut `top` de l'objet, vous obtenez toujours la valeur `50px`, non `100px`.

L'application Drag montre comment mouvoir un `<div>` sur l'écran en utilisant une fonction de déplacement. Pour cela, des gestionnaires `ontouchstart`, `ontouchchange` et `ontouchend` sont associés au `<div>` déplacé.

Les événements de toucher diffèrent des événements `onclick`, `onmousedown`, `onmousemove` et `onmouseup` standard utilisés dans les implémentations JavaScript classiques du glisser-déposer. Puisqu'un toucher peut être constitué de deux touchers individuels, ou plus, par exemple lorsque l'utilisateur pose deux ou trois doigts sur l'écran, un événement de toucher doit contenir des informations concernant chacun d'eux.

Chaque toucher individuel et ses informations sont enregistrés dans le tableau `targetTouches`, qui est un attribut de l'objet `event`. Ce tableau est dimensionné d'après le nombre de doigts posé sur l'élément par l'utilisateur. Sa taille est donc égale à un lorsqu'un doigt est posé, à deux, lorsque deux doigts sont utilisés.

Chaque objet enregistré dans le tableau est de type `Touch` et comprend de nombreux attributs généralement associés à un événement de la souris en JavaScript. Le Tableau 3.3 recense ces attributs.

Openmirrors.com

Tableau 3.3 : Attributs de la classe `Touch`

Attribut	*Description*
`pageX`	Le décalage horizontal à partir du bord gauche du document, y compris les informations de défilement horizontal
`pageY`	Le décalage vertical à partir du bord supérieur du document, y compris les informations de défilement vertical
`screenX`	Le décalage horizontal à partir du bord gauche de l'écran de l'appareil
`screenY`	Le décalage vertical à partir du bord supérieur de l'écran de l'appareil
`clientX`	Le décalage horizontal à partir du bord gauche de la fenêtre de l'application
`clientY`	Le décalage vertical à partir du bord supérieur de la fenêtre de l'application
`target`	L'objet du DOM qui représente l'élément HTML qui a été touché

La mise en œuvre du glisser-déposer proposée se fonde sur les attributs `clientX` et `clientY` car le défilement n'est pas autorisé dans l'application d'exemple. Dans le cas contraire, il faudrait utiliser les attributs `pageX` et `pageY`.

Les éléments "sautillants" qui apparaissent lorsque le glisser-déposer débute posent un problème. Ce sautillement se produit car l'utilisateur a sélectionné l'objet en le touchant quelque part à l'intérieur de son contour alors que le déplacement est appliqué au coin supérieur gauche. Si cette discordance n'est pas gérée, le coin supérieur gauche de l'objet que l'utilisateur fait glisser "saute" à l'emplacement de son doigt au début de l'opération.

Évidemment, l'utilisateur considère ce comportement comme anormal. Par exemple, s'il sélectionne le centre de l'objet pour débuter l'opération, il peut raisonnablement s'attendre à ce que son doigt reste au centre de l'objet pendant l'opération. La Figure 3.7 illustre le fonctionnement de l'application Drag.

Pour remédier à ce problème, l'application affecte la fonction `setStartLocation` au gestionnaire JavaScript `ontouchstart`. Cette fonction, extraite du fichier *main.js* et reproduite ci-après, obtient et enregistre la position du toucher d'origine, en pixels, par rapport au coin supérieur gauche de la fenêtre de l'application.

```
1 function setStartLocation(event)
2 {
3     var element = event.target;
4     element.offsetX = event.targetTouches[0].clientX;
5     element.offsetY = event.targetTouches[0].clientY;
6 }
```

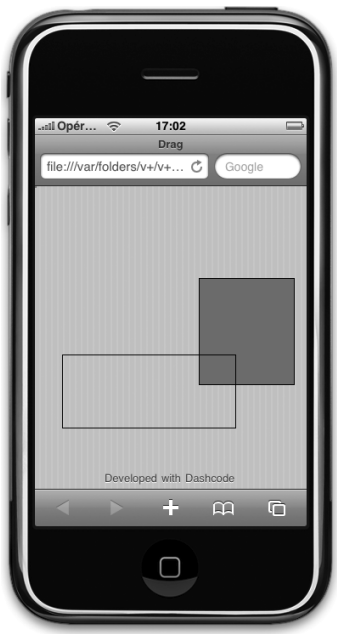


Figure 3.7
L'application Drag après le déplacement du `<div>` *vert.*

En enregistrant cette distance dans les attributs `offsetX` et `offsetY` de l'élément touché, nous pouvons ensuite l'utiliser pendant le déplacement de l'élément de manière à empêcher le sautillement. Le déplacement de l'élément se produit non pas dans la fonction `setStartLocation`, mais dans la fonction `drag` définie comme gestionnaire `ontouchchange`. Cette fonction se trouve également dans le fichier *main.js*.

La ligne 3 de la fonction `drag` donnée ci-après est indispensable à toute mise en œuvre du glisser-déposer pour l'iPhone et l'iPod Touch. Normalement, lorsqu'un événement de type modification du toucher est déclenché, le navigateur Safari ou `UIWebView` procède à un défilement. Pour désactiver ce comportement standard, nous invoquons la méthode `preventDefault` de `event`. Lorsque cette méthode est invoquée dans le gestionnaire `ontouchchange`, la vue ne défile pas si le doigt est déplacé dans l'élément auquel le gestionnaire est affecté.

Étant débarrassé du défilement par défaut, vous pouvez modifier l'emplacement auquel l'élément est affiché en utilisant `webkitTransform`. Pour cela, l'emplacement du toucher actuel doit être obtenu et comparé à l'emplacement du toucher initial enregistré par la fonction `setStartLocation`.

```
1  function drag(event)
2  {
3      event.preventDefault();
```

Openmirrors.com

```
 4     var element = event.target;
 5     element.x = event.targetTouches[0].clientX
 6                     - event.target.offsetX;
 7     element.y = event.targetTouches[0].clientY
 8                     - event.target.offsetY;
 9     if(element.lastX || element.lastY){
10         element.x += element.lastX;
11         element.y += element.lastY;
12     }
13     element.style.webkitTransform = 'translate('
14                 + element.x + 'px, '
15                 + element.y + 'px)';
16 }
```

Le code des lignes 5 à 8 calcule les décalages de la position de l'élément selon les axes X et Y. Ces résultats, en pixels, sont enregistrés dans les attributs `x` et `y` de l'élément courant en vue de leur utilisation ultérieure, c'est-à-dire aux lignes 13 à 15 pour modifier l'affichage à l'aide de la fonction `translate` décrite précédemment.

Puisqu'il est possible que ce glisser ne soit pas le premier effectué par l'utilisateur sur un élément, il est nécessaire de conserver les déplacements réalisés précédemment. Le déplacement se fait aux lignes 9 à 11 du code précédent et les décalages sont enregistrés dans la méthode `done` ci-après, qui est définie comme gestionnaire `ontouchend`.

```
function done(event)
{
    var element = event.target;
    element.lastX = element.x;
    element.lastY = element.y;
}
```

La méthode `done` existe pour une seule raison : enregistrer le décalage actuel pour le retrouver si l'utilisateur venait à déplacer à nouveau l'élément. Pour cela, elle enregistre les attributs `x` et `y` actuels de l'élément dans les attributs `lastX` et `lastY`. Ainsi, nous sommes certains de pouvoir en disposer à chaque déclenchement des événements `ontouchchange` lorsque l'utilisateur déplace son doigt sur l'écran.

En affectant ces trois méthodes en tant que gestionnaires aux éléments de l'interface, ils peuvent être déplacés par l'utilisateur de manière simple. À la section suivante, vous verrez comment créer et utiliser un module de glisser-déposer moins naïf et plus simple d'emploi.

Outre le glisser-déposer, les applications pour l'iPhone ont souvent besoin d'une possibilité de redimensionnement et de rotation des éléments de l'interface. Ces éléments peuvent être des `<div>`, des boutons, des images ou tout autre élément d'organisation ou graphique.

Le code nécessaire à la mise en œuvre de ces fonctionnalités est plus court que celui du glisser-déposer. Il semble évident qu'Apple a voulu que les développeurs incluent ces comportements dans leurs applications.

Les applications d'illustration du traitement des gestes montrent que le redimensionnement et la rotation sont faciles à réaliser. À la place des touchers, elles utilisent des gestes (*gesture*). Les gestes diffèrent des touchers en cela qu'ils supposent toujours que plusieurs doigts sont employés, par exemple lors d'un pincement.

Pour représenter ces gestes, un `GestureEvent` est passé à tout gestionnaire de gestes. Puisque cette classe représente un geste, elle ne comprend aucune information de position comme la classe `TouchEvent`. En revanche, elle transmet trois éléments d'information importants recensés au Tableau 3.4.

Tableau 3.4 : Les attributs importants de `GestureEvent`

Attribut	*Description*
`scale`	Une valeur réelle positive ou négative qui représente le changement de la distance séparant les deux doigts utilisés dans un geste. Les valeurs négatives indiquent que les doigts se sont croisés. Cet attribut est utilisé dans la gestion du pincement.
`rotation`	Une valeur réelle positive ou négative qui indique, en degrés, la différence angulaire entre les positions des deux doigts utilisés dans un geste et une ligne verticale. Cet attribut est utilisé dans la gestion des rotations.
`target`	L'objet du DOM qui représente l'élément HTML concerné par le geste.

L'application d'illustration du traitement des gestes utilise trois événements pour redimensionner et faire pivoter un `<div>`. Pour cela, nous ajoutons au `<div>` un gestionnaire `ongesturechange` nommé `changeIt`. Pour ajouter des gestionnaires d'événements, utilisez l'onglet `Comportements` de l'inspecteur. Voici le code de la méthode `changeIt` :

```
function changeIt(event)
{
    event.preventDefault();
    var element = event.target;

    element.style.webkitTransform=
            'rotateZ('+event.rotation
            +'deg) scale('+event.scale+')';
}
```

Openmirrors.com

À l'instar de l'application Drag, le comportement par défaut associé à l'événement doit être désactivé de manière à empêcher le défilement. Mais, contrairement à Drag, les informations de rotation et de redimensionnement ne sont pas enregistrées. Il est inutile d'enregistrer les informations de rotation car elles sont relatives à une ligne de base, non à l'objet transformé.

Les informations de redimensionnement doivent être enregistrées pour être utilisées lors du geste suivant, car elles sont relatives à l'élément transformé et se cumulent. Cette sauvegarde n'est pas effectuée dans notre exemple car nous voulons illustrer le dysfonctionnement correspondant. La section suivante montre comment les enregistrer et les réutiliser.

Notez que deux fonctions sont employées dans la chaîne qui définit la commande de `webkitTransform`. Cela permet d'effectuer la rotation et le redimensionnement du `<div>` en un seul appel. Rien ne vous empêche de les séparer et de les invoquer de manière conditionnelle.

Vous disposez de trois fonctions de rotation pour votre application. Chacune fait pivoter l'élément autour de l'un des axes du téléphone. L'axe X est horizontal, l'axe Y est vertical et l'axe Z sort de l'écran. Dans le code précédent, nous choisissons `rotateZ` pour que le `<div>` pivote sur le plan X-Y de l'appareil (voir Figure 3.8).



Figure 3.8
Application de la fonction `rotateZ` de webkitTransform.

Il est très facile de modifier la façon dont le `<div>` pivote. En utilisant `rotateY`, il pivote selon l'axe Y et semble devenir plus étroit avant d'afficher son verso. Si vous utilisez `rotateX`, il pivote le long de l'axe X et semble rapetisser avant d'afficher son verso.

Vous pourriez penser à une mise en œuvre de type Cover Flow à l'aide de la rotation. Au moment de l'écriture de ces lignes, elle est déconseillée. En effet, le nombre de transformations nécessaires à un tel comportement va solliciter énormément le processeur de l'iPhone ou de l'iPod Touch et l'utilisateur risque d'être déçu.

Puisque vous connaissez à présent les implémentations naïves du glisser-déposer, du redimensionnement et de la rotation, vous êtes en mesure de comprendre un module sophistiqué qui propose ces fonctionnalités.

Section 6 : créer et utiliser un module de glisser-déposer, de redimensionnement et de rotation

Tel qu'il est expliqué au Chapitre 2, les modules sont indépendants et mettent en œuvre l'intégralité d'une fonctionnalité. Autrement dit, ils sont faiblement couplés au reste du code d'une application et affichent une cohésion forte. Les modules bien conçus fournissent toujours une API. Celle que nous allons implémenter dans cette section est décrite au Tableau 3.5.

Tableau 3.5 : API de glisser-déposer, redimensionnement et rotation

Fonction	*Paramètres*	*Description*
`makeDraggable`	`element` (obligatoire) – l'élément du DOM cible du glissement. `startDragCmd` (facultatif) – une commande associée aux fonctions de contrôle invoquées à la fin de l'événement `ontouchstart`. `dragCmd` (facultatif) – une commande associée aux fonctions de contrôle invoquées à la fin de tous les événements `ontouchmove`. `dropCmd` (facultatif) – une commande associée aux fonctions de contrôle invoquées à la fin de l'événement `ontouchend`.	Cette fonction configure les gestionnaires d'événements sur l'élément indiqué afin que l'utilisateur puisse le faire glisser.

Openmirrors.com

Tableau 3.5 : API de glisser-déposer, redimensionnement et rotation *(suite)*

Fonction	*Paramètres*	*Description*
`makeChangeable`	`element` (obligatoire) – l'élément du DOM cible du redimensionnement et de la rotation. `startChangeCmd` (facultatif) – une commande associée aux fonctions de contrôle invoquées à la fin de l'événement `ongesturestart`. `dragCmd` (facultatif) – une commande associée aux fonctions de contrôle invoquées à la fin de tous les événements `ongesturechange`. `doneChangeCmd` (facultatif) – une commande associée aux fonctions de contrôle invoquées à la fin de l'événement `ongestureend`.	Cette fonction configure les gestionnaires d'événements sur l'élément indiqué afin que l'utilisateur puisse le redimensionner et le faire pivoter autour de l'axe Z.

Dans votre code, vous devez simplement invoquer ces deux fonctions pour que l'utilisateur dispose des fonctionnalités de glisser-déposer, de redimensionnement et de rotation (voir Figure 3.9). L'application dragAndGesture se trouve dans le répertoire *Examples* du paquetage QuickConnectiPhone téléchargé. Les fonctions sont définies dans le fichier *QCUtilities.js* fourni par le framework.

Voici le code de la fonction `load` définie dans le fichier *main.js*. Il montre comment utiliser ces fonctions avec des éléments de l'interface utilisateur.

```
function load()
{
    ...
    var anElement = document.getElementById('button');
    makeDraggable(anElement);
    makeChangeable(anElement);
    anElement = document.getElementById('imageBox');
    makeDraggable(anElement);
    makeChangeable(anElement);
    anElement = document.getElementById('box');
    makeDraggable(anElement);
    makeChangeable(anElement);
    anElement = document.getElementById('stuff');
    makeDraggable(anElement);
}
```



Figure 3.9
L'application dragAndGesture en cours d'exécution, avec un élément déplacé et pivoté.

Dans cet exemple, trois éléments de l'interface sont modifiés afin que l'utilisateur puisse les faire glisser, les redimensionner et les faire pivoter ; le quatrième pourra seulement être déplacé. Après avoir obtenu une référence à l'élément de l'interface, elle est passée aux fonctions de l'API. Cela suffit pour que les éléments de l'interface utilisateur deviennent actifs.

Les mises en œuvre naïves du glisser-déposer, du redimensionnement et de la rotation décrites précédemment dans cette section fonctionnent indépendamment l'une de l'autre. L'exemple ci-dessus montre qu'elles doivent pouvoir opérer de concert. Pour cela, elles doivent connaître l'effet, s'il existe, que les autres fonctions ont appliqué à l'élément. La modification requise commence dans les fonctions `makeDraggable` et `makeChangeable`.

La fonction `makeDraggable` de l'API prend en charge la configuration et la gestion du gestionnaire `ontouchstart`, ainsi que les commandes passées à exécuter après l'événement.

```
function makeDraggable(anElement, startDragCmd, dragCmd, dropCmd){
    anElement.ontouchstart = prepareDrag;
    anElement.isDraggable = true;
    if(startDragCmd){
        anElement.startDragCmd = startDragCmd;
    }
```

Openmirrors.com

```
    if(dragCmd){
        anElement.dragCmd = dragCmd;
    }
    if(dropCmd){
        anElement.dropCmd = dropCmd;
    }
}
```

Vous remarquerez que l'attribut `isDraggable` de l'élément est fixé à `true`. Il s'agit de la première information que nous devons enregistrer pour que les fonctionnalités puissent cohabiter. Notez également qu'un seul gestionnaire de toucher, `ontouchstart`, est affecté dans cette fonction. En effet, nous voulons ignorer les touchers lorsqu'un élément est redimensionné ou pivoté ; nous y reviendrons plus loin dans cette section.

La fonction `makeChangeable` est comparable. Elle affecte les gestionnaires de gestes, fixe l'attribut `isChangeable` à `true` et enregistre les commandes qui devront être exécutées après l'événement. Contrairement à la fonction `makeDraggable`, elle fixe les gestionnaires de gestes. En effet, lorsque des événements de toucher sont déclenchés suite au toucher simple d'un glissement, aucun événement de geste n'est déclenché. Si un élément est déplaçable et que le glissement se produise, les événements de toucher sont déclenchés et sont traités. Si un élément est modifiable et qu'un geste se produise, les événements de toucher et de geste sont déclenchés, les événements de toucher sont ignorés et les événements de geste sont traités.

```
function makeChangeable(anElement, startChangeCmd, changeCmd, doneChangeCmd){
    anElement.ongesturestart = prepareGesture;
    anElement.ongesturechange = changeIt;
    anElement.ongestureend = gestureDone;
    anElement.isChangeable = true;
    anElement.oldRotation = 0;
    anElement.oldScale = 1;
    anElement.startChangeCmd = startChangeCmd;
    anElement.changeCmd = changeCmd;
    anElement.doneChangeCmd = doneChangeCmd;
}
```

Deux autres paramètres sont initialisés dans la méthode `makeChangeable`. Il s'agit des attributs de mémorisation du redimensionnement et de la rotation nommés `oldScale` et `oldRotation`.

Dans l'application gestures, chaque fois que l'élément est pivoté ou redimensionné, il commence par reprendre sa taille d'origine, car le redimensionnement effectué précédemment n'est pas mémorisé automatiquement. L'attribut `oldScale` de l'élément permet de résoudre ce problème de taille.

L'attribut `oldScale` est initialisé à `1`, car le redimensionnement correspond à un multiplicateur appliqué à la largeur et à la hauteur de l'élément ; nous y reviendrons plus loin dans cette section. Si le multiplicateur est supérieur ou égal à `0` et inférieur à `1`, l'élément est réduit. Si le multiplicateur est égal à `1`, l'élément reste inchangé. Si le multiplicateur est supérieur à `1`, l'élément est agrandi.

Lorsqu'un élément a déjà été redimensionné, le facteur utilisé doit être combiné au nouveau multiplicateur pour obtenir la taille correcte de l'élément. Par exemple, si le premier redimensionnement de l'élément avait doublé sa taille, la valeur de `oldScale` doit être égale à `2`. Si l'utilisateur pince l'élément de manière à le réduire de 10 %, le nouveau facteur de redimensionnement doit être égal à `2 × 0,9`, c'est-à-dire `1,8`. Si la valeur de `oldScale` n'était pas conservée, la taille de l'élément serait fixée à `0,9` et ne correspondrait pas aux intentions de l'utilisateur.

Précédemment, la fonction `prepareDrag` était affectée au gestionnaire d'événements `ontouchstart`. Cette fonction, dont le code est donné ci-après, contient plusieurs points intéressants. Tout d'abord, un tableau d'objets `Touch` est enregistré. Ainsi, les fonctions de traitement des événements de geste peuvent accéder aux informations de toucher pertinentes. Par exemple, il peut être nécessaire de connaître le nombre de touchers qui ont provoqué l'événement de geste. Cette information n'est pas disponible dans les événements passés aux gestionnaires de gestes.

```
function prepareDrag(event){
    stopDefault(event);
    this.touches = event.targetTouches;
    var self = this;
    this.timeOut = setTimeout(function(){
        if(self.changing){
            return;
        }
        self.dragging = true;
        self.ontouchmove = dragIt;
        self.ontouchend = dragDone;
        self.offsetX = event.targetTouches[0].clientX;
        self.offsetY = event.targetTouches[0].clientY;
        self.oldZIndex = self.style.zIndex;
        self.style.zIndex = 50;
        if(self.startDragCmd){
            var params = new Array();
            params.push(event);
            params.push(self);
            handleRequest(self.startDragCmd, params);
        }
    }, 75);
}
```

Openmirrors.com

Un autre point intéressant du code précédent se trouve à la ligne 5. Au lieu d'enregistrer immédiatement la position initiale du toucher dans les attributs `offsetX` et `offsetY` de l'élément cible du glissement, une minuterie retarde l'affectation de ces valeurs et d'autres. Nous procédons ainsi, car la fonction `prepareDrag` peut avoir été appelée suite à un geste effectué par l'utilisateur.

Les événements de toucher déclenchés par des gestes surviennent toujours avant les événements de geste déclenchés. Si l'événement passé à `prepareDrag` vient effectivement d'un geste, les gestionnaires `ontouchmove` et `ontouchend` ne doivent pas être fixés pour éviter qu'ils ne soient appelés lorsque le geste change et lorsqu'il se termine. Si ces deux gestionnaires étaient invoqués, ils produiraient un comportement de glissement et provoqueraient un dysfonctionnement du comportement associé au geste.

La minuterie définie doit être suffisamment longue pour que la fonction `prepareGesture` gestionnaire du geste soit appelée. En effet, elle met à jour l'attribut `changing` de l'élément en cours de modification.

La ligne 22 montre que le retard est de 75 millisecondes. Cela suffit pour que le gestionnaire de geste soit invoqué et exécuté si un geste se produit, tout en restant suffisamment court pour que l'utilisateur ne soit pas gêné dans le cas d'un glissement. Si la minuterie est trop longue, l'utilisateur risque d'avoir fait glisser son doigt en dehors de l'élément avant que celui-ci ne se déplace.

Le dernier point intéressant concerne le remplissage du tableau `params` et l'appel à `handleRequest`. L'invocation de la fonction `handleRequest` à la ligne 20 permet de créer des fonctions différées dont l'exécution a lieu chaque fois qu'un événement de toucher correspond à un glissement. Ces fonctions différées sont définies à l'aide des fonctions `mapCommandTo*` fournies dans le fichier *mappings.js* (voir Chapitre 2). Vous pouvez appeler autant de fonctions de contrôle métier (BCF, *Business Control Function*) et de fonctions de contrôle de l'affichage (VCF, *View Control Function*) que vous le souhaitez lorsque le glissement débute. Par exemple, vous pouvez les utiliser pour retirer l'élément de son parent, modifier sa couleur d'arrière-plan ou ses bordures, etc.

Puisque rien n'indique comment et si vous utiliserez ces fonctions différées, certaines informations sont placées dans le tableau `param`. Comme le montre le code précédent, ce tableau comprend l'élément déplacé et l'événement correspondant. Ces informations ne seront peut-être pas utiles dans les fonctions différées que vous allez créer, mais elles sont à votre disposition en cas de besoin.

Lorsque l'utilisateur déplace son doigt sur l'écran, le gestionnaire `dragIt` de l'événement `ontouchmove` est invoqué de manière répétée. À l'instar de la fonction `drag` de l'application Drag, cette fonction prend en charge le déplacement de l'élément conformément au

doigt de l'utilisateur. Elle montre l'intérêt des informations enregistrées précédemment dans les fonctions gestionnaires de gestes et dans `prepareDrag`.

Puisque le glisser-déposer se fond sur une transformation, lors d'un glissement, toute rotation et tout redimensionnement déjà effectués doivent être appliqués en plus du déplacement. En effet, le `webkitTransform` du style est réinitialisé à chacune de ses utilisations. Si les informations de rotation et de redimensionnement n'étaient pas incluses dans la chaîne de transformation, l'élément serait considéré dans sa taille et son orientation d'origine lors du glissement.

Une chaîne de transformation qui provoque un déplacement, une rotation et un redimensionnement contient plusieurs fonctions et prend l'aspect suivant :

```
"translate(-1px, 5px) rotateZ(21deg) scale(0.9)"
```

Cette instruction déplace l'élément de un pixel vers la gauche et de cinq pixels vers le bas. Puis elle pivote l'élément autour de son axe Z. Enfin, la taille de l'élément est fixée à 90 % de sa taille d'origine.

L'ordre des fonctions est important. Si la rotation se trouve à gauche du déplacement dans la chaîne, elle se produit avant celui-ci. Par conséquent, le déplacement se fait selon un angle avec les axes X et Y au lieu de se faire le long de ces axes. Le glisser-déposer aurait un comportement étrange, car l'élément se déplacerait en décalage du mouvement du doigt de l'utilisateur au lieu de le suivre.

Le code de création de la chaîne se trouve aux lignes 13 à 25. Elles concatènent une sous-chaîne à une chaîne qui contient la déclaration de la fonction `translate`.

```
 1 function dragIt(event){
 2     stopDefault(event);
 3
 4     this.x = event.targetTouches[0].clientX - this.offsetX;
 5     this.y = event.targetTouches[0].clientY - this.offsetY;
 6
 7     if(this.lastX || this.lastY){
 8         this.x += this.lastX;
 9         this.y += this.lastY;
10     }
11     this.style.webkitTransformOriginX = '50%';
12     this.style.webkitTransformOriginY = '50%';
13     var modStringFragment = '';
14     if(this.isChangeable){
15         if(this.rotation){
16             modStringFragment +=
17                     ' rotateZ('+this.oldRotation+'deg)';
18         }
```

Openmirrors.com

```
19          if(this.oldScale){
20              modStringFragment +=
21                      ' scale('+this.oldScale+')';
22          }
23      }
24      var modString = 'translate(' + this.x + 'px, '
25              + this.y + 'px)'+modStringFragment;
26
27      this.style.webkitTransform = modString;
28      if(this.dragCmd){
29          var params = new Array();
30          params.push(event);
31          params.push(this);
32          handleRequest(this.dragCmd, params);
33      }
34  }
```

Les lignes 11 et 12 présentent un certain intérêt. Les attributs `webkitTransformOriginX` et `webkitTransformOriginY` sont fixés à leur valeur par défaut. Comme vous le verrez plus loin dans cette section, cette opération doit être effectuée dans le cas de tout redimensionnement ou rotation. Sinon, si l'utilisateur fait glisser un élément, celui-ci saute et le doigt ne se trouve pas sur le point de l'élément touché initialement par l'utilisateur.

Lorsque l'utilisateur retire son doigt de l'écran de l'appareil, un événement `ontouchend` est déclenché et la fonction gestionnaire `dragDone` est invoquée. Le code suivant, extrait du fichier *QCUtilities.js*, montre que son rôle est de réinitialiser certains attributs de l'élément à leur valeur d'origine et d'enregistrer des informations en vue de leur utilisation ultérieure.

```
function dragDone(event){
    this.dragging = false;
    this.ontouchmove = null;
    this.ontouchend = null;
    this.lastX = this.x;
    this.lastY = this.y;
    this.style.zIndex = this.oldZIndex;

    if(this.dropCmd){
        var params = new Array();
        params.push(event);
        params.push(this);
        handleRequest(this.dropCmd, params);
    }
}
```

Dans cette fonction, les gestionnaires `ontouchmove` et `ontouchend` sont retirés afin d'éviter les interférences avec le traitement du geste. Les coordonnées `x` et `y` actuelles de l'événement sont enregistrées et une commande est exécutée si l'une d'elles est positionnée.

Après avoir vu l'ensemble des méthodes sophistiquées du glisser-déposer, le traitement des gestes est plus facile à comprendre. Tout comme la prise en charge des gestes peut affecter le code du glisser-déposer, la gestion du glisser-déposer peut affecter le code de redimensionnement et de rotation.

La fonction `prepareGesture`, dont le code donné ci-après se trouve dans le fichier *QCUtilities.js*, est plus simple que la fonction `prepareDrag`. Elle fixe quelques attributs, mais elle n'a pas besoin de retarder certaines opérations, comme nous l'avons expliqué précédemment dans cette section.

```
function prepareGesture(event){
    stopDefault(event);
    this.changing = true;
    this.oldZIndex = this.style.zIndex;
    this.style.zIndex = 50;

    if(this.startChangeCmd){
        var params = new Array();
        params.push(event);
        params.push(this);
        handleRequest(this.startChangeCmd, params);
    }
}
```

À l'instar des autres fonctions de prise en charge des événements de geste, celle-ci demande au framework d'exécuter une commande en invoquant la fonction `handleRequest`. Vous pouvez exécuter autant de fonctions différées que vous souhaitez après la terminaison du gestionnaire d'événements de geste. Pour de plus amples informations concernant l'association de commandes à des fonctions, consultez le Chapitre 2.

Lorsque l'utilisateur déplace son doigt sur l'écran, un événement `ongesturechange` est déclenché de manière répétée. Cela conduit à l'invocation de la fonction `changeIt`, dont le code extrait du fichier *QCUtilities.js* est donné ci-après. Cette fonction est responsable du redimensionnement et de la rotation de l'élément en fonction de l'interaction de l'utilisateur avec l'appareil.

Notez qu'il est possible d'effectuer un geste du doigt sur deux éléments différents ou plus. En général, cela vient d'une erreur de l'utilisateur et, par conséquent, le code doit éviter que les éléments ne réagissent lorsqu'ils sont touchés (lignes 5 à 8).

Openmirrors.com

Il est également possible que l'utilisateur effectue un pincement qui produit un élément trop petit pour accepter deux doigts. Si ce comportement était autorisé, l'utilisateur ne pourrait plus redimensionner l'élément, qui ne serait donc pas modifiable. Les lignes 14 à 19 évitent que l'utilisateur ne réduise trop un élément.

Nous l'avons expliqué précédemment, l'ordre des déclarations des fonctions dans la chaîne de transformation est important. Pour que la rotation et le redimensionnement d'un élément réussissent, le déplacement doit se produire en dernier, en le déclarant à la fin de la chaîne.

```
"rotateZ(21deg) scale(0.9) translate(-1px, 5px)"
```

Si le déplacement avait lieu en premier alors que l'utilisateur tente de faire pivoter un élément, celui-ci se déplacerait progressivement au cours de la rotation. Si le déplacement n'était pas inclus, l'élément pivoterait autour de son coin supérieur gauche d'origine, comme défini dans la classe CSS associée. Ces deux comportements ne sont pas acceptables.

```
 1 function changeIt(event){
 2     stopDefault(event);
 3     // L'utilisateur peut avoir posé un seul doigt
 4     // à l'intérieur de la cible.
 5     if(this.dragging
 6             || (this.touches && this.touches.length < 2)){
 7         return;
 8     }
 9
10     this.rotation = event.rotation;
11     var rotationValue = this.rotation + this.oldRotation;
12     var scaleValue = event.scale * this.oldScale;
13     // Faire en sorte que l'élément ne soit pas trop petit pour accepter deux
       ➥touchers.
14     if(this.offsetWidth * scaleValue < 150){
15         scaleValue = 150/this.offsetWidth;
16     }
17     else if(this.offsetHeight * scaleValue < 150){
18         scaleValue = 150/this.offsetHeight;
19     }
20     this.scale = scaleValue;
21
22     var modString = 'rotateZ('+rotationValue+
23             'deg) scale('+scaleValue+')';
24     if(this.lastX || this.lastY){
25         modString += ' translate(' + this.lastX + 'px, '
26                 + this.lastY + 'px)';
```

```
27          // Actualiser le centre de rotation.
28          this.xCenterOffset = 50
29                  + (this.lastX/this.offsetWidth)
30                  * 100;
31          this.yCenterOffset = 50
32                  + (this.lastY/this.offsetHeight)
33                  * 100;
34
35          this.style.webkitTransformOriginX =
36                  (this.xCenterOffset)+'%';
37          this.style.webkitTransformOriginY =
38                  this.yCenterOffset)+'%';
39      }
40      this.style.webkitTransform = modString;
41
42      if(this.changeCmd){
43          var params = new Array();
44          params.push(event);
45          params.push(this);
46          handleRequest(this.changeCmd, params);
47      }
48  }
```

Les lignes 22 à 39 créent la chaîne de transformation et fixent les valeurs de `webkitTransformOrigin` pour que le redimensionnement et la rotation puissent se produire par rapport au centre visuel de l'élément modifié. Les changements d'origine effectués dans le code précédent expliquent les réinitialisations réalisées dans la méthode `dragIt` décrite précédemment. L'origine de la transformation est fixée au milieu de l'élément, conformément à son décalage courant par rapport à sa position initiale.

Lorsque l'utilisateur retire ses doigts de l'écran, un événement `ongestureend` est déclenché et le gestionnaire `gestureDone` est invoqué. Il est comparable à la méthode `dragDone`, à une exception près : comme la fonction `prepareDrag`, il utilise une minuterie.

```
1 function gestureDone(event){
2      this.style.zIndex = this.oldZIndex;
3      // L'utilisateur n'a peut-être pas effectué une rotation.
4      // Dans ce cas, rotation n'est pas défini.
5      if(this.rotation){
6          this.oldRotation += this.rotation;
7      }
8      // L'utilisateur n'a peut-être pas effectué un pincement.
9      // Dans ce cas, scale n'est pas défini.
10     if(this.scale){
11         this.oldScale = this.scale;
12     }
```

Openmirrors.com

```
13
14      if(this.doneChangeCmd){
15          var params = new Array();
16          params.push(event);
17          params.push(this);
18          handleRequest(this.doneChangeCmd, params);
19      }
20      var self = this;
21      this.timeOut = setTimeout(function(){
22          self.changing = false;
23      },75);
24  }
```

Cette minuterie (lignes 21 à 23) existe pour des raisons semblables à celles données pour `prepareDrag`. Lorsque l'événement `ongestureend` est déclenché, certains événements de toucher ne doivent pas être traités. Si l'attribut `changing` de l'élément est immédiatement fixé à `false` (ligne 22), alors, d'après le code de `prepareDrag` vu précédemment, les gestionnaires du glissement sont activés.

Si ces gestionnaires du glissement sont activés pour un geste de redimensionnement ou de rotation, l'élément saute de manière erratique sur l'écran et l'expérience de l'utilisateur en est affectée négativement.

Bien que les comportements de glisser-déposer, de redimensionnement et de rotation doivent connaître l'existence des uns et des autres, ils doivent être strictement séparés. Dans le cas contraire, un comportement confus et aléatoire de l'application est possible. Avec le code présenté dans cette section, extrait du fichier *QCUtilities.js*, vous disposez d'une mise en œuvre clé en main des fonctionnalités de glisser-déposer, de redimensionnement et de rotation.

En résumé

Du point de vue de l'utilisateur, l'interface de l'application constitue votre application. Une mauvaise conception de l'interface utilisateur, comme dans le cas du jeu DollarStash, risque de couler votre application avant que vous ayez l'opportunité de l'améliorer ou de la corriger. Nous l'avons expliqué dans ce chapitre, une conception avisée de l'interface tire parti des trois principes de base suivants :

- Ne pas surprendre l'utilisateur. Les interactions doivent déclencher les comportements attendus.
- Rendre l'interface intuitive. L'utilisateur ne doit pas avoir besoin de lire une documentation pour utiliser l'application.
- Ne pas demander aux appareils plus qu'ils ne peuvent réaliser. Les ressources processeur et mémoire étant limitées, elles doivent être utilisées de manière appropriée.

Si vous respectez ces règles de base et le HIG d'Apple, votre application aura des chances de succès beaucoup plus élevées. Bien que ces règles puissent sembler strictes, elles laissent le champ libre à la créativité, comme l'a montré le module de glisser-déposer, de redimensionnement et de rotation.

Ce module, s'il est créé et employé intelligemment, permet d'améliorer énormément l'expérience de l'utilisateur avec votre application. En effet, il se fonde sur les possibilités multitouch de l'iPhone et de l'iPod Touch, ainsi que sur le HIG d'Apple.

Openmirrors.com



4

GPS, accéléromètre et autres fonctions natives avec QuickConnectiPhone

L'iPhone propose des fonctionnalités uniques que vous pouvez exploiter dans vos applications. Il est notamment possible de faire vibrer le téléphone, de jouer des sons système, d'accéder aux données de l'accéléromètre et de consulter les informations de localisation GPS. Durant le développement de l'application, vous pouvez également écrire des messages de débogage sur la console Xcode. Ces fonctionnalités ne sont pas réservées aux applications écrites en Objective-C. Les applications hybrides peuvent les utiliser depuis du code JavaScript. La première section de ce chapitre explique comment employer les fonctionnalités natives de l'iPhone avec l'API JavaScript de QuickConnect. La deuxième présente le code Objective-C qui sous-tend la bibliothèque JavaScript QuickConnect.

Section 1 : activation de l'appareil en JavaScript

L'iPhone change la donne. Les développeurs d'applications peuvent en effet accéder à ses composants matériels, comme l'accéléromètre. L'exploitation des capacités natives de l'iPhone permet de créer des applications innovantes. Leur comportement peut dépendre des données de l'accéléromètre ou des informations de localisation GPS. Vous pouvez décider du moment où le téléphone vibre ou joue certains sons.

Le fichier *com.js* de QuickConnectiPhone définit une fonction qui permet d'accéder facilement et simplement à ces possibilités. Une application invoque `makeCall` pour envoyer des requêtes au téléphone. Son premier paramètre est une chaîne de commande, le second contient les paramètres nécessaires à l'exécution de la commande, indiqués sous forme d'une chaîne de caractères. Le Tableau 4.1 recense chaque commande standard, les paramètres correspondants et le comportement obtenu.

Tableau 4.1 : Commandes reconnues par `makeCall`

Chaîne de commande	*Chaîne de message*	*Comportement*
`logMessage`	Les informations à afficher sur la console Xcode.	Le message apparaît sur la console Xcode à l'exécution du code.
`rec`	Une chaîne JSON qui représente un tableau JavaScript dont le premier élément indique le nom du fichier audio à créer. Le second élément est `start` ou `stop` selon que vous souhaitez démarrer ou arrêter l'enregistrement des données audio.	Un fichier audio *caf*, dont le nom est défini par la chaîne de message, est créé.
`play`	Une chaîne JSON qui représente un tableau JavaScript dont le premier élément indique le nom du fichier audio à lire. Le second élément est `start` ou `stop` selon que vous souhaitez démarrer ou arrêter la lecture du fichier audio.	Le fichier audio *caf*, s'il existe, est joué au travers des haut-parleurs de l'appareil ou des écouteurs.
`loc`	Aucune.	Le comportement Core Location de l'appareil est déclenché et les informations de latitude, de longitude et d'altitude sont retournées à l'application JavaScript.
`playSound`	–1	Le téléphone vibre.

Openmirrors.com

Tableau 4.1 : Commandes reconnues par `makeCall` *(suite)*

Chaîne de commande	*Chaîne de message*	*Comportement*
`playSound`	0	Le fichier audio laser est joué.
`showDate`	`DateTime`.	Le sélecteur de date et d'heure natif est affiché.
`showDate`	`Date`.	Le sélecteur de date natif est affiché.

L'application DeviceCatalog comprend un bouton `Vibrate` qui, lorsque l'utilisateur clique dessus, fait vibrer le téléphone. Le gestionnaire d'événements `onclick` de ce bouton est la fonction `vibrateDevice` présentée ci-après. Elle appelle `makeCall` en lui passant la commande `playSound` avec le paramètre `–1`, ce qui déclenche le vibreur du téléphone. La commande `playSound` est utilisée car l'iPhone traite les vibrations et les sons système brefs comme des sons.

```
function vibrateDevice(event)
{
    // Le paramètre -1 déclenche le vibreur du téléphone.
    makeCall("playSound", -1);
}
```

Puisque le vibreur et les sons système sont traités de la même manière, la lecture d'un son système est quasiment identique au déclenchement du vibreur. Le gestionnaire d'événements `onclick` du bouton `Sound` se nomme `playSound`. Vous le constatez dans le code suivant, la seule différence entre cette fonction et la fonction `vibrateDevice` se trouve dans le second paramètre.

```
function playSound(event)
{
    // Le paramètre 0 demande au téléphone d'émettre le son d'un laser.
    makeCall("playSound", 0);
}
```

Lorsque le second paramètre est `0`, le fichier *laser.wav* inclus dans les ressources du projet DeviceCatalog est joué en tant que son système. Les sons système ne doivent pas durer plus de 5 secondes et ne sont pas joués comme les autres sons "normaux". Les fichiers audio plus longs sont lus avec la commande `play`, que nous examinerons plus loin dans cette section.

La fonction `makeCall` employée par les fonctions précédentes est écrite en JavaScript. Elle est constituée de deux parties. La première place le message dans une file d'attente s'il ne peut pas être envoyé immédiatement. La seconde envoie le message au code Objective-C

sous-jacent en vue de son traitement. Pour passer le message, une URL inexistante, `call`, est affectée à la propriété `window.location`, et les paramètres de la fonction sont précisés dans l'URL.

```
function makeCall(command, dataString)
{
    var messageString = "cmd="+command+"&msg="+dataString;
    if(storeMessage || !canSend){
        messages.push(messageString);
    }
    else{
        storeMessage = true;
        window.location = "call?"+messageString;
    }
}
```

Avec ce type d'URL, un message comprenant l'URL et ses paramètres est envoyé à un composant Objective-C du framework QuickConnectiPhone. Ce composant interrompt le chargement de la nouvelle page et passe la commande et le message reçus au code de traitement fourni par le framework. La Section 2 détaille ce fonctionnement.

Les commandes `playSound`, `logMessage`, `rec` et `play` sont unidirectionnelles. Autrement dit, les communications se font depuis JavaScript vers Objective-C, sans que des données ne soient retournées. Les autres commandes unidirectionnelles standard déclenchent la transmission de données depuis les composants Objective-C vers le code JavaScript.

La transmission des données au code JavaScript peut se faire de deux manières. La première est employée pour passer les informations d'accélération dans les coordonnées x, y et z par un appel à la fonction JavaScript `handleRequest` décrite au Chapitre 2. Cet appel utilise la commande `accel` et les coordonnées x, y et z passées sous forme d'un objet JavaScript depuis les composants Objective-C du framework.

Le fichier *mappings.js* montre que la commande `accel` est associée à la fonction `displayAccelerationVCF` :

```
mapCommandToVCF('accel', displayAccelerationVCF);
```

Ainsi, `displayAccelerationVCF` est appelée chaque fois que l'accéléromètre détecte un mouvement. Cette fonction est responsable du traitement de tous les événements d'accélération. Dans l'application DeviceCatalog, elle place simplement les valeurs x, y et z dans un élément HTML `<div>`. Vous devez modifier cette fonction pour exploiter ces valeurs dans votre application.

La seconde manière de renvoyer des données au code JavaScript se fonde sur un appel à la fonction JavaScript `handleJSONRequest`. Elle fonctionne de manière semblable à la fonction

Openmirrors.com

`handleRequest` décrite au Chapitre 2, mais elle attend une chaîne JSON en second paramètre. Cette fonction est une façade pour la fonction `handleRequest`. Le code suivant montre qu'elle convertit simplement la chaîne JSON indiquée en un objet JavaScript, puis passe la commande et le nouvel objet à la méthode `handleRequest`. Cette manière de transférer des données est utilisée pour la demande de localisation GPS initiée par un appel à `makeCall("loc")` et la demande d'affichage d'un sélecteur de date et d'heure.

```
function handleJSONRequest(cmd, parametersString){
    var paramsArray = null;
    if(parametersString){
        var paramsArray = JSON.parse(parametersString);
    }
    handleRequest(cmd, paramsArray);
}
```

Dans les deux cas, les données résultantes sont converties en une chaîne JSON et passées à `handleJSONRequest`. Pour de plus amples informations concernant le format JSON, consultez l'Annexe A.

Puisqu'il existe des bibliothèques JSON pour JavaScript et Objective-C, ce format constitue une bonne solution pour échanger des informations complexes entre ces deux langages dans une application. Il est par exemple employé par les gestionnaires `onclick` pour le démarrage et l'arrêt de l'enregistrement et de la lecture des fichiers audio.

La fonction `playRecording` est caractéristique des gestionnaires associés aux boutons de l'interface utilisateur qui activent les comportements de l'appareil. L'exemple suivant montre qu'elle crée un tableau JavaScript, ajoute deux valeurs, convertit le tableau en une chaîne JSON et appelle la fonction `makeCall` avec la commande `play`.

```
function playRecording(event)
{
    var params = new Array();
    params[0] = "recordedFile.caf";
    params[1] = "start";
    makeCall("play", JSON.stringify(params));
}
```

Pour stopper la lecture, un appel à `makeCall` avec la commande `play` est également effectué, mais le paramètre `start` est remplacé par `stop`. La fonction `terminatePlaying` définie dans le fichier *main.js* met en œuvre cette procédure.

Le démarrage et l'arrêt de l'enregistrement d'un fichier audio sont codés de manière semblable à `playRecording` et à `terminatePlaying`, excepté que la commande `play` est

remplacée par `rec`. Puisque le contrôle de ces deux fonctionnalités connexes se fait de la même façon, il est plus facile d'ajouter ces comportements dans une application.

Nous l'avons vu précédemment dans cette section, certains comportements de l'appareil, comme le vibreur, nécessitent une communication depuis le code JavaScript vers les composants Objective-C. D'autres, comme la lecture des coordonnées GPS ou des résultats d'un sélecteur, exigent une communication dans les deux directions. La Figure 4.1 présente l'application DeviceCatalog qui affiche les informations GPS.



Figure 4.1
L'application DeviceCatalog affiche les informations GPS.

À l'instar des exemples unidirectionnels déjà présentés, la communication débute dans le code JavaScript de l'application. La fonction `getGPSLocation` définie dans le fichier *main.js* initie la communication en appelant `makeCall`. Notez que dans les exemples précédents `makeCall` ne retourne aucune valeur. Elle utilise un protocole asynchrone pour communiquer avec le côté Objective-C de la bibliothèque, même lorsque la communication est bidirectionnelle.

```
function getGPSLocation(event)
{
    document.getElementById('locDisplay').innerText = '';
    makeCall("loc");
}
```

Openmirrors.com

Puisque la communication est asynchrone, comme dans le cas d'AJAX, une fonction de rappel doit être créée et invoquée pour recevoir les informations GPS. Dans le framework QuickConnectiPhone, cette opération est réalisée en associant la commande `showLoc` à une fonction :

```
mapCommandToVCF('showLoc', displayLocationVCF);
```

Dans cet exemple, elle est associée à la fonction de contrôle de l'affichage nommée `displayLocationVCF`, qui se contente d'afficher la localisation GPS courante dans un `<div>`. Bien évidemment, les valeurs obtenues peuvent également servir à calculer des distances, qui seront enregistrées dans une base de données ou envoyées à un serveur à l'aide de `ServerAccessObject`, décrit au Chapitre 8.

```
function displayLocationVCF(data, paramArray){
    document.getElementById('locDisplay').innerText = 'latitude: '
            +paramArray[0]+'\nlongitude: '+paramArray[1]+'\naltitude: '
            +paramArray[2];
}
```

L'affichage d'un sélecteur, par exemple le sélecteur de date et d'heure standard, et la présentation de la sélection se font de manière semblable à l'exemple précédent. La procédure débute également par un appel JavaScript au code de gestion de l'appareil. Dans ce cas, la fonction gestionnaire du bouton se nomme `showDateSelector` ; elle est définie dans le fichier *main.js*.

```
function showDateSelector(event)
{
    makeCall("showDate", "DateTime");
}
```

Comme pour les informations GPS, une association doit être créée. Elle lie la commande `showPickResults` à la fonction de contrôle de l'affichage `displayPickerSelectionVCF` :

```
mapCommandToVCF('showPickResults', displayPickerSelectionVCF);
```

La fonction associée à la commande insère les résultats de la sélection effectuée par l'utilisateur dans un simple `<div>`. Bien évidemment, les informations obtenues peuvent être employées de nombreuses autres manières.

```
function displayPickerSelectionVCF(data, paramArray){
    document.getElementById('pickerResults').innerHTML = paramArray[0];
}
```

Certaines utilisations de `makeCall`, notamment dans les premiers exemples de cette section, mettent en place une communication unidirectionnelle entre le code JavaScript et les composants Objective-C. Celles que nous venons de présenter emploient une communication bidirectionnelle. Il existe toutefois un autre type de communication unidirectionnelle : de l'appareil vers le code JavaScript. L'utilisation des informations de l'accéléromètre en est un exemple.

Le gestionnaire Objective-C pour les événements d'accélération (voir Section 2) effectue directement un appel à la fonction JavaScript `handleRequest` en passant la commande `accel`. Cette commande est associée à la fonction de contrôle de l'affichage `displayAccelerationVCF` :

```
mapCommandToVCF('accel', displayAccelerationVCF);
```

À l'instar des autres VCF, elle place les valeurs d'accélération dans un `<div>`.

```
function displayAccelerationVCF(data, param){
    document.getElementById('accelDisplay').innerText ='x: '
            +param.x+'\ny: '+param.y+'\nz: '+param.z;
}
```

À la différence des autres fonctions, elle reçoit dans son paramètre `param` non pas un tableau mais un objet. La Section 2 montre comment cet objet est créé à partir des informations venant du gestionnaire Objective-C des événements d'accélération.

Cette section a expliqué comment ajouter aux applications JavaScript les fonctionnalités de l'iPhone les plus demandées. La Section 2 décrit les parties Objective-C du framework qui le permettent.

Section 2 : activation de l'appareil en Objective-C

Cette section suppose que vous maîtrisiez Objective-C et que vous sachiez comment l'utiliser pour créer des applications pour l'iPhone. Si ce n'est pas le cas, lisez l'ouvrage *The iPhone Developer's Cookbook*, d'Erica Sadun. Si vous souhaitez uniquement utiliser le framework QuickConnectiPhone pour développer des applications JavaScript pour l'iPhone, vous n'êtes pas obligé de lire cette section.

Faire vibrer l'iPhone avec du code Objective-C est l'un des comportements les plus simples à réaliser. Si vous incluez le framework AudioToolbox dans les ressources du projet, la ligne suivante suffit :

```
AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
```

Openmirrors.com

La question qui se pose alors est : "Comment puis-je faire en sorte que la fonction `AudioServicesPlaySystemSound` soit appelée lorsque le code modifie l'attribut `location` de `UIWebView` ?"

La classe `QuickConnectViewController` implémente la méthode déléguée `shouldStartLoadWithRequest`. Puisque `QuickConnectViewController` est le délégué du `UIWebView` embarqué, nommé `aWebView`, cette méthode est invoquée chaque fois que le `UIWebView` embarqué voit son attribut `location` modifié. Le code suivant, ainsi que la ligne 90 du fichier *QuickConnectViewController.m*, montre l'affectation de ce délégué :

```
[aWebView setDelegate:self];
```

Le comportement de base de la fonction `shouldStartLoadWithRequest` est simple. Sa conception vous permet de décider si la nouvelle page demandée doit être réellement chargée. Le framework se fonde sur cette faculté pour interdire le chargement de la page lors des requêtes effectuées par les appels JavaScript décrits à la Section 1 et pour exécuter un autre code Objective-C.

La méthode `shouldStartLoadWithRequest` prend plusieurs paramètres :

- `curWebView`. Le `UIWebView` qui contient l'application JavaScript.
- `request`. Un `NSURLRequest` qui contient, entre autres, la nouvelle URL.
- `navigationType`. Un `UIWebViewNavigationType` qui peut être utilisé pour déterminer si la requête est le résultat d'un clic sur un lien ou si elle a été générée suite à une autre action.

```
-(BOOL)webView:(UIWebView *)curWebView
    shouldStartLoadWithRequest:(NSURLRequest *)request
        navigationType:(UIWebViewNavigationType)navigationType
```

L'URL composée par la fonction JavaScript `makeCall` pour faire vibrer le téléphone, `call?cmd=playSound&msg=-1`, est contenue dans l'objet `request` et peut être facilement retrouvée sous forme d'une chaîne de caractères en envoyant le message `URL` à cet objet. Ce message retourne un objet de type `NSURL`, auquel est ensuite passé le message `absoluteString`. Nous obtenons ainsi un pointeur `NSString` qui représente l'URL. Cette chaîne, enregistrée dans la variable `url`, peut être décomposée dans un tableau, en utilisant le point d'interrogation (`?`) comme caractère de séparation. Le tableau résultant contient des pointeurs `NSString`.

```
NSString *url = [[request URL] absoluteString];
NSArray *urlArray = [url componentsSeparatedByString:@"?"];
```

urlArray contient deux éléments. Le premier correspond à la partie call de l'URL, tandis que le second est la chaîne de commande cmd=playSound&msg=-1. Pour déterminer la commande et ses paramètres, dans ce cas –1, nous devons poursuivre l'analyse de la chaîne de commande. Pour cela, commandString est décomposée en fonction du caractère & dans le tableau nommé urlParamsArray.

```
NSString *commandString = [urlArray objectAtIndex:1];
NSArray *urlParamsArray = [commandString componentsSeparatedByString:@"&"];
// La commande est le premier paramètre dans l'URL.
cmd = [[[urlParamsArray objectAtIndex:0]
    componentsSeparatedByString:@"="] objectAtIndex:1];
```

Dans notre exemple, demander au téléphone de vibrer, le premier élément du tableau urlParamsArray est cmd=playSound et le second est msg=-1. Ainsi, en découpant les éléments de urlParamsArray avec le caractère = comme délimiteur, nous pouvons obtenir la commande à exécuter et son paramètre.

Les lignes 1 à 3 du code suivant enregistrent le paramètre passé dans l'URL en tant que valeur associée à la clé msg dans la variable parameterArrayString de type NSString. Puisque le code JavaScript qui a composé l'URL convertit toutes ses composantes au format JSON, ce NSString est un objet qui a été converti dans ce format. Cela inclut des nombres, comme dans notre exemple, des chaînes de caractères, des tableaux et d'autres paramètres passés depuis JavaScript. Par ailleurs, si des espaces ou d'autres caractères spéciaux apparaissent dans les données, UIWebView leur applique l'échappement dans l'URL. C'est pourquoi les lignes 6 à 8 suppriment l'échappement sur ces caractères spéciaux dans la chaîne JSON.

```
 1 NSString *parameterArrayString = [[[urlParamsArray
 2     objectAtIndex:1] componentsSeparatedByString:@"="]
 3     objectAtIndex:1];
 4 // Retirer tout encodage ajouté, comme l'échappement des caractères
 5 // effectué dans l'URL par UIWebView.
 6 parameterArrayString = [parameterArrayString
 7     stringByReplacingPercentEscapesUsingEncoding:
 8     NSASCIIStringEncoding];
 9 SBJSON *generator = [SBJSON alloc];
10 NSError *error;
11 paramsToPass = [[NSMutableArray alloc]
12     initWithArray:[generator
13     objectWithString:parameterArrayString
14     error:&error]];
```

Openmirrors.com

```
15 if([paramsToPass count] == 0){
16     // Si aucun tableau de données n'est envoyé, une chaîne a dû être passée
17     // comme seul paramètre.
18     [paramsToPass addObject:parameterArrayString];
19 }
20 [generator release];
```

Les lignes 9 à 14 convertissent la chaîne JSON `parameterArrayString` en un objet Objective-C `NSArray` natif. La ligne 9 instancie un objet `generator` de type `SBJSON`, auquel le message `objectWithString` est ensuite envoyé :

```
- (id)objectWithString:(NSString*)jsonrep error:(NSError**)error;
```

Ce message en plusieurs parties comprend une chaîne JSON, dans ce cas `parameterArrayString`, et un pointeur `error` de type `NSError`. Lorsqu'une erreur se produit au cours du processus de conversion, le pointeur `error` est affecté, sinon il vaut `nil`.

Dans ce cas, la valeur de retour de ce message est –1. Si un tableau JavaScript est converti en chaîne, la valeur de retour est un pointeur `NSArray`, ou, dans le cas d'une chaîne JavaScript, il s'agit d'un pointeur `NSString`. Lorsqu'un objet JavaScript de type personnalisé est passé, l'objet retourné est un pointeur `NSDictionary`.

À ce stade, puisque nous disposons de la commande et de ses paramètres, il est possible d'utiliser une instruction `if` ou `case` pour effectuer l'opération demandée. Toutefois, ces instructions conditionnelles ne constituent pas la solution optimale car elles doivent être modifiées chaque fois qu'une commande est ajoutée ou retirée. Au Chapitre 2, un problème comparable a été résolu dans la partie JavaScript de QuickConnectiPhone en mettant en place une fonction contrôleur frontal, nommée `handleRequest`, qui contient des appels aux implémentations des contrôleurs d'application. Puisque le problème est identique ici, une version Objective-C de `handleRequest` doit permettre de le résoudre. La Section 3 s'intéresse à la mise en œuvre des contrôleurs frontaux et des contrôleurs d'application en Objective-C. La ligne de code suivante obtient une instance de l'objet `QuickConnect` et lui passe le message `handleRequest withParameters`. Aucune autre opération n'est nécessaire dans la méthode déléguée `shouldStartLoadWithRequest`.

```
[[QuickConnect getInstance] handleRequest:cmd withParameters:paramsToPass];
```

Puisque nous utilisons le message `handleRequest` des objets `QuickConnect`, nous avons besoin d'un mécanisme pour lier les commandes à la fonctionnalité requise, à la manière du Chapitre 2 en JavaScript. L'objet `QCCommandMappings`, déclaré dans les fichiers *QCCommandMappings.m* et *.h* du groupe `QCObjC`, contient toutes les associations pour les objets de contrôle métier (BCO, *Business Control Object*) et les objets de contrôle de l'affichage (VCO, *View Control Object*) de cet exemple.

Le code suivant correspond à la méthode `mapCommands` de l'objet `QCCommandMappings` qui est invoquée au démarrage de l'application. Elle reçoit l'implémentation d'un contrôleur d'application, qui crée les associations entre les commandes et les fonctionnalités. La Section 3 expliquera le code du message `mapCommandToVCO` et de l'appel à `mapCommands`.

```
1  + (void) mapCommands:(QCAppController*)aController{
2      [aController mapCommandToVCO:@"logMessage" withFunction:@"LoggingVCO"];
3      [aController mapCommandToVCO:@"playSound" withFunction:@"PlaySoundVCO"];
4      [aController mapCommandToBCO:@"loc" withFunction:@"LocationBCO"];
5      [aController mapCommandToVCO:@"sendloc" withFunction:@"LocationVCO"];
6      [aController mapCommandToVCO:@"showDate" withFunction:@"DatePickerVCO"];
7      [aController mapCommandToVCO:@"sendPickResults" withFunction:@"Pick-
       ➥ResultsVCO"];
8      [aController mapCommandToVCO:@"play" withFunction:@"PlayAudioVCO"];
9      [aController mapCommandToVCO:@"rec" withFunction:@"RecordAudioVCO"];
10 }
```

La ligne 3 du code précédent est en rapport avec le déclenchement du vibreur. Nous l'avons vu précédemment dans cette section, la commande reçue depuis la partie JavaScript de l'application se nomme `playSound`. En passant cette commande en premier paramètre du message `mapCommandToVCO` et `PlaySoundVCO` comme paramètre de la seconde partie, `withFunction`, nous créons un lien qui conduit le contrôleur d'application à envoyer un message `doCommand` avec le paramètre `–1` à la classe `PlaySoundVCO`. Vous le constatez, toutes les autres commandes de l'exemple DeviceCatalog envoyées depuis JavaScript sont associées dans cette fonction.

Le code de `PlaySoundVCO` à laquelle la commande `playSound` est associée se trouve dans les fichiers *PlaySoundVCO.m* et *PlaySoundVCO.h*. La méthode `doCommand` prend en charge tous les comportements de l'objet.

Pour jouer un son système, un son prédéfini (seule la vibration est définie au moment de l'écriture de ces lignes) doit être utilisé ou un son système doit être généré à partir d'un fichier audio. La méthode `doCommand` de la classe `PlaySoundVCO` illustre ces deux fonctionnements.

```
1  + (id) doCommand:(NSArray*) parameters{
2      SystemSoundID aSound =
3              [((NSNumber*)[parameters objectAtIndex:1]) intValue];
4      if(aSound == -1){
5          aSound = kSystemSoundID_Vibrate;
6      }
7      else{
8          NSString *soundFile =
9                  [[NSBundle mainBundle] pathForResource:@"laser"
```

Openmirrors.com

```
10                  ofType:@"wav"];
11          NSURL *url = [NSURL fileURLWithPath:soundFile];
12          // Si la lecture du fichier audio est trop longue,
13          // nous recevons une erreur -1500.
14          OSStatus error = AudioServicesCreateSystemSoundID(
15                  (CFURLRef) url, &aSound );
16      }
17      AudioServicesPlaySystemSound(aSound);
18      return nil;
19  }
```

Vous pouvez le voir à la ligne 4 de l'exemple précédent, si le paramètre d'indice 1 a la valeur –1, la variable `SystemSoundID aSound` est fixée à la valeur prédéfinie `kSystemSoundID_Vibrate`. Sinon un son système est créé à partir du fichier *laser.wav* qui se trouve dans le groupe `Resources` de l'application et un identifiant généré pour ce nouveau son est affecté à la variable `aSound`.

Dans tous les cas, la fonction C `AudioServicesPlaySystemSound` est appelée, le son est joué ou l'appareil vibre. Si l'appareil est un iPod Touch, les demandes de vibration sont ignorées. Dans une application réelle qui dispose de plusieurs sons, il est facile d'étendre cette fonction en passant d'autres nombres désignant les sons à jouer.

Puisque `SystemSoundID` est de type numérique, les sons système doivent être générés au début de l'application et leurs identifiants doivent être passés à la partie JavaScript de l'application en vue d'une utilisation ultérieure. Cela évite la charge nécessaire à la création du son système chaque fois qu'il doit être joué et, par conséquent, l'utilisateur ne constate aucun retard dans la lecture du son.

Puisque nous connaissons à présent la procédure qui permet de passer des commandes depuis JavaScript à Objective-C et la manière de faire vibrer l'appareil ou de jouer un son bref, il est facile de comprendre le passage d'une commande à Objective-C et le retour des résultats à la partie JavaScript de l'application.

En raison de la similitude de ces communications, la détection de la localisation GPS, très utilisée dans les applications pour l'iPhone, nous servira d'exemple. Elle se fonde sur les possibilités de communication bidirectionnelle entre JavaScript et Objective-C apportées par le framework QuickConnectiPhone.

À l'instar de la gestion des commandes envoyées depuis le framework JavaScript, nous avons besoin d'un mécanisme pour lier la commande `loc` et ainsi pouvoir obtenir les données et la réponse renvoyées.

```
[aController mapCommandToBCO:@"loc" withFunction:@"LocationBCO"];
[aController mapCommandToVCO:@"sendloc" withFunction:@"LocationVCO"];
```

Dans ce cas, il existe deux associations : la première avec un BCO, la seconde avec un VCO. À l'instar des BCF et VCF vues au Chapitre 2, les BCO permettent d'obtenir les données, les VCO, de les afficher.

Puisque les BCO d'une commande donnée sont exécutés par le framework QuickConnect-iPhone avant tous les VCO, un message `doCommand` est tout d'abord envoyé à la classe `LocationBCO` de manière à obtenir et à retourner les données GPS. La méthode `doCommand` suivante appartient à la classe `LocationBCO`. Elle effectue les appels permettant à l'appareil de déterminer sa localisation GPS.

```
+ (id) doCommand:(NSArray*) parameters{
    QuickConnectViewController *controller =
        (QuickConnectViewController*)[parameters objectAtIndex:0];
    [[controller locationManager] startUpdatingLocation];
    return nil;
}
```

Cette méthode sollicite le matériel de localisation GPS en obtenant le premier élément du tableau passé en paramètre à la méthode et en lui demandant d'activer le matériel. Le framework fixe toujours le premier paramètre à `QuickConnectViewController` afin que les BCO ou les VCO associés aux commandes puissent s'en servir en cas de besoin. Dans tous les BCO et VCO Objective-C, les paramètres provenant de JavaScript débutent à l'indice 1.

L'objet `QuickConnectViewController` comprend un attribut `CLLocationManager`, nommé `locationManager`, qui est activé et désactivé en fonction de l'application. Il est important que ce gestionnaire ne s'exécute pas plus longtemps que nécessaire car il consomme beaucoup d'énergie de la batterie. Par conséquent, le code précédent active le dispositif de localisation en lui envoyant un message `startUpdatingLocation` chaque fois que les informations sont demandées. Le dispositif est désactivé dès que la localisation est déterminée.

Les objets `CLLocationManager` se comportent de manière asynchrone. Autrement dit, lorsqu'une demande de localisation est effectuée, une fonction de rappel prédéfinie est invoquée après leur obtention. Cette fonction prédéfinie permet d'accéder au gestionnaire de localisation et à deux localisations : une localisation précédemment déterminée et la localisation actuelle.

Le gestionnaire de localisation affine progressivement la localisation de l'appareil. Au cours de cette procédure, il appelle plusieurs fois `didUpdateToLocation`. L'exemple de code suivant calcule le temps nécessaire à déterminer la nouvelle localisation. La ligne 9 vérifie s'il est inférieur à cinq secondes et, dans l'affirmative, poursuit la localisation.

```
1  (void)locationManager:(CLLocationManager *)manager
2      didUpdateToLocation:(CLLocation *)newLocation
```

Openmirrors.com

```
 3     fromLocation:(CLLocation *)oldLocation
 4 {
 5     // Si l'événement est relativement récent, désactiver les actualisations
       ➥pour économiser la batterie.
 6     NSDate* eventDate = newLocation.timestamp;
 7     NSTimeInterval howRecent =
 8             [eventDate timeIntervalSinceNow];
 9     if (abs(howRecent) < 5.0){
10         [manager stopUpdatingLocation];
11         NSMutableArray *paramsToPass =
12                 [[NSMutableArray alloc] initWithCapacity:2];
13         [paramsToPass addObject:self];
14         [paramsToPass addObject:newLocation];
15         [[QuickConnect getInstance]
16                 handleRequest:@"sendloc"
17                 withParameters:paramsToPass];
18     }
19     // Sinon, ignorer l'événement et passer au suivant.
20 }
```

Après avoir terminé la localisation, le code envoie un message à la classe contrôleur frontal de `QuickConnect` en lui indiquant qu'elle doit traiter une requête `sendloc` avec `QuickConnectViewController`, `self` et la nouvelle localisation passée en paramètre supplémentaire.

La commande `sendloc` est associée au gestionnaire `LocationVCO` dont la méthode `doCommand` est reproduite ci-après. Cette méthode obtient le `UIWebView` nommé `webView` à partir du `QuickConnectViewController` qui a demandé initialement les informations de localisation GPS. Ces informations sont ensuite placées dans le `NSArray` nommé `passingArray`.

Pour retourner les informations GPS à l'objet `webView`, le `NSArray` qui les contient doit être converti en une chaîne JSON. La classe `SBJSON`, déjà employée précédemment pour créer un tableau à partir d'une chaîne JSON, est à présent utilisée pour créer un `NSString` à partir du `NSArray` (lignes 21 et 22).

```
1  + (id) doCommand:(NSArray*) parameters{
2      QuickConnectViewController *controller =
3              (QuickConnectViewController*)[parameters
4              objectAtIndex:0];
5      UIWebView *webView = [controller webView];
6      CLLocation *location = (CLLocation*)[parameters
7              objectAtIndex:1];
8
9      NSMutableArray *passingArray = [[NSMutableArray alloc]
```

```
10              initWithCapacity:3];
11      [passingArray addObject: [NSNumber numberWithDouble:
12              location.coordinate.latitude]];
13      [passingArray addObject: [NSNumber numberWithDouble:
14              location.coordinate.longitude]];
15      [passingArray addObject: [NSNumber numberWithFloat:
16              location.altitude]];
17
18      SBJSON *generator = [SBJSON alloc];
19
20      NSError *error;
21      NSString *paramsToPass = [generator
22              stringWithObject:passingArray error:&error];
23      [generator release];
24      NSString *jsString = [[NSString alloc]
25              initWithFormat:@"handleJSONRequest('showLoc', '%@')",
26              paramsToPass];
27      [webView
28              stringByEvaluatingJavaScriptFromString:jsString];
29      return nil;
30  }
```

Après avoir converti les informations de localisation GPS en une chaîne JSON qui représente un tableau de nombre, un appel au moteur JavaScript est effectué depuis l'objet `webView`. Pour cela, nous commençons par créer un `NSString` qui contient le code JavaScript à exécuter. Dans cet exemple, il s'agit de `handleJSONRequest`, avec `showLoc` pour commande et les informations GPS au format JSON comme chaîne. Nous l'avons vu à la Section 1, cette requête fait apparaître les données GPS dans un `<div>` de la page HTML affichée.

Ayant à présent étudié cet exemple, vous pouvez examiner `DatePickerVCO` et `PickResultsVCO` dans l'application DeviceCatalog et voir comment cette même approche est employée pour afficher les sélecteurs de date et d'heure standard disponibles en Objective-C. Même si des sélecteurs prédéfinis sont disponibles en JavaScript dans `UIWebView`, ils ne sont pas aussi jolis que ceux fournis par Objective-C. En utilisant les sélecteurs standard et ceux que vous pourriez définir, l'expérience de l'utilisateur avec votre application hybride n'en sera que meilleure.

Openmirrors.com

Section 3 : implémentation Objective-C de l'architecture de QuickConnectiPhone

Le code présenté aux Sections 1 et 2 dépend énormément de l'implémentation en Objective-C de l'architecture expliquée au Chapitre 2. Cette section décrit cette implémentation. Pour une explication complète de chaque composant, consultez le Chapitre 2, qui s'intéresse à l'implémentation JavaScript.

À l'instar de l'implémentation JavaScript, toutes les demandes de comportement applicatif se font au travers d'un contrôleur frontal. Celui-ci est mis en œuvre par la classe `QuickConnect`, dont le code source se trouve dans les fichiers *QuickConnect.m* et *QuickConnect.h*. Puisque les messages envoyés à `QuickConnect` peuvent provenir de différents endroits de l'application, cette classe est un singleton.

Les classes singletons sont écrites de manière qu'une seule instance de la classe soit allouée dans une application. Lorsqu'elles sont bien implémentées, il existe toujours un moyen d'obtenir un pointeur sur cet objet unique depuis n'importe quel point de l'application. Avec l'objet singleton `QuickConnect`, cela se fait au travers de la méthode de classe `getInstance`, qui retourne l'unique instance de `QuickConnect` allouée lors de la première invocation de cette méthode.

Puisqu'il s'agit d'une méthode de classe, un message `getInstance` peut être envoyé à la classe sans instancier un objet `QuickConnect`. Son invocation retourne un pointeur sur l'instance de `QuickConnect` allouée. Le code suivant montre que cette opération est réalisée en affectant une instance de la classe à un pointeur `QuickConnect` défini de manière statique.

```
+ (QuickConnect*)getInstance{
    // Puisque cette ligne est déclarée avec static,
    // elle est exécutée une seule fois.
    static QuickConnect *mySelfQC = nil;

    @synchronized([QuickConnect class]) {
        if (mySelfQC == nil) {
            mySelfQC = [QuickConnect singleton];
             [mySelfQC init];
        }
    }
    return mySelfQC;
}
```

Le message `singleton` envoyé avant `init` utilise le comportement défini dans la superclasse `FTSWAbstractSingleton` de l'objet `QuickConnect`. Cette superclasse met en

œuvre le comportement de singleton, comme redéfinir les méthodes `new`, `clone` et d'autres que le programmeur pourrait utiliser par erreur pour allouer une autre instance de `QuickConnect`. C'est pourquoi seule la méthode `getInstance` est en mesure de créer et d'utiliser un objet `QuickConnect`. À l'instar de tous les objets bien écrits en Objective-C, après son allocation, l'objet `QuickConnect` doit être initialisé.

L'allocation et l'initiation de l'objet ont lieu uniquement si aucun objet `QuickConnect` n'a été affecté à l'attribut `mySelfQC`. Par ailleurs, en raison de la synchronisation du contrôle de l'existence de l'objet `QuickConnect` instancié, ces opérations sont sûres vis-à-vis des threads.

`- (void) handleRequest: (NSString*) aCmd withParameters:(NSArray*) parameters` est une autre méthode de la classe `QuickConnect`. De même que la fonction JavaScript `handleRequest(aCmd, parameters)` du Chapitre 2, elle représente le mécanisme permettant d'exécuter dans votre application la fonctionnalité demandée.

Une chaîne de commande et un tableau de paramètres sont passés à la méthode. Dans l'exemple suivant, les lignes 3 à 9 montrent qu'une suite de messages est envoyée au contrôleur d'application. Les lignes 3 et 4 commencent par exécuter les VCO associés à la commande. Si la commande et les paramètres passent avec succès la validation, les BCO associés à la commande sont exécutés *via* un message `dispatchToBCO`. Ce message retourne un `NSMutableArray` qui contient les données du tableau `parameters` d'origine, auxquelles ont été ajoutées celles accumulées au cours des invocations des BCO.

```
 1 - (void) handleRequest: (NSString*) aCmd
 2         withParameters:(NSArray*) parameters{
 3     if([self->theAppController dispatchToValCO:aCmd
 4             withParameters:parameters] != nil){
 5         NSMutableArray *newParameters =
 6                 [self->theAppController dispatchToBCO:aCmd
 7                 withParameters:parameters];
 8         [self->theAppController dispatchToVCO:aCmd
 9                 withParameters:newParameters];
10     }
11 }
```

Lorsque l'appel à `dispatchToBCO:withParameters` est terminé, un message `dispatchToVCO:withParameters` est envoyé. Les VCO associés à la commande indiquée sont alors exécutés.

En utilisant la méthode `handleRequest:withParameters` pour toutes les demandes de fonctionnalités, chaque requête passe par le processus en trois étapes suivant :

- validation ;

Openmirrors.com

- exécution des règles métier (BCO) ;
- exécution des modifications de l'affichage (VCO).

Comme dans l'implémentation JavaScript, chaque méthode `dispatchTo` est une façade. Dans ce cas, la méthode Objective-C sous-jacente est `dispatchToCO:withParameters`.

Cette méthode commence par obtenir tous les objets de commande associés à la commande `default` à partir du paramètre `aMap`. Il contient des BCO, des VCO ou des ValCO, selon la méthode façade invoquée. Ces objets de commande par défaut, s'ils existent, sont obtenus et utilisés pour toutes les commandes. Si vous souhaitez que certains objets de commande soient employés avec toutes les commandes, il est inutile de les associer tous à chaque commande individuelle. Il suffit de les associer une fois à la commande `default`.

Pour utiliser les objets de commandes obtenus, un message `doCommand` doit leur être envoyé. Les lignes 19 à 23 de l'exemple suivant montre ce message obtenu comme un sélecteur et le passage du message `performSelector`. Cela déclenche l'exécution du message `doCommand` que vous avez implémenté dans votre `QCCommandObject`.

```
1 - (id) dispatchToCO: (NSString*)command withParameters:
2         (NSArray*)parameters andMap:(NSDictionary*)aMap{
3     // Créer un tableau modifiable qui contient tous
4     // les paramètres existants.
5     NSMutableArray *resultArray;
6     if(parameters == nil){
7         resultArray = [[NSMutableArray alloc]
8         initWithCapacity:0];
9     }
10    else{
11        resultArray = [NSMutableArray
12        arrayWithArray:parameters];
13    }
14    // Affecter quelque chose à result afin que
15    // l'exécution se poursuive même s'il n'existe
16    // aucune association.
17    id result = @"Continue";
18    if([aMap objectForKey:@"default"] != nil){
19        SEL aSelector = @selector(doCommand);
20        while((result = [((QCCommandObject*)
21                [aMap objectForKey:@"default"])
22                performSelector:aSelector
23                withObject:parameters]) != nil){
```

```
            if(aMap == self->businessMap){
                [resultArray addObject:result];
            }
        }
    }
    // Si tous les appels de méthode des objets de commande par défaut
    // retournent une valeur, exécuter les objets personnalisés.
    if(result != nil && [aMap objectForKey:command] !=
            nil){
        NSArray *theCommandObjects =
                [aMap objectForKey:command];
        int numCommandObjects = [theCommandObjects count];
        for(int i = 0; i < numCommandObjects; i++){
            QCCommandObject *theCommand =
                    [theCommandObjects objectAtIndex:i];
            result = [theCommand doCommand:parameters];
            if(result == nil){
                resultArray = nil;
                break;
            }
            if(aMap == self->businessMap){
                [resultArray addObject:result];
            }
        }
    }
    if(aMap == self->businessMap){
        return resultArray;
    }
    return result;
}
```

Après l'envoi des messages `doCommand` à tous les `QCCommandObject` associés à la commande `default`, la même opération est effectuée pour les `QCCommandObject` que vous avez associés à la commande passée en paramètre à la méthode. L'existence de ces `QCCommandObject` se justifie de la même façon que les fonctions de contrôle dans l'implémentation JavaScript. Puisque les `QCCommandObject` contiennent le code du comportement de l'application, un exemple permettra de mieux comprendre leur création.

Puisque `QCCommandObject` est la classe mère de `LoggingVCO`, celle-ci doit implémenter la méthode `doCommand`. L'intégralité du contenu du fichier *LoggingVCO.m* de l'application DeviceCatalog est donnée ci-après. Sa méthode `doCommand` écrit dans le journal de l'application en cours d'exécution. Ce VCO journalise les messages de débogage générés par le code JavaScript de l'application. La Figure 4.2 présente les appels requis.

Openmirrors.com

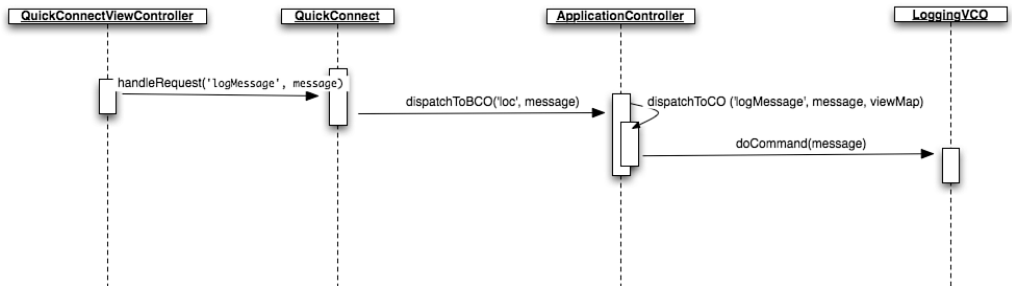


Figure 4.2
Ce diagramme de séquence montre les méthodes Objective-C invoquées pour traiter une requête de journalisation d'un message de débogage JavaScript.

La méthode `doCommand` de la classe `LoggingVCO` est courte. Toutes les méthodes `doCommand` des différents types d'objets de commande doivent toujours être brèves. Elles doivent faire une seule chose et la faire bien. Si vous constatez que la méthode `doCommand` en cours de développement devient longue, vous devez envisager son découpage en composants logiques et la création de plusieurs classes d'objets de commande. En effet, lorsque ces méthodes deviennent longues, il est probable qu'elles font plus d'une chose.

Dans l'exemple suivant, `LoggingVCO` s'occupe uniquement de la journalisation des messages sur la console de débogage de Xcode. Bien évidemment, ce petit composant peut être réutilisé avec de nombreuses commandes et d'autres objets de commande.

Le comportement de ce VCO est mis en œuvre par une seule ligne qui exécute la fonction `NSLog`. Le premier objet du tableau `parameters` est concaténé à une chaîne statique et affiché.

```
#import "LoggingVCO.h"

@implementation LoggingVCO

+ (id) doCommand:(NSArray*) parameters{
    NSLog(@"JavaScriptMessage: %@",
            [parameters objectAtIndex:1]);
    return nil;
}

@end
```

Pour que la journalisation soit effective, il faut une association entre la commande `logMessage` et la classe `LoggingVCO`. Comme dans l'implémentation JavaScript, cette association se fait en ajoutant `logMessage`, en tant que clé, et le nom de la classe `LoggingVCO`,

en tant que valeur, à une mappe. Le code suivant, tiré du fichier *QCCommandMappings.m* de l'application DeviceCatalog, associe `logMessage` à la classe `LoggingVCO`.

```
[aController mapCommandToVCO:@"logMessage" withFunction:@"LoggingVCO"];
```

Le contrôleur d'application reçoit le message `mapCommandToVCO:withFunction`, dans lequel le premier paramètre est la commande et le second, le nom du VCO. Cette méthode, ainsi que d'autres comme elle, utilisée pour associer les types d'objets de commande, sont des façades qui appellent la méthode `mapCommandToCO` sous-jacente.

La méthode `mapCommandToCO` permet d'associer plusieurs objets de commande à une seule commande en utilisant un `NSMutableArray`. Ce tableau contient les objets `Class` qui correspondent aux noms de classe passés en second paramètre. Le code suivant présente l'implémentation de la méthode `mapCommandToCO`.

```
- (void) mapCommandToCO:(NSString*)aCommand
        withFunction:(NSString*)aClassName
        toMap:(NSMutableDictionary*)aMap{
    NSMutableArray *controlObjects =
            [[aMap objectForKey:aCommand] retain];
    if(controlObjects == nil){
        NSMutableArray *tmpCntrlObjs =
                [[NSMutableArray alloc] initWithCapacity:1];
        [aMap setObject: tmpCntrlObjs forKey:aCommand];
        controlObjects = tmpCntrlObjs;
        [tmpCntrlObjs release];
    }
    // Obtenir la classe de l'objet de contrôle
    // pour le nom indiqué et ajouter un objet
    // de ce type au tableau de la commande.
    Class aClass = NSClassFromString(aClassName);
    if(aClass != nil){
        [controlObjects addObject:aClass];
    }
    else{
        MESSAGE(@"Erreur : classe %@ non trouvée.
            Vérifiez qu'elle existe sous ce nom et recommencez.");
    }
}
```

L'ajout des objets `Class` à `NSMutableArray` permet d'associer n'importe quel nombre d'objets de commande de même type, VCO, BCO ou autres, à la même commande et de les exécuter ensuite individuellement dans l'ordre où les messages `mapCommandTo*` ont été envoyés. Ainsi, plusieurs VCO peuvent être exécutés séquentiellement.

Openmirrors.com

Par exemple, un VCO peut afficher un `UIView` suivi d'un autre qui modifie l'opacité d'un autre `UIView`, pour terminer par la journalisation d'un message. Pour cela, il suffit d'envoyer trois messages `mapCommandToVCO` avec la même commande, mais avec trois noms d'objets de commande différents.

Vous trouverez d'autres exemples de BCO et de VCO dans l'application DeviceCatalog. Chacun est activé par des requêtes effectuées depuis la partie JavaScript de l'application.

En résumé

Ce chapitre a montré comment activer plusieurs fonctionnalités très demandées de l'iPhone ou de l'iPod Touch depuis une application JavaScript. L'utilisation de la localisation GPS, des valeurs de l'accéléromètre, du vibreur du téléphone et du lecteur de sons ou de fichiers audio permet d'enrichir votre application.

En étudiant le code de l'application DeviceCatalog et en connaissant Objective-C, vous devez être en mesure d'ajouter d'autres fonctionnalités, comme le scan du réseau Bonjour à la recherche des appareils du voisinage, l'ajout, la suppression et l'obtention des contacts à partir de l'application Contacts, ou l'exploitation d'autres comportements intégrés disponibles aux applications Objective-C.

En se fondant sur l'approche décrite dans ce chapitre, votre application JavaScript a les moyens de réaliser quasiment tout ce qu'une application Objective-C peut effectuer. Le Chapitre 6 en présentera un exemple, qui consistera à embarquer des cartes Google dans une application en conservant l'aspect de l'application de cartographie d'Apple.

Openmirrors.com



5

GPS, accéléromètre et autres fonctions natives avec PhoneGap

La bibliothèque PhoneGap constitue une alternative au framework QuickConnect, qui a fait l'objet du Chapitre 4. Bien qu'elle n'offre pas toutes les possibilités de QuickConnect, elle est souvent utilisée pour accéder aux informations de l'appareil et à ses fonctions. La première section de ce chapitre explique comment accéder aux fonctionnalités natives de l'iPhone en utilisant l'API JavaScript de PhoneGap. La seconde présente le code JavaScript et Objective-C qui se cache derrière cette API.

Section 1 : activation de l'appareil en JavaScript

Au moment de l'écriture de ces lignes, le développement de PhoneGap en est toujours à ses débuts. Il gère donc un nombre limité de fonctionnalités natives. Cette section présente uniquement les comportements qui sont pris en charge du côté JavaScript et du côté

Objective-C de l'application. Certaines parties du code de la bibliothèque ne sont pas encore terminées ou pleinement opérationnelles. Les comportements correspondants ne seront donc pas examinés.

Le Tableau 5.1 recense les méthodes, les fonctions et les attributs qui font partie de l'API opérationnelle. Contrairement à QuickConnectiPhone, elles ne se trouvent pas dans un framework plus vaste et doivent donc être invoquées directement. Puisque les contrôles effectués par la version actuelle sont minimaux, vous devez vérifier que les paramètres passés aux fonctions sont valides et non `null`.

Nous l'avons expliqué au Chapitre 4, QuickConnectiPhone inclut les fichiers HTML, CSS et JavaScript dans l'application installée sur l'appareil. Ce n'est pas le cas de PhoneGap[1]. À la place, une application Objective-C générique de démarrage, à laquelle est donné le nom de votre application, charge temporairement ces fichiers à partir d'un serveur web sur lequel vous les avez préalablement publiés (voir Figure 5.1). Pour exécuter votre application, l'appareil doit donc être connecté au réseau. Par conséquent, vous ne devez pas supposer que votre application fonctionnera correctement si l'utilisateur se trouve dans un avion où le réseau est inaccessible.

Démarrage d'une application PhoneGap

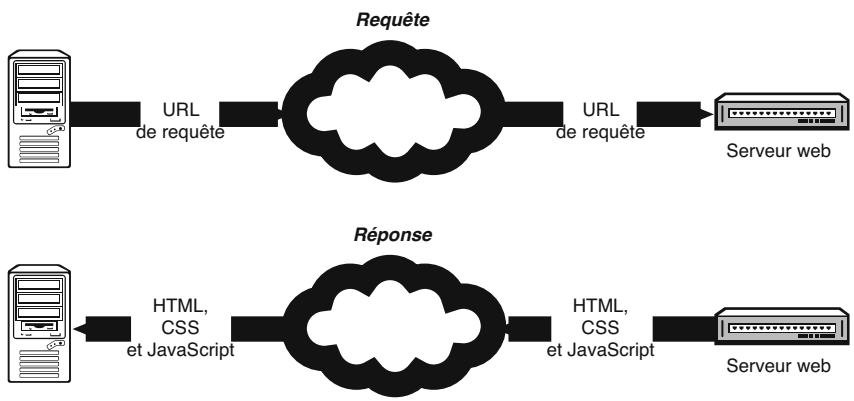


Figure 5.1
Avec PhoneGap, l'obtention des fichiers HTML, CSS et JavaScript d'une application se fait par un échange requête-réponse.

1. N.d.T. : rappelons qu'à partir de la version 0.7.3 PhoneGap enregistre les fichiers JavaScript, HTML et CSS sur l'appareil, non plus sur un serveur web.

Openmirrors.com

Lorsqu'une application PhoneGap débute le chargement de ses composants web à partir du serveur, un appel est effectué depuis l'application Objective-C générique vers le code JavaScript. Cet appel fixe les valeurs de plusieurs variables globales qui décrivent l'appareil sur lequel l'application s'exécute. La méthode `Device.init` de l'API JavaScript, généralement appelée depuis une fonction définie comme gestionnaire des événements `onload`, collecte les variables globales définies par la partie Objective-C de l'application et les enregistre dans des attributs de l'objet `Device` de PhoneGap. Le code Objective-C qui définit ces variables sera présenté à la Section 2.

```
init: function(model, version) {
    ...
    Device.available = __gap;
    Device.model = __gap_device_model;
    Device.version = __gap_device_version;
    Device.gapVersion = __gap_version;
    Device.uuid = __gap_device_uniqueid;
    ...
}
```

Le code précédent concerne les iPhone et iPod Touch. Chaque nom de variable globale débute par __ (deux soulignés) et a une valeur unique. La variable `__gap` est utilisée comme indicateur qui précise si le code JavaScript s'exécute depuis une application PhoneGap. Au moment de l'écriture de ces lignes, cette variable est indispensable car les fichiers HTML, CSS et JavaScript sont non pas inclus dans une application compilée et installée, mais téléchargés *via* Internet par l'enveloppe PhoneGap. Si ces fichiers de l'application sont chargés depuis un navigateur web à la place d'une application PhoneGap, la variable `__gap` vaut `null` et peut être utilisée pour exécuter du code conditionnel.

La variable `__gap_device_model` contient une chaîne de caractères qui décrit l'appareil sur lequel s'exécute l'application, comme iPhone, iPod Touch ou iPhone Simulator. Elle inclut également la version du système d'exploitation de l'appareil, qui est enregistrée dans la variable globale `__gap_device_version`. L'identifiant global unique de l'appareil se trouve dans la variable `__gap_device_uniqueid`, tandis que la version de PhoneGap est placée dans `__gap_version`. Chacune de ces variables est disponible dans les attributs publics de `Device` ou dans les variables globales. Le Tableau 5.1 décrit chaque fonction JavaScript qui vous permet d'accéder aux fonctionnalités de l'appareil.

L'exemple PGDeviceCatalog comprend un bouton `Vibrate` qui, lorsque l'utilisateur clique dessus, fait vibrer le téléphone.

Tableau 5.1 : API JavaScript de PhoneGap

Élément	*Paramètres*	*Comportement*
`gotAcceleration()`	`x` – la valeur de l'accélération selon l'axe X. `y` – la valeur de l'accélération selon l'axe Y. `z` – la valeur de l'accélération selon l'axe Z.	Cette fonction n'est pas prédéfinie. Pour obtenir les informations de l'accéléromètre, vous devez créer cette fonction quelque part dans votre application. Le code Objective-C peut ensuite l'invoquer en lui passant les valeurs en paramètres.
`Device.init()`	Aucun.	Collecte les informations concernant l'appareil à partir des variables globales définies au démarrage de l'application. Ces informations sont placées dans l'objet `Device` et incluent le type de l'appareil (iPhone/iPodTouch), la version de son système d'exploitation, son identifiant unique, ainsi que la version de PhoneGap.
`Device.vibrate()`	Aucun.	Déclenche le vibreur du téléphone pendant la durée standard.
`Device.sound()`	`clip` – une chaîne de caractères qui précise le nom et le type du fichier audio à jouer, par exemple "tweet.wav".	Le fichier audio de nom indiqué doit se trouver dans les ressources de l'application ou une erreur d'exécution sera générée.
`Device.Location.init()`	Aucun.	Déclenche l'envoi des informations de localisation déjà présentes dans la partie Objective-C au code JavaScript en vue de leur traitement et/ou de leur affichage.
`Device.Location.callback`		Si une fonction de rappel est affectée à cet attribut de l'objet `Location`, elle est invoquée lorsque les informations GPS sont disponibles.
`Device.Location.wait()`	`func` – la fonction de rappel exécutée lorsque les informations GPS sont disponibles.	Il s'agit d'une alternative à la fonction `Device.Location.init()`.
`Device.exec()`	`command` – une chaîne de commande passée à la partie Objective-C de l'application en vue de son traitement, par exemple "vibrate", "sound" ou "getLoc".	Cette fonction prend en charge toutes les communications avec la partie Objective-C de l'application. Chaque commande envoyée déclenche un comportement natif différent de l'appareil. Cette méthode doit être invoquée depuis JavaScript si vous créez un code Objective-C personnalisé que vous souhaitez exécuter.

Openmirrors.com

Le gestionnaire d'événements `onclick` du bouton est la fonction nommée `vibrateDevice`. Elle appelle la méthode `Device.vibrate` qui déclenche le vibreur.

```
function vibrateDevice(event)
{
    Device.vibrate();
}
```

La méthode `Device.vibrate` suivante est une façade pour la méthode `exec` de l'objet `Device`. Elle invoque `Device.exec` en lui passant la commande `vibrate`. Toutes les méthodes PhoneGap associées aux fonctionnalités de l'appareil sont en réalité des façades pour `Device.exec`.

```
vibrate: function() {
    return Device.exec("vibrate")
}
```

La fonction `Device.exec` utilisée dans le code précédent est écrite en JavaScript (voir ci-après). À l'instar de QuickConnectiPhone, PhoneGap construit l'URL et crée un message composé de l'URL et de la commande, comme `vibrate`, qui est envoyé à un composant Objective-C faisant partie du framework PhoneGap sous-jacent. Ce composant arrête le chargement de la nouvelle page et évalue les commandes envoyées. Pour de plus amples informations concernant cette procédure, consultez la Section 2.

```
exec: function(command) {
    if (Device.available) {
        try {
            document.location = "gap:" + command;
        } catch(e) {
            console.log("La commande '" + command +
                    "' n'a pas été exécuté en raison de l'exception : " + e);
            alert("Erreur d'exécution de la commande '" + command + "'.")
        }
    }
}
```

Dans la partie `catch` du code précédent, deux mécanismes sont utilisés pour indiquer un problème à l'utilisateur. Le premier, un appel à `console.log`, écrit un message sur la console de Dashcode. Il fonctionne uniquement lorsque l'application s'exécute dans Dashcode, non sur l'appareil.

Le second mécanisme se fonde sur une boîte d'alerte. Puisque PhoneGap a mis en œuvre les boîtes d'alerte en Objective-C, cette solution fonctionne sur l'appareil, mais, selon les directives d'Apple, elle ne devrait jamais être utilisée dans les applications pour l'iPhone.

Pour de plus amples informations sur la conception de l'interface utilisateur des applications pour l'iPhone, consultez le Chapitre 3.

Les méthodes `playSound` et `vibrate` sont toutes deux unidirectionnelles, avec une communication depuis JavaScript vers Objective-C, sans données retournées. La méthode `Device.Location.init` est bidirectionnelle et des données fournies par la partie Objective-C de PhoneGap sont donc attendues. Comme le montre l'exemple suivant, cette méthode `init` est également une façade pour la méthode `Device.exec`. Dans son cas, la commande passée se nomme `getloc`.

```
init: function() {
    ...
    Device.exec("getloc");
    ...
}
```

La Figure 5.2 illustre l'exécution d'une application PhoneGap qui a demandé des informations de localisation GPS. Cette requête se fait également au travers d'une façade qui invoque `Device.exec`.



Figure 5.2
L'application PGDevice-Catalog affiche des informations concernant la localisation GPS et l'appareil.

Dans l'application PGDeviceCatalog, la méthode `Device.Location.init` est appelée à partir de la fonction `getGPS` constituée de quatre lignes. La troisième ligne invoque `init` et signale à la partie Objective-C que ses données GPS enregistrées sont requises.

Openmirrors.com

```
function getGPS(event){
    Device.Location.callback = updateLocation;
    Device.Location.init();
}
```

La deuxième ligne de `getGPS` informe l'objet `Device.Location` que la fonction `updateLocation`, définie dans *main.js*, doit être appelée lorsque les données GPS ont été obtenues. Dans PhoneGap, ce processus est beaucoup plus rapide que dans QuickConnectiPhone, car la bibliothèque Objective-C de PhoneGap active le matériel GPS de l'appareil dès que l'application est démarrée et le désactive lorsqu'elle est quittée.

L'activation du matériel GPS pendant tout le temps d'exécution des applications PhoneGap, même si les données ne sont pas employées, utilise une quantité importante d'énergie de la batterie. Cette consommation est tellement importante qu'Apple précise que laisser le matériel GPS activé pendant la durée d'exécution d'une application revient à être "un mauvais voisin". Avec cette approche, la batterie de l'appareil risque de ne plus avoir suffisamment de puissance pour les appels téléphoniques et l'exécution d'autres applications. C'est pourquoi les applications PhoneGap doivent être conçues de manière à s'exécuter pendant de courtes périodes de temps, même si elles n'exploitent pas les informations GPS. Puisque QuickConnectiPhone démarre le matériel GPS lorsque les informations de localisation sont demandées et l'arrête dès qu'elles ont été obtenues, les applications développées avec ce framework peuvent être conçues pour des durées d'utilisation beaucoup plus longues.

Pour obtenir les données de l'accéléromètre, une fonction nommée `gotAcceleration(x, y, z)` doit être implémentée quelque part dans l'application. Dans l'exemple de PGDeviceCatalog, elle se trouve dans le fichier *main.js*.

```
function gotAcceleration(x, y, z){
    document.getElementById('accelDisplay').innerHTML =
            'X: '+x+'<br/>Y: '+y+' <br/>Z: '+z;
}
```

Cette version de la fonction `gotAcceleration` affiche uniquement les valeurs de l'accéléromètre. Dans votre implémentation, vous pouvez les utiliser de différentes manières, par exemple comme un filtre passe-bas ou pour modifier l'interface utilisateur.

Puisque le simulateur de l'iPhone ne donne pas accès à l'accéléromètre, vous devez exécuter l'application sur un appareil réel pour voir l'affichage de ces informations. Dans ce cas, la fonction `gotAcceleration` est appelée chaque fois que l'accéléromètre détecte un mouvement.

La bibliothèque PhoneGap permet également de jouer un fichier audio depuis du code JavaScript. La fonction `playTweetSound` constitue un exemple de cette fonctionnalité.

Elle appelle la méthode `Device.playSound`, qui, à l'instar de la méthode `Device.vibrate`, est une façade pour `Device.exec`.

```
function playTweetSound(event) {
    Device.playSound('bird.mp3');
}
```

La méthode `playSound` requiert en argument le nom complet du fichier audio à jouer. Ce fichier doit se trouver dans le groupe `Resources` du projet Xcode de l'application. Il ne peut pas être placé sur le serveur web avec les fichiers HTML, JavaScript et CSS. S'il n'est pas inclus aux ressources de l'application, il n'est pas lu.

Dans notre exemple, *bird.mp3* est passé à la méthode `playSound` de l'objet `Device` et il est utilisé en paramètre de la commande `sound` (voir ci-après). À ce stade du développement de PhoneGap, `playSound` est la seule méthode opérationnelle qui gère une commande avec des paramètres. Il semble que cela changera lorsque l'équipe de développements de PhoneGap ajoutera d'autres fonctionnalités. Pour de plus amples informations concernant la feuille de route de PhoneGap, consultez l'Annexe C.

```
playSound: function(clip) {
    return Device.exec('sound:' + clip);
}
```

Cette section a montré comment activer les fonctionnalités natives de l'appareil au travers de PhoneGap. La Section 2 décrit la partie Objective-C de la bibliothèque PhoneGap qui prend en charge cette possibilité.

Section 2 : activation de l'appareil en Objective-C

Si vous n'êtes pas familier du langage Objective-C et de son utilisation dans le développement d'applications pour l'iPhone, consultez le livre d'Erica Sadun, *The iPhone Developer's Cookbook*. Si vous souhaitez simplement utiliser la bibliothèque PhoneGap pour écrire des applications JavaScript pour l'iPhone, vous n'êtes pas obligé de lire cette section.

Après que les fichiers HTML, CSS et JavaScript de l'application ont été obtenus à partir du serveur web, l'API déclenche un événement intercepté et traité par la méthode `webViewDidStartLoad` de l'objet `GlassAppDelegate`. Son rôle est d'initialiser un objet PhoneGap `Device`.

```
// Au chargement de l'application web, lui passer
// les informations concernant l'appareil.
```

Openmirrors.com

```
- (void)webViewDidStartLoad:(UIWebView *)theWebView {
    [theWebView stringByEvaluatingJavaScriptFromString:
            [[Device alloc] init]];
}
```

La méthode `init` de l'objet `Device` crée une chaîne de caractères constituée d'une suite d'appels JavaScript. Chacun de ces appels fixe la valeur d'une variable globale, notamment le modèle de l'appareil, iPhone ou iPod Touch, son identifiant unique et la version de son système d'exploitation.

La chaîne créée est retournée par la méthode `init` afin qu'elle puisse être utilisée par la méthode `stringByEvaluatingJavaScriptFromString` de l'objet `UIWebView`. L'évaluation de la chaîne par `UIWebView` permet de fixer les valeurs de variables JavaScript globales, comme `__gap_device_uniqueid`. Nous verrons plus loin dans ce chapitre comment ces variables globales sont assemblées dans un objet JavaScript.

```
@implementation Device

- (NSString *)init{
    jsCallBack = nil;
    myCurrentDevice = [UIDevice currentDevice];
    return jsCallBack = [[NSString alloc] initWithFormat:@"\
            __gap = true; \
            __gap_version='0.1'; \
            __gap_device_model='%s'; \
            __gap_device_version='%s';\
            __gap_device_uniqueid='%s';",
            [[myCurrentDevice model] UTF8String],
            [[myCurrentDevice systemVersion] UTF8String],
            [[myCurrentDevice uniqueIdentifier] UTF8String]
            ];
}

- (void)dealloc {
    [jsCallBack release];
    [myCurrentDevice release];
    [super dealloc];
}

@end
```

Ces variables globales sont fixées sans effectuer une requête. D'autres fonctionnalités vous obligent à écrire du code qui déclenche le comportement souhaité.

L'un des comportements les plus faciles à implémenter en Objective-C est l'activation du vibreur de l'iPhone. Il suffit en effet d'une seule ligne de code lorsque le framework AudioToolbox est inclus aux ressources du projet :

```
AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
```

Comment faire en sorte que la fonction `AudioServicesPlaySystemSound` soit appelée lorsque le code change l'attribut `location` de `UIWebView` ?

La classe `GlassAppDelegate` implémente la méthode `webView:shouldStartLoadWithRequest:navigationType`. Puisque `GlassAppDelegate` est le délégué du `UIWebView` embarqué (voir ligne 36 du fichier *GlassAppDelegate.m*), nommé `aWebView`, cette méthode est invoquée chaque fois que le `UIWebView` embarqué voit son attribut `location` modifié.

```
webView.delegate = self;
```

Le comportement de base de la fonction `webView:shouldStartLoadWithRequest:navigationType` est simple. Sa conception vous permet d'écrire du code qui décide si la nouvelle page demandée doit être réellement chargée. La bibliothèque PhoneGap se fonde sur cette possibilité de décision pour interdire les requêtes de commande effectuées par les appels JavaScript décrits à la Section 1 et pour exécuter un autre code Objective-C.

La méthode `shouldStartLoadWithRequest` prend plusieurs paramètres :

- `curWebView`. Le `UIWebView` qui contient l'application JavaScript.
- `request`. Un `NSURLRequest` qui contient, entre autres, la nouvelle URL.
- `navigationType`. Un `UIWebViewNavigationType` qui peut être utilisé pour déterminer si la requête est le résultat d'un clic sur un lien ou si elle a été générée suite à une autre action.

```
-(BOOL)webView:(UIWebView *)curWebView
    shouldStartLoadWithRequest:(NSURLRequest *)request
    navigationType:(UIWebViewNavigationType)navigationType
```

L'URL composée par la méthode JavaScript `Device.exec` pour faire vibrer le téléphone, `gap:vibrate`, est contenue dans l'objet `request` et peut être facilement retrouvée sous forme d'une chaîne de caractères en envoyant le message `URL` à cet objet. Ce message retourne un objet de type `NSURL`, auquel est ensuite passé le message `absoluteString`. Nous obtenons ainsi un pointeur `NSString` qui représente l'URL.

```
NSString *url = [[request URL] absoluteString];
NSArray *urlArray = [url componentsSeparatedByString:@"?"];
```

Openmirrors.com

Dans PhoneGap, on détermine si l'URL demandée est relative à celle de l'application indiquée dans le fichier *url.txt*. Dans la négative, le navigateur Safari est lancé pour afficher la page désignée par l'URL. Dans ce cas, l'application se termine, car l'iPhone n'autorise l'exécution que d'une seule application à la fois.

Le code suivant obtient l'hôte demandé, indiqué dans le paramètre `url`, et l'hôte de l'application indiqué dans `appURL`. Pour cela, le message `host` est envoyé aux deux objets `NSURL`.

Les lignes 3 et 4 envoient le message `rangeOfString` à la variable `urlHost`. Cela équivaut à l'appel de la méthode `indexOf` sur un objet JavaScript `String`. Le code suivant détermine si la valeur de l'hôte de l'application est présent dans l'URL de requête.

```
 1 NSString* urlHost = [url host];
 2 NSString* appHost = [appURL host];
 3 NSRange range = [urlHost rangeOfString:appHost
 4         options:NSCaseInsensitiveSearch];
 5 if (range.location == NSNotFound)
 6     [[UIApplication sharedApplication] openURL:url];
 7
 8 NSString * jsCallBack = nil;
 9 NSArray * parts = [urlString
10         componentsSeparatedByString:@":"];
```

La ligne 5 examine le résultat de la méthode `rangeOfString` pour déterminer si `appHost` a été trouvé dans `urlHost`. Dans la négative, le message `openURL` est envoyé à votre application. Chaque fois que ce message `openURL` est envoyé, votre application se termine et l'application appropriée est démarrée. Si une URL commençant par `map:` est demandée, l'application de cartographie de l'iPhone est lancée avec l'URL indiquée. Les autres types possibles sont notamment `http`, qui lance Safari, `tel`, qui lance l'outil de numérotation téléphonique, `mailto`, qui lance l'application de messagerie, et les URL `youtube.com`, qui lancent l'application YouTube. Dans tous les cas, votre application se termine.

Les lignes 9 et 10 décomposent l'URL de manière à obtenir les composantes de la commande et placent chacune d'elles dans un tableau nommé `parts`. Celui-ci est ensuite évalué pour déterminer la commande envoyée et ses paramètres.

PhoneGap utilise une instruction conditionnelle `if-then-else` pour évaluer le tableau `parts`. Chaque commande active une condition différente. Dans la condition de la commande `vibrate`, un objet `Vibrate` est instancié et le message `vibrate` lui est passé.

```
else if([(NSString *)[parts objectAtIndex:1]
        isEqualToString:@"vibrate"]){
    Vibrate *vibration = [[Vibrate alloc] init];
    [vibration vibrate];
```

```
    [vibration release];
    NSLog(@"vibreur déclenché");
}
```

C'est dans la méthode `vibrate` de l'objet `Vibrate` (fichier *Vibrate.m*) que l'appel `AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);` est effectué.

Le traitement des requêtes concernant les informations GPS est différent. Cette commande est gérée directement au lieu d'être passée à un objet qui la prendra en charge. Les lignes 5 à 7 créent une chaîne qui contient un appel JavaScript à la fonction `gotLocation(lat, lon)` définie dans le fichier *phonegap.js*, où `lat` et `lon` sont remplacés par la latitude et la longitude courantes ; ces informations sont déterminées en permanence depuis le démarrage de l'application.

```
 1 if([(NSString *)[parts objectAtIndex:1]
 2         isEqualToString:@"getloc"]){
 3     NSLog(@"location request!");
 4
 5     jsCallBack = [[NSString alloc]
 6             initWithFormat:@"gotLocation('%f','%f');"
 7             , lat, lon];
 8     NSLog(@"callback: %@",jsCallBack);
 9     [theWebView
10             stringByEvaluatingJavaScriptFromString:
11             jsCallBack];
12
13     [jsCallBack release];
14 }
```

Pour déclencher l'exécution de la chaîne JavaScript `jsCallBack`, le message `stringByEvaluatingJavaScriptFromString`, accompagné de la chaîne JavaScript en paramètre, doit être envoyé à l'objet `UIWebView` passé dans le paramètre `theWebView` à la fonction `shouldStartLoadWithRequest`. À ce stade, la partie Objective-C de la bibliothèque a terminé son travail.

La fonction JavaScript `gotLocation` appelle à présent la méthode `set` de l'objet `Device.Location` :

```
function gotLocation(lat, lon) {
    return Device.Location.set(lat, lon)
}
```

La méthode `Device.Location.set` enregistre la latitude et la longitude dans l'objet `Device.Location` et appelle ensuite la fonction de rappel `callback`, si elle existe, comme nous l'avons expliqué à la Section 1. Notez qu'après l'invocation de la fonction de rappel

Openmirrors.com

la ligne 6 fixe l'attribut `callback` à `null`. Autrement dit, chaque fois que vous demandez des informations de localisation, vous devez repréciser la méthode de rappel.

```
1  set: function(lat, lon) {
2      Device.Location.lat = lat;
3      Device.Location.lon = lon;
4      if(Device.Location.callback != null) {
5          Device.Location.callback(lat, lon)
6          Device.Location.callback = null;
7      }
8  }
```

Si vous ne définissez pas une méthode `callback` chaque fois que vous demandez des informations de localisation, aucune fonction de rappel n'est invoquée.

Le traitement de la commande `sound` est comparable à celui de la commande `getloc`. Toutefois, puisque la partie JavaScript de l'application n'attend aucune donnée, le message `stringByEvaluatingJavaScriptFromString` n'est pas envoyé à `UIWebView`. À la place, un objet `Sound` est créé comme dans le cas de la commande `vibrate`.

```
 1 else if ([(NSString *)[parts objectAtIndex:1]
 2         isEqualToString:@"sound"]) {
 3     NSLog(@"playing sound");
 4     NSLog([parts objectAtIndex:2]);
 5     NSString *ef = (NSString *)[parts objectAtIndex:2];
 6     NSArray *soundFile = [ef componentsSeparatedByString:@"."];
 7
 8     NSString *file = (NSString *)[soundFile objectAtIndex:0];
 9     NSString *ext = (NSString *)[soundFile objectAtIndex:1];
10     NSLog(@"about to allocate %@, %@",file, ext);
11     sound = [[Sound alloc] initWithContentsOfFile:
12             [mainBundle pathForResource:file ofType:ext]];
13     NSLog(@"sound allocated");
14     [sound play];
15 }
```

Cet objet `Sound`, défini dans *Sound.m*, est initialisé avec le chemin du fichier audio. Nous l'avons indiqué à la Section 1, ce fichier doit résider dans le groupe `Resources` du projet Xcode. Par conséquent, nous pouvons envoyer à l'objet `mainBundle`, qui représente l'application installée, le message `pathForResource:ofType` afin d'obtenir le chemin complet du fichier audio sur l'appareil (lignes 11 et 12). La ligne 14 envoie le message `play` à l'objet `Sound` pour que le fichier soit joué.

La méthode `initWithContentsOfFile` de l'objet `Sound` montre comment convertir des fichiers audio, par exemple des fichiers mp3, en sons système. Pour cela, les fichiers audio

doivent être très courts. En réalité, Apple suggère qu'ils durent moins de 5 secondes. Chaque son système est créé à partir de l'URL de son emplacement. La ligne 1 du code suivant illustre cette procédure avec une chaîne de chemin quelconque. La ligne 3 crée l'URL.

```
1  - (id) initWithContentsOfFile:(NSString *)path
2  {
       ...
3      NSURL *filePath = [NSURL fileURLWithPath:path isDirectory:NO];
4      AudioServicesCreateSystemSoundID((CFURLRef)filePath, &soundID);
       ...
5  }
```

La ligne 4 convertit le fichier audio en son système. Les sons système diffèrent des fichiers audio standard car ils sont interprétés et enregistrés dans le système d'exploitation lui-même. Pour les jouer, aucun lecteur multimédia n'est nécessaire. L'invocation de la fonction `AudioServicesCreateSystemSoundID` suffit.

Cette fonction prend deux arguments. Le premier correspond à l'URL du fichier audio, le second est un pointeur sur un `SystemSoundID`. Dans l'exemple, ce `SystemSoundID` est l'attribut `soundID` de l'objet `Sound`, qui est utilisé ensuite pour jouer le son dans la méthode `play` de l'objet `Sound`.

Comme le montre le code suivant, cette méthode `play` occupe une seule ligne de code. Un appel à la fonction `AudioServicesPlaySystemSound`, en lui passant un `SystemSoundID`, suffit pour que l'utilisateur entende le son.

```
- (void) play {
    AudioServicesPlaySystemSound(soundID);
}
```

Quel que soit le fichier audio choisi comme son système, voici les étapes de sa création et de son utilisation :

1. Obtenir une URL qui désigne l'emplacement du fichier audio sur l'appareil.
2. Générer le son système et enregistrer son identifiant.
3. Jouer le son système.

Avec PhoneGap, le son système est généré chaque fois que vous demandez sa lecture, ce qui n'est pas très efficace. Il est préférable de créer le son une seule fois et de le jouer ensuite autant de fois que vous voulez.

Nous l'avons mentionné précédemment, PhoneGap active le matériel GPS au démarrage de l'application. Cela se passe dans les trois premières lignes de la méthode

Openmirrors.com

`applicationDidFinishLaunching` de la classe `GlassAppDelegate`. Le code suivant montre que ces trois lignes initialisent un `CLLocationManager`, l'enregistre dans l'attribut `locationManager` de la classe `GlassAppDelegate` et lui demande de commencer l'actualisation des informations GPS.

```
locationManager = [[CLLocationManager alloc] init];
locationManager.delegate = self;
[locationManager startUpdatingLocation];
```

La classe `CLLocationManager` sert d'enveloppe au matériel GPS et Wi-Fi qui détermine la localisation courante de l'appareil. Elle se fonde sur la puce GPS et les points d'accès Wi-Fi ouverts pour déterminer la latitude et la longitude actuelles.

La deuxième ligne du code précédent demande à l'objet `locationManager` d'appeler la méthode `didUpdateToLocation` de `GlassAppDelegate` chaque fois qu'un changement de localisation est détecté. Pour cela, elle fixe le délégué de l'objet `locationManager` au `GlassAppDelegate` courant représenté par le mot clé `self`. Pour de plus amples informations concernant les délégués, consultez les Chapitres 2 et 4 du livre *The iPhone Developer's Cookbook: Building Applications with the iPhone SDK*, d'Erica Sadun.

La méthode déléguée appelée à chaque changement de localisation se nomme `didUpdateToLocation` ; elle est définie dans le fichier *GlassAppDelegate.m*. Comme le montre le code suivant, elle efface toute localisation déjà enregistrée, pour la remplacer par la localisation actuelle passée à la méthode dans le paramètre `newLocation`. Nous l'avons vu précédemment, cette information est utilisée par `shouldStartLoadWithRequest` dans sa condition `getloc`.

```
-(void)locationManager:(CLLocationManager *)manager
    // Note de l'auteur.
    // Il y a un bogue potentiel ici.
    // Si newLocation == lastKnown alors
    // l'objet newLocation est libéré
    // [newLocation retain] doit être appelé
    // avant [lastKnownLoation release]
    // Le code présenté est celui fourni
    // par PhoneGap.
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    [lastKnownLocation release];
    lastKnownLocation = newLocation;
    [lastKnownLocation retain];
}
```

L'activation de l'accéléromètre est gérée de manière semblable à celle du GPS. Les trois lignes de code suivantes demandent à l'accéléromètre d'enregistrer ses données quarante fois par seconde et fixent ensuite le délégué à l'objet `GlassAppDelegate` courant, comme pour le gestionnaire de localisation.

```
[[UIAccelerometer sharedAccelerometer] setUpdateInterval:1.0/40.0];
[[UIAccelerometer sharedAccelerometer] setDelegate:self];
```

Dans ce cas, la méthode appelée est non pas `didUpdateToLocation` mais `didAccelerate`.

Le code suivant montre que la méthode `didAccelerate` ressemble énormément à la méthode `didUpdateToLocation`. Elle obtient les informations de l'accéléromètre, mais, au lieu de les enregistrer localement côté Objective-C de la bibliothèque, elle les envoie à la partie JavaScript de manière comparable au traitement de la commande `gotloc` vue précédemment.

```
-(void) accelerometer:(UIAccelerometer *)accelerometer
        didAccelerate:(UIAcceleration *)acceleration {
    NSString * jsCallBack = nil;
    NSLog(@"accelerating");
    jsCallBack = [[NSString alloc]
            initWithFormat:
            @"gotAcceleration('%f','%f','%f');",
            acceleration.x,
            acceleration.y,
            acceleration.z];
    [webView stringByEvaluatingJavaScriptFromString:jsCallBack];
}
```

Si la méthode `shouldStartLoadWithRequest` reconnaît d'autres commandes, aucune d'elles n'est opérationnelle au moment de l'écriture de ces lignes, et elles n'ont donc pas été présentées. Toutes les commandes décrites dans cette section fonctionnent et sont disponibles dans le programme d'installation du modèle Xcode pour les applications PhoneGap.

En résumé

Ce chapitre a montré comment activer plusieurs fonctionnalités très demandées de l'iPhone ou de l'iPod Touch depuis une application JavaScript en utilisant la bibliothèque PhoneGap. Grâce aux fonctionnalités natives de ces appareils, comme la localisation GPS, l'accéléromètre, le vibreur et les sons, vous pouvez enrichir vos applications.

Openmirrors.com

En étudiant les exemples inclus avec PGDeviceCatalog et en connaissant Objective-C, vous devez être en mesure d'ajouter des fonctionnalités supplémentaires, comme le scan du réseau Bonjour à la recherche des appareils du voisinage, l'ajout, la suppression et l'obtention des contacts à partir de l'application Contacts, ou l'exploitation d'autres comportements natifs disponibles aux applications Objective-C.

En se fondant sur l'approche décrite dans ce chapitre, votre application JavaScript dispose pratiquement des mêmes possibilités qu'une application Objective-C.

Openmirrors.com



6

Cartes Google

De nombreuses applications pour l'iPhone utilisent des cartes géographiques. Pour afficher ces cartes, il existe différentes méthodes, comme fermer l'application en cours et ouvrir l'application de cartographie fournie par Apple, ou utiliser une carte Google. Ces deux approches présentent certaines limites. Ce chapitre explique comment créer une carte Google et l'utiliser à la manière de l'application de cartographie de l'iPhone sans fermer l'application en cours.

Section 1 : afficher une carte dans une application JavaScript QuickConnect

Dans les applications hybrides pour l'iPhone, les programmeurs peuvent utiliser des cartes de différentes manières. La solution la plus simple consiste à ajouter un lien qui débute par **http://maps.google.com** et contient les informations géographiques souhaitées. Lorsque l'utilisateur ouvre un tel lien, l'application en cours se termine et l'application de cartographie standard est démarrée pour afficher la carte demandée.

Cette approche simple est facile et rapide à mettre en œuvre. Toutefois, l'application en cours est terminée, ce qui constitue généralement une mauvaise conception logicielle.

L'application donne un sentiment d'inachevé, et les utilisateurs ont en réalité besoin d'une approche plus intégrée.

Par ailleurs, même si Google peut répondre à des requêtes, comme "pizza", et placer de nombreuses punaises, il est actuellement impossible de mettre plusieurs punaises sur des emplacements que vous définissez. Par exemple, vous pourriez souhaiter placer des punaises en différents lieux d'une ville. Si ces lieux ne correspondent pas à des points d'intérêt que l'on peut rechercher, comme "pizza", vous ne pourrez pas placer les punaises en ajoutant une description de chaque lieu à l'URL soumise à Google. Cette limitation est gênante lorsque l'on veut définir les lieux punaisés par leur latitude et longitude.

Une autre solution consiste à mettre en œuvre dans l'application un comportement semblable à celui obtenu avec une page web standard. L'API AJAX de Google est alors utilisée pour embarquer une carte dans un `<div>` du contenu HTML affiché. Ensuite, chaque punaise est placée indépendamment en passant par l'API JavaScript de Google.

Si cette approche permet de garder l'utilisateur dans l'application, elle présente également des inconvénients. Tout d'abord, le `UIWebView` qui affiche la carte n'autorise pas le défilement dans les `<div>`. Les événements de toucher et de déplacement sont traités à un niveau inférieur et ne sont pas transmis à la partie JavaScript de Google qui interagit avec la carte embarquée. Autrement dit, vous pouvez afficher la carte, mais il est impossible de la déplacer pour changer la zone affichée.

Par ailleurs, la taille des bulles d'information sur les lieux proposées en standard par Google pose problème. Elles sont dimensionnées pour un affichage dans un navigateur sur un ordinateur. Lorsqu'elles sont affichées sur l'iPhone, elles ont tendance à recouvrir une grande partie de la carte. La plupart des bulles apparaissent généralement hors de l'écran et, en raison du problème de défilement mentionné précédemment, ne sont pas visibles. S'il est possible de limiter la longueur du contenu de ces bulles, il est impossible d'en changer la taille.

La solution idéale serait d'embarquer la carte dans l'application et de proposer l'affichage de plusieurs punaises, comme dans la seconde option, tout en gardant les possibilités de défilement et d'affichage de la première. Le framework QuickConnectiPhone propose un composant qui permet de mettre en œuvre cette solution par un seul appel de JavaScript.

Le projet Xcode MapExample montre comment procéder. L'écran principal de cet exemple comprend un seul bouton HTML (voir Figure 6.1). Le gestionnaire `onclick` de ce bouton est la fonction `showTheMap` définie dans le fichier *main.js* et dont le code est donné ci-après. Elle configure quatre lieux : la ville de Rexburg dans l'Idaho, le Wyoming, une contrée plus sauvage et une sandwicherie.

Openmirrors.com



Figure 6.1
L'écran principal de l'application MapExample comprend un bouton HTML.

```
function showTheMap(event)
{
    // Un lieu est défini par une latitude,
    // une longitude et une description.
    var locationsArray = new Array();
    rexburg = new Array();
    rexburg.push(43.82211);
    rexburg.push(-111.76860);
    rexburg.push("Mairie");
    locationsArray.push(rexburg);

    var wyoming = new Array();
    wyoming.push(42.86);
    wyoming.push(-109.45);
    wyoming.push("Place de Wyoming");
    locationsArray.push(wyoming);

    var wilderness = new Array();
    wilderness.push(45.35);
    wilderness.push(-115);
    wilderness.push("Rivière du sans retour ");
    locationsArray.push(wilderness);

    var sandwichShop = new Array();
    sandwichShop.push(42.86);
```

```
        sandwichShop.push(-112.45);
        sandwichShop.push("Sandwicherie");
        locationsArray.push(sandwichShop);

        showMap(event, locationsArray);
    }
```

Chaque lieu défini est un tableau composé de trois éléments : une latitude, une longitude et une courte description à afficher sur chaque punaise placée. Ces lieux sont ajoutés à `locationsArray`. Si l'ordre des informations définissant chaque lieu est figé, le tableau `locationsArray` n'impose aucun ordre.

Dans une application réelle, les informations concernant chaque lieu pourraient provenir d'une base de données, d'un flux RSS ou d'une autre source. Vous pouvez même les obtenir dynamiquement pendant l'exécution de l'application. Si vous connaissez des adresses, utilisez l'API JavaScript de géocodage de Google pour obtenir la latitude et la longitude correspondantes (**http://code.google.com/apis/maps/documentation/services.html#Geocoding_Object**). Toutefois, ces requêtes prennent du temps. Il est préférable de commencer par obtenir les coordonnées des lieux intéressants en lançant une tâche lors de la conception et d'enregistrer les résultats avant de livrer l'application.

Une fois que le tableau JavaScript contenant tous les lieux souhaités est créé, il est passé à la fonction `showMap` du framework, accompagné de l'événement qui a déclenché l'appel à la fonction `showTheMap`. À ce stade, le framework prend le relais et, à l'aide de la fonction `makeCall` décrite au Chapitre 4, demande à la partie Objective-C d'afficher une carte avec des punaises sur chaque lieu (voir Figure 6.2).

Le framework affiche la carte dans un objet Objective-C `MapView`. La classe `MapView`, décrite à la Section 2, permet à l'utilisateur d'employer le toucher et le balayement pour contrôler la carte à la façon de l'application de cartographie d'Apple. Par un double-toucher sur un lieu de la carte, l'application centre l'affichage sur ce lieu et réalise un zoom avant. L'utilisateur peut également faire un double-toucher sur une punaise pour centrer la carte et effectuer un zoom sur le lieu correspondant.

Lorsque l'utilisateur touche simplement une punaise, une courte description est affichée dans une petite boîte noire (voir Figure 6.3). S'il touche et fait glisser une punaise, il la repositionne sur un nouveau lieu de la carte.

Lorsque l'utilisateur n'a plus besoin de la carte, il sélectionne le bouton Done pour faire disparaître le `MapView`. L'affichage de l'application revient dans l'état où il se trouvait au moment où l'application a affiché la carte. Cela résout les problèmes d'utilisation provoqués par la fermeture de l'application, l'ouverture de l'application de cartographie, la fermeture de celle-ci et le redémarrage de l'application initiale.

Openmirrors.com



Figure 6.2
L'application MapExample place une punaise sur chaque lieu.



Figure 6.3
L'application MapExample affiche une courte description.

En invoquant la fonction JavaScript `showMap` du framework, l'application embarque des cartes. La Section 2 détaille la conception et l'utilisation de la classe Objective-C `MapView` du framework, ainsi que d'autres.

Section 2 : implémentation Objective-C du module de cartographie de QuickConnect

Le module de cartographie de QuickConnect est constitué de trois classes :

- `MapView`. L'élément d'affichage principal qui contient des images de la carte.
- `Pin`. Une punaise qui doit être affichée sur un lieu.
- `InfoWindow`. Une classe utilisée pour afficher la courte description associée à une punaise.

La Figure 6.4 illustre les relations entre ces classes. Chaque `MapView` peut avoir plusieurs `Pin`, et chaque `Pin` doit avoir au moins un `MapView`. Il existe également une relation un à un entre les `Pin` et les `InfoWindow`. Autrement dit, pour chaque `Pin`, il doit y avoir au moins un `InfoWindow` et, pour chaque `InfoWindow`, il doit y avoir au moins un `Pin`.

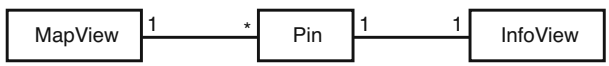


Figure 6.4
Les classes du module de cartographie et leurs relations.

Étant modulaire par nature, une application Objective-C doit interagir directement avec la classe `MapView` et son API, dont la seule méthode se nomme `initWithFrame:andLocations`. Lorsque cette méthode est invoquée, une carte est générée, des punaises sont placées et de courtes descriptions sont disponibles à l'utilisateur lorsqu'il touche une punaise. Par ailleurs, si l'utilisateur réalise un double-toucher sur une punaise ou un lieu de la carte, l'affichage est centré sur ce lieu et un zoom avant est effectué. Le code ci-après montre comment cette API de `MapView` est employée dans le framework QuickConnect.

À l'instar de la localisation GPS, du débogage et des autres requêtes décrites au Chapitre 2, l'appel qui permet d'afficher une carte embarquée passe par un contrôleur frontal et des contrôleurs d'application. De même, la commande `showMap` est associée à `showMapVCO` dans le fichier *QCCommandMappings.m*. La méthode `doCommand` de ce VCO est courte et consiste principalement à placer les informations de latitude, de longitude et de description passées dans un tableau par la requête JavaScript. Pour cela, le premier élément du tableau `parameters` est écarté, car il s'agit du `QuickConnectViewController`

Openmirrors.com

de l'application. La méthode `doCommand`, dont le code est donné ci-après, est extraite du projet Xcode MapExample.

```
+ (id) doCommand:(NSArray*) parameters{
    NSRange aRange = NSMakeRange(1, [parameters count]-1);
    NSArray *locations = [parameters subarrayWithRange:aRange];
    // Dimensionner le MapView à la taille de l'écran.
    MapView *aMapView =
    ➥[[MapView alloc] initWithFrame:[[UIScreen mainScreen]applicationFrame]
            andLocations:locations];
    QuickConnectViewController *theController = [parameters objectAtIndex:0];
    // Ajouter la vue de la carte à la vue principale de l'application.
    [[[theController webView] superview] addSubview:aMapView];
    return nil;
}
```

Puisque le `QuickConnectViewController` possède une référence au `UIWebView` qui affiche et exécute l'application, il permet d'obtenir un pointeur sur sa vue principale. Pour cela, le message `superview` est envoyé au `UIWebView`. Le nouveau `MapView` est ensuite ajouté à la vue principale de l'application en le passant comme paramètre du message `addSubview`. Après l'envoi de ce message, le `MapView` apparaît et occupe l'intégralité de l'écran en masquant le `UIWebView`.

Puisque `MapView` est un module autonome, sa fonctionnalité peut être facilement réutilisée dans de nombreuses applications différentes (voir Chapitre 2 pour les questions de modularité). Il peut même servir dans des applications Mac hybrides, après quelques modifications mineures à la procédure d'affichage de la carte.

Toutes les cartes Google, quel que soit le contenu affiché, sont des pages web. Par conséquent, l'objet `MapView` possède un attribut nommé `webMapView` qui correspond à son propre `UIWebView`. Il est différent de l'instance de `UIWebView` qui affiche l'application.

Comme le montre le code suivant, `webMapView` affiche le fichier *mapView.html* présent dans le groupe `MapView` des ressources, et son délégué est la classe `MapView`. Le groupe `MapView` comprend un `UIView` embarquable et un `WebViewDelegate` qui traite tous les événements pour le `UIWebView`.

```
1  (id)initWithFrame:(CGRect)frame
2          andLocations:(NSArray*)aLocationList {
3      if (self = [super initWithFrame:frame]) {
4          OKToTouch = NO;
5          self.locations = aLocationList;
6          frame.origin.y -= 20;
7          UIWebView *aWebView = [[UIWebView alloc]
8                  initWithFrame:frame];
```

```
 9          self.webMapView = aWebView;
10          [aWebView release];
11
12          aWebView.userInteractionEnabled = NO;
13          // Fixer le délégué de la vue web à la
14          // vue web elle-même.
15          [aWebView setDelegate:self];
16
17          // Déterminer le chemin du fichier mapView.html qui
18          // se trouve dans le répertoire des ressources.
19          NSString *filePathString =
            ➥[[NSBundle mainBundle] pathForResource:@"mapView" ofType:@"html"];
20          MESSAGE(@"%@", filePathString);
21          // Créer l'URL et la requête pour
22          // le fichier mapView.html.
23          NSURL *aURL = [NSURL fileURLWithPath:filePathString];
24          NSURLRequest *aRequest =
25                  [NSURLRequest requestWithURL:aURL];
26
27          // Charger le fichier mapView.html dans la vue web.
28          [aWebView loadRequest:aRequest];
29
30          // Ajouter la vue web à la vue de contenu.
31          [self addSubview:self.webMapView];
32      }
33      return self;
34  }
```

Vous pourriez penser que, pour des raisons de simplicité, la classe `MapView` n'est pas nécessaire, mais ce n'est pas vrai. Puisque le `UIWebView` capture tous les événements de toucher et ne permet pas à ces événements d'être traités par les éléments HTML qu'il affiche, le problème de défilement décrit à la Section 1 apparaît.

Pour résoudre ce problème, la ligne 12 du code précédent désactive le traitement des événements par le `UIWebView` contenu. Cela permet à l'objet `MapView` contenant d'obtenir tous les événements en tant que délégué. Le délégué a ensuite la charge d'indiquer au `UIWebView` que la page qu'il contient doit être décalée.

Pour faire défiler une carte, il faut déterminer que le doigt de l'utilisateur s'est déplacé après le toucher. Pour cela, la méthode standard `touchesMoved:withEvent` de `MapView` doit être implémentée.

```
-(void)touchesMoved:(NSSet *)touches
        withEvent:(UIEvent *)event{
    if(OKToTouch){
        if([touches count] == 1){
```

Openmirrors.com

```
            UITouch *touch = [touches anyObject];
            CGPoint prevLoc = [touch previousLocationInView:self];
            CGPoint curLoc = [touch locationInView:self];
            double touchDeltaX = curLoc.x - prevLoc.x;
            double touchDeltaY = curLoc.y - prevLoc.y;

            NSString *javaScriptCall =
                    [[NSString alloc] initWithFormat:@"scroll(%f, %f)"
                    ,touchDeltaX, touchDeltaY];
            NSString *result = [webMapView
                    stringByEvaluatingJavaScriptFromString javaScriptCall];

            if([result compare:@"true"] == 0){
                int numPins = [pins count];
                for(int i = 0; i < numPins; i++){
                    Pin *aPin = [pins objectAtIndex:i];
                    [aPin moveX:touchDeltaX andY:touchDeltaY];
                    [aPin.info moveX:touchDeltaX andY:touchDeltaY];
                }
            }
        } else if([touches count] == 2){
            // Pincement.
        }
    }
}
```

Cette implémentation de `touchesMoved:withEvent` commence par déterminer le mouvement du toucher. Pour cela, elle calcule la différence entre l'emplacement de l'événement courant et celui de l'événement précédent. Le résultat est passé à la page HTML dans `webMapView` en envoyant le message `stringByEvaluatingJavaSCriptFromString` avec un appel à la fonction JavaScript `scroll`.

Cette fonction, dont le code donné ci-après se trouve dans le fichier *map.js* du groupe `MapView`, utilise l'API JavaScript de cartographie de Google pour recentrer la partie visible de la carte. Elle applique les modifications nécessaires aux valeurs x et y du centre de la carte et indique ensuite à l'objet `map` de se centrer sur cette nouvelle position.

```
function scroll(deltaX, deltaY){
    try{
        var centerPoint = map.fromLatLngToDivPixel(map.getCenter());
        centerPoint.x -= deltaX;
        centerPoint.y -= deltaY;
        var centerLatLng = map.fromDivPixelToLatLng(centerPoint);
        map.setCenter(centerLatLng);
    }
```

```
    catch(error){
        return false;
    }
    return true;
}
```

En cas de succès, cette fonction JavaScript retourne `true` afin que la suite de la méthode `touchesMoved:withEvent` puisse demander à chaque punaise affichée de changer de position.

En raison de la capture désactivée sur le `UIWebView` sous-jacent, pour que le défilement de la carte puisse être réalisé, les punaises standard de Google ne sont pas en mesure de savoir qu'elles ont été touchées. Par conséquent, les bulles de description ne peuvent pas être affichées et masquées pour ces punaises. Il est donc nécessaire de créer notre propre classe `Pin` de gestion des punaises.

Cette classe, qui se trouve dans le groupe `Classes:MapView`, utilise l'image *pinInserted.png*, définie dans `Resources:Images`, pour représenter les punaises à l'écran. Il est très facile de remplacer cette image par le fichier de votre choix.

Un objet de classe `Pin` est initialisé pour chaque emplacement envoyé par l'application depuis le côté JavaScript. Pour cela, la méthode `initWithFrame:andImage:andLocation`, dont le code est donné ci-après, est invoquée. Elle crée un `UIImageView` pour l'image de la punaise, fixe son emplacement et active la capture des événements en fixant son attribut `userInteractionEnabled` à `true`.

```
-(id)initWithFrame:(CGRect)frame
        andImage:(NSString*)anImage
        andLocation:(MapViewLocation)aLocation{

    if (self = [super initWithFrame:frame]) {
        UIImageView *pinImage = [[UIImageView alloc]
                initWithImage:[UIImage imageNamed:anImage]];
        [self addSubview:pinImage];
        [pinImage release];
        location = aLocation;
        self.userInteractionEnabled = YES;
        self.backgroundColor = [UIColor clearColor];
    }
    return self;
}
```

En permettant les interactions avec l'utilisateur sur un objet `Pin`, la punaise peut intercepter et traiter les événements. Lorsqu'une punaise est touchée, la méthode `touchesBegan:withEvent` de l'objet `Pin` est invoquée et modifie l'affichage.

Openmirrors.com

Cette fonction est définie dans le fichier *Pin.m*. Elle vérifie que les événements actuel et précédent ont lieu au même emplacement sur l'écran. Ainsi, le code de cette méthode n'est pas exécuté lorsque l'utilisateur fait glisser son doigt sur l'écran avec un mouvement de type balayement.

Si l'on suppose que le balayement n'a pas lieu, deux cas sont pris en charge par `touchesBegan:withEvent`. Le premier correspond à l'utilisateur qui touche simplement la punaise. Le comportement souhaité consiste alors à l'affichage d'un message simple, qui a été envoyé avec la position par la partie JavaScript de l'application. Pour cela, le mécanisme de réflexion d'Objective-C est utilisé de manière à envoyer un message `singleTouch` à la punaise elle-même.

Le message `performSelector:withObject:afterDelay` est envoyé à la punaise au lieu d'effectuer directement un appel. Cela permet de différencier un toucher simple d'un toucher double. Comme le montre la ligne 13, l'utilisateur dispose de 0,4 seconde pour effectuer un double-toucher. Si ce double-toucher ne se produit pas, le premier toucher est capturé et le second est traité comme un autre toucher.

En revanche, si l'utilisateur effectue un double-toucher sur la punaise dans la limite des 0,4 seconde, un message `cancelPreviousPerformRequestsWithTarget:selector:object` est envoyé de manière à arrêter le passage du message `singleTouch`. Cette solution délai/exécution ou annulation représente la méthode standard pour détecter le nombre de frappes dans les touchers.

Lorsqu'un toucher simple est détecté, le message `singleTouch` est passé et la description courte associée à la punaise est affichée. Dans le cas d'un double-toucher, un zoom est appliqué à la carte et elle est centrée sur l'emplacement de la punaise.

```
 1 -(void)touchesBegan:(NSSet *)touches
 2         withEvent:(UIEvent *)event{
 3     UITouch *touch = [touches anyObject];
 4     CGPoint prevLoc = [touch previousLocationInView:self];
 5     CGPoint curLoc = [touch locationInView:self];
 6
 7     if(CGPointEqualToPoint(prevLoc, curLoc)){
 8         NSUInteger tapCount = [touch tapCount];
 9         switch (tapCount) {
10             case 1:
11                 [self performSelector:
12                         @selector(singleTouch)
13                         withObject:nil afterDelay:.4];
14                 break;
15             case 2:
16                 [NSObject cancelPreviousPerformRequestsWithTarget:self
17                         selector:@selector(singleTouch)
```

```
18                          object:nil];
19                  // Zoom et centrage sur la punaise.
20                  MESSAGE(@"double tap pin %i, %i",
21                          location.x, location.y);
22
23                  double latOffset = (location.y+42)/((MapView*)self.superview)
                    ➥.pixelsPerDegLatitude;
24                 double lngOffset = (location.x+10)/((MapView*)self.superview)
                   ➥.pixelsPerDegLongitude;
25                  double latitude = [[((MapView*)self.superview).northWest
                    ➥objectAtIndex:0] doubleValue] - latOffset;
26                  double longitude = [[((MapView*)self.superview).northWest
                    ➥objectAtIndex:1] doubleValue] + lngOffset;
27                  MESSAGE(@"latitude: %f longitude: %f northWest: %@",
28                          latitude,
29                          longitude,
30                          ((MapView*)self.superview).northWest);
31                  NSString *javaScriptCall = [[NSString alloc] initWithFormat:
                    ➥@"zoomTo(%f, %f)",latitude, longitude];
32                  NSString *mapDescription = [((MapView*)self.superview).
                    ➥webMapView stringByEvaluatingJavaScriptFromString:
                    ➥javaScriptCall];
33
34                 [((MapView*)self.superview) setMapLatLngFrameWithDescription:
                    ➥mapDescription];
35                  [((MapView*)self.superview) updatePinLocations];
36
37                  break;
38              default:
39                  break;
40          }
41      }
42  }
```

En autorisant le zoom et le centrage par un double-toucher sur une punaise ou sur la carte, les possibilités d'utilisation de l'application sont étendues. L'application de cartographie standard ne permet pas ce type de centrage par un double-toucher ; seul le zoom est mis en œuvre. Lorsque l'utilisateur effectue un double-toucher, il le fait généralement suivre d'un balayement pour afficher la région autour de la zone qui l'intéresse. Notre application ne souffre pas de cet inconvénient.

Il est également possible de déplacer les punaises sur la carte en les faisant glisser. Cela se passe dans la méthode `touchesMoved:withEvent`, qui est comparable au défilement de la carte décrit précédemment. La différence réside dans le message `cancelPreviousPerformRequestsWithTarget:selector:object`, qui est de nouveau envoyé de manière à éviter l'affichage de la description en raison de l'appel à la méthode `touchesBegan` avant

Openmirrors.com

celui de `touchesMoved`. Si le message d'annulation n'est pas envoyé, la courte description est affichée et la punaise est déplacée. L'utilisateur risque de ne pas être satisfait de ce comportement.

```
1  -(void)touchesMoved:(NSSet *)touches
2           withEvent:(UIEvent *)event{
3      if([touches count] == 1){
4
5           [NSObject cancelPreviousPerformRequestsWithTarget:self selector:
            ➥@selector(singleTouch) object:nil];
6           moved = YES;
7           UITouch *touch = [touches anyObject];
8           CGPoint prevLoc = [touch previousLocationInView:self];
9           CGPoint curLoc = [touch locationInView:self];
10          double touchDeltaX = curLoc.x - prevLoc.x;
11          double touchDeltaY = curLoc.y - prevLoc.y;
12          [self moveX:touchDeltaX andY:touchDeltaY];
13          MapViewLocation mapLoc = self.location;
14          mapLoc.x += touchDeltaX;
15          mapLoc.y += touchDeltaY;
16          self.location = mapLoc;
17          if(self.info != nil){
18              self.info.hidden = TRUE;
19              self.info.label.text = @"Unknown";
20          }
21      }
22 }
```

À la manière du défilement de la carte, le déplacement d'une punaise se fonde sur les positions précédente et actuelle pour mettre à jour l'emplacement. Dans le cas du défilement de la carte, l'emplacement correspond au centre de la carte. Dans le cas du déplacement de la punaise, il correspond au point d'affichage supérieur gauche de la punaise et à sa latitude et sa longitude.

Puisque la punaise ne représente plus la position précisée à l'origine sur la carte, la description associée initialement à la punaise est probablement invalide. Elle est par conséquent fixée à `Unknown`. Par ailleurs, si le court message s'affiche pendant que l'utilisateur fait glisser la punaise, il est fermé de manière à réduire la charge processeur imposée par le tracé de la punaise et du message pendant leur déplacement. Cette fermeture est réalisée à la ligne 18.

L'attribut `info` de la classe `Pin` sert à l'affichage de la description fournie par la partie JavaScript de l'application. Il s'agit d'un objet `InfoWindow` qui est lui-même un `UIView` avec `label` pour seul attribut.

La méthode `initWithFrame:andDescription` de `InfoWindow`, donnée ci-après et définie dans le fichier *InfoWindow.m* présent dans `Classes:MapView`, fixe l'emplacement de la fenêtre pour l'affichage de la description. L'élément d'interface employé pour afficher cette description est un `UILabel`.

```
-(id)initWithFrame:(CGRect)frame
        andDescription:(NSString*)description{
    if (self = [super initWithFrame:frame]) {
        [self setBackgroundColor:[UIColor blackColor]];
        CGRect labelFrame = CGRectMake(0, 0,
                frame.size.width,
                frame.size.height);
        label = [[UILabel alloc] initWithFrame:labelFrame];
        label.text = description;
        label.backgroundColor = [UIColor clearColor];
        label.textColor = [UIColor whiteColor];
        label.textAlignment = UITextAlignmentCenter;
        [self addSubview:label];
        [label release];
    }
    return self;
}
```

En utilisant un `UILabel` pour afficher la description, les possibilités sont nombreuses. Nous pouvons changer la police de caractères, son alignement, sa taille, sa couleur et bien d'autres attributs, comme les ombres portées. Nous pourrions dessiner le texte directement au lieu d'utiliser un `UILabel`, mais le code serait plus long. Grâce au `UILabel` prédéfini, l'application résultante est plus facile à maintenir.

Les trois classes `MapView`, `Pin` et `InfoWindow` forment le module `MapView`. Elles contiennent le code nécessaire à l'affichage d'une carte Google de base, mais il est facile de les modifier pour ajouter des comportements plus complexes.

Par exemple, vous pouvez modifier la classe `InfoWindow` pour qu'elle présente d'autres détails, soit en changeant sa taille d'affichage, soit en affichant un autre `UIView` complet, comme dans le cas de l'application de cartographie d'Apple. Il est également possible de changer la classe `MapView` de manière à obtenir des indications routières. Même si ces extensions requièrent des connaissances en Objective-C, elles ne sont pas techniquement difficiles.

L'ajustement des positions de tous les objets `Pin` et de leur `InfoWindow` pendant le défilement ou le zoom de la carte est un autre point intéressant que nous n'avons pas encore abordé. Chacune de ces classes possède une méthode `moveX:andY`. Elles prennent en paramètre le

Openmirrors.com

nombre de pixels, dans les directions x et y, nécessaires au décalage de la punaise ou de la fenêtre d'information (voir le code ci-après).

Un programmeur Objective-C novice pourrait modifier la position par une ligne de code comme `[self frame].origin.x += xChange`. Cette ligne ne lance aucune exception, ne provoque aucune erreur de compilation, mais ne modifie pas la position de `Pin` ou de `InfoWindow`.

Dans les classes Objective-C qui dérivent de `UIView`, comme `Pin` et `InfoWindow`, le cadre qui représente la position supérieure gauche, ainsi que la largeur et la hauteur, est appliqué uniquement dans deux cas. Le premier correspond à son initialisation avec le message `initWithFrame`. Dans ce cas, la structure du cadre fournie en paramètres est utilisée pour dimensionner la vue. Le second correspond au remplacement de l'attribut `frame` par un autre, comme dans la méthode `moveX:andY` suivante.

```
- (void) moveX:(double) xChange andY:(double)yChange{
    CGRect frame = [self frame];
    frame.origin.x += xChange;
    frame.origin.y += yChange;
    self.frame = frame;
}
```

En raison de cette limitation, la manipulation de l'attribut `frame` de la vue n'a aucun effet.

La méthode `moveX:andY` doit être appelée pour chaque défilement ou zoom de la carte. Dans la méthode `touchesMoved:withEvent` de la classe `MapView`, chaque punaise reçoit ce message lors de la capture d'un événement.

```
for(int i = 0; i < numPins; i++){
    Pin *aPin = [pins objectAtIndex:i];
    [aPin moveX:touchDeltaX andY:touchDeltaY];
    [aPin.info moveX:touchDeltaX andY:touchDeltaY];
}
```

La méthode `touchesEnded:withEvent` de la place `MapView` prend en charge le zoom de la carte. Ce message est envoyé par l'appareil à un objet `MapView` après le traitement de tous les messages `touchesBegan` et `touchesMoved`.

Dans le code suivant, vous remarquerez qu'après avoir établi qu'il s'agit d'un double-toucher la position du toucher sur la carte est déterminée. Ensuite, la fonction JavaScript `zoomTo` est appelée, comme dans le cas du zoom d'une punaise.

```
1  -(void)touchesEnded:(NSSet *)touches
2          withEvent:(UIEvent *)event{
3
```

```
 4     if(OKToTouch){
 5         UITouch *touch = [touches anyObject];
 6         if ([touch tapCount] == 2){
 7             // Zoom et centrage.
 8             CGPoint curLoc = [touch locationInView:self];
 9
10             double latOffset = curLoc.y/self.pixelsPerDegLatitude;
11             double lngOffset = curLoc.x/pixelsPerDegLongitude;
12
13            double latitude = [[northWest objectAtIndex:0] doubleValue] - latOffset;
14            double longitude = [[northWest objectAtIndex:1] doubleValue] + lngOffset;
15             NSString *javaScriptCall = [[NSString alloc] initWithFormat:
               ➥@"zoomTo(%f, %f)",latitude, longitude];
16             NSString *mapDescription = [webMapView stringByEvaluatingJava
               ➥ScriptFromString:javaScriptCall];
17
18             [self setMapLatLngFrameWithDescription:mapDescription];
19             [self updatePinLocations];
20         }
21         else{
22             NSLog(@"toucher");
23         }
24     }
25 }
```

Une fois le zoom terminé, que ce soit dans le cas d'un double-toucher pour l'objet `Pin` ou `MapView`, un message `updatePinLocations` est envoyé à l'objet `MapView` (ligne 19 du code précédent). Le code suivant montre que cet appel provoque la mise à jour de la position de chaque punaise en fonction de sa latitude, de sa longitude et du facteur de zoom de la carte.

Le facteur de zoom est représenté par les attributs `pixelsPerDegLatitude` et `pixelsPerDegLongitude` de la classe `MapView` fixés lors d'un appel précédent à la méthode `setMapLatLngFrameWithDescription`.

```
1  -(void) updatePinLocations{
2      int numPins = [pins count];
3      for(int i = 0; i < numPins; i++){
4          Pin *aPin = (Pin*)[pins objectAtIndex:i];
5          double latitudeDelta = [[northWest objectAtIndex:0] doubleValue] -
           ➥aPin.location.latitude;
6          double longitudeDelta = aPin.location.longitude - [[northWest
           ➥objectAtIndex:1] doubleValue];
7
8          double yPixels = latitudeDelta * pixelsPerDegLatitude;
9          double xPixels = longitudeDelta * pixelsPerDegLongitude;
```

Openmirrors.com

```
10          MapViewLocation aPinLocation = aPin.location;
11          aPinLocation.x = xPixels - 10 - 4;
12          // Le visuel de la punaise se trouve dans l'image à 10 pixels
13          // à partir de la gauche.
14          aPinLocation.y = yPixels - 42 + 10;
15          // Le visuel de la punaise se trouve dans l'image à 10 pixels
16          // à partir du bas.
17
18          CGRect pinFrame = aPin.frame;
19          pinFrame.origin.x = aPinLocation.x;
20          pinFrame.origin.y = aPinLocation.y;
21          aPin.frame = pinFrame;
22          aPin.location = aPinLocation;
23
24          CGRect infoFrame = aPin.info.frame;
25          infoFrame.origin.x = aPinLocation.x - 100;
26          infoFrame.origin.y = aPinLocation.y - 30;
27          aPin.info.frame = infoFrame;
28      }
29 }
```

Comme dans la méthode `moveX:andY`, le cadre doit être retrouvé, modifié et initialisé. Ces opérations sont réalisées aux lignes 18 à 22 du code précédent pour la punaise et aux lignes 24 à 27 pour le `InfoWindow` associé à l'objet `Pin`.

La position des objets `Pin` et `InfoWindow` est mise à jour lors de chaque zoom. Il est possible que certains d'entre eux ne soient plus dans les limites de visibilité fixées par le `MapView` courant, mais ce n'est pas un problème. Ils sont simplement affichés dans une partie non visible, en dehors de l'écran. Le code n'a pas besoin de les masquer.

En résumé

Ce chapitre a expliqué comment embarquer des cartes Google dans une application et comment le module d'affichage d'une carte a été créé. Il a également montré comment manipuler les positions des vues personnalisées et des vues standard.

En utilisant judicieusement l'API JavaScript de Google avec une carte affichée par un `UIWebView`, il est possible d'effectuer un zoom et un recentrage.

Puisque toutes ces fonctionnalités sont incluses dans le framework QuickConnect, il vous suffit d'un seul appel JavaScript pour afficher une carte Google pleinement opérationnelle. Si le côté JavaScript du framework est simple d'utilisation, il est également facile d'utiliser le module de cartographie dans n'importe quelle application Objective-C pour l'iPhone. En réalité, seules quelques lignes de code sont nécessaires.

Les cartes Google personnalisées peuvent désormais être embarquées dans n'importe quelle application pour l'iPhone. La version 3.0 du système d'exploitation de l'iPhone rend même cette fonctionnalité encore plus facile.

Openmirrors.com



7

Bases de données

La plupart des applications écrites en JavaScript ont besoin d'un serveur web pour enregistrer des données. Depuis la version 2.0 du système d'exploitation de l'iPhone et sa classe `UIWebView`, il est possible d'enregistrer des données sur le téléphone sans passer par le réseau. Autrement dit, l'application que vous créez est une entité de première classe sur l'iPhone. Ce chapitre explique comment enregistrer et retrouver des données, et comment créer des bases de données et des tables. Il fournit une enveloppe JavaScript simple d'emploi pour accéder à la base de données SQLite disponible sur l'iPhone. Les premières sections décrivent l'utilisation de la base de données, les dernières présentent le code de l'enveloppe.

Section 1 : application BrowserDBAccess

L'application BrowserDBAccess a été créée pour vous aider à comprendre l'utilisation des bases de données dans les applications JavaScript. Elle crée une base de données SQLite nommée sampleDB, dans laquelle une table, `score`, est définie. Cette base de données existe dans le `UIWebView` et l'utilisateur peut la consulter. Les bases de données créées de cette manière sont persistantes entre les exécutions de l'application, même si elles n'ont pas été installées avec l'application. Cette dernière peut ainsi retrouver les données enregistrées

dans la base à chaque exécution. La Figure 7.1 montre l'exécution de cette application avant l'envoi d'une requête à la base de données.

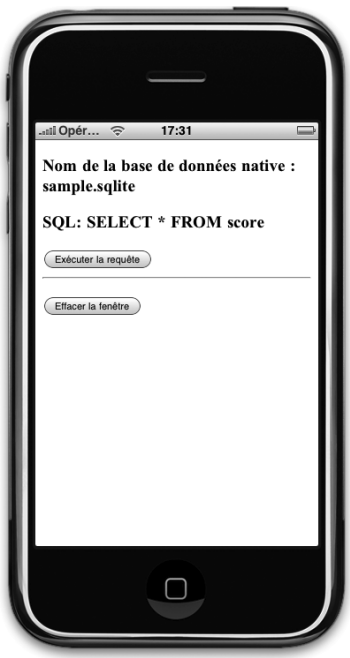


Figure 7.1
L'application BrowserDBAccess avant l'exécution de la requête.

La table `score` de l'application BrowserDBAccess comprend deux champs. Le premier, de type caractère nommé `player_name`, correspond à la clé primaire. Le second, nommé `score`, est de type entier. Puisque la clé primaire n'est pas incrémentée automatiquement, sa valeur doit être fournie chaque fois qu'un enregistrement est ajouté. BrowserDBAccess utilise plusieurs classes, méthodes et fonctions JavaScript prédéfinies pour accéder aux données de sampleDB.

Terminologie des bases de données

Le monde des bases de données possède sa propre terminologie. Les tables servent à regrouper des éléments semblables. Chacun de ces éléments est appelé enregistrement. Les enregistrements sont constitués de valeurs saisies dans des champs. Les champs sont définis par leur nom et leur type. Les enregistrements sont comparables aux lignes d'une feuille de calcul, les champs, aux colonnes. Les tables sont les feuilles de calcul. Si vous ajoutez un nouvel enregistrement à une table `chiens`, cela équivaut à ajouter une nouvelle ligne à une feuille de calcul `chiens`. En réalité, ce n'est pas vraiment identique, mais suffisamment comparable pour faciliter la compréhension.

Openmirrors.com

Une clé primaire identifie de manière unique chaque enregistrement d'une table. Il peut s'agir d'un entier ou d'une chaîne de caractères, mais les doublons sont interdits. Une clé primaire peut correspondre, par exemple, à la combinaison de votre nom d'utilisateur et de votre mot de passe ADC (*Apple Developer Connection*). Bien évidemment, si deux personnes avaient les mêmes informations d'ouverture de session, le résultat serait dramatique.

Une clé étrangère est un autre élément. Elle est utilisée pour lier des données enregistrées dans deux tables différentes. Imaginez les deux tables `propriétaires` et `chiens`. La table `propriétaires` possède une clé primaire pour chaque enregistrement. Pour lier les chiens et les propriétaires, une clé étrangère est ajoutée à la table `chiens`. Cette clé étrangère contient la clé primaire du propriétaire d'un chien dans la table `propriétaires`. Cela équivaut au numéro d'identification tatoué dans l'oreille du chien et qui permet de retrouver son propriétaire.

Section 2 : utilisation des bases de données SQLite avec *WebView*

L'utilisation des bases de données peut paraître intimidante. Un programmeur doit garder à l'esprit toute une ribambelle de pièges potentiels lors de l'enregistrement ou de l'obtention des informations. En raison de ces écueils, les programmeurs choisissent souvent de ne pas employer des bases de données pour le stockage de données simples.

Sur l'iPhone, la solution native pour le stockage des données des applications passe par une base de données SQLite embarquée, non un fichier texte ou binaire. La rapidité et la facilité d'utilisation de ce moteur de bases de données peut conditionner le succès de vos applications. Pour accélérer le développement, le framework QuickConnectiPhone propose une classe JavaScript qui vous évite tous les tracas associés à l'utilisation des bases de données.

Le fichier *DataAccessObject.js*, présent dans le groupe `QCiPhone` des modèles Dashcode et Xcode, contient une enveloppe qui prend en charge l'accès à la base de données SQLite. Les deux modèles incluent automatiquement ce fichier JavaScript dans le fichier *index.html* de l'application. La classe `DataAccessObject` définie par l'enveloppe comprend un constructeur et quatre méthodes (voir Tableau 7.1).

Cette classe JavaScript est employée dans le fichier *databaseDefinition.js* inclus dans votre application par les modèles. C'est dans ce fichier que vous devez indiquer la ou les bases de données utilisées par votre application.

Le code suivant, extrait du fichier *databaseDefinition.js* de l'application BrowserDBAccess, montre comment utiliser le constructeur pour créer une base de données gérée par le

Tableau 7.1 : API de `DataAccessObject`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`DataAccessObject (dbName, dbVersion, dbDescription, dbSize)`	`DataAccess-Object`	Crée un `Data-AccessObject` lorsqu'elle est appelée avec le mot clé `new`.	`dbName` – une chaîne d'iden-tification unique employée pour désigner la base de données. `dbVersion` – une chaîne donnée généralement sous forme d'un nombre à virgule flottante. `dbDescription` – une chaîne précisant le rôle de la base de données. `dbSize` – la quantité mini-male d'espace disque alloué à la base de données (en octets). Si ce paramètre est absent ou si `null` est passé, la taille par défaut est utilisée (5 Mo).
`getData(SQL, parameterArray)`	Aucune	Cette méthode permet d'extraire les informations, conformément à l'instruction SQL passée, à partir d'une base de données créée dans `UIWebView`.	`SQL` – une chaîne de com-mande SQL valide. `parameterArray` – un tableau des valeurs utilisées par la commande SQL dans le cas d'une instruction préparée.
`setData(SQL, parameterArray)`	Aucune	Cette méthode permet d'enre-gistrer des informa-tions, conformé-ment à l'instruction SQL passée, dans une base de données créée dans `UIWebView`.	`SQL` – une chaîne de com-mande SQL valide. `parameterArray` – un tableau des valeurs utilisées par la commande SQL dans le cas d'une instruction préparée.

Openmirrors.com

Tableau 7.1 : API de `DataAccessObject` *(suite)*

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`getNativeData(SQL, parameterArray)`	Aucune	Cette méthode permet d'obtenir des informations à partir d'une base de données installée avec l'application.	`SQL` – une chaîne de commande SQL valide. `parameterArray` – un tableau des valeurs utilisées par la commande SQL dans le cas d'une instruction préparée.
`setNativeData(SQL, parameterArray)`	Aucune	Cette méthode permet d'enregistrer des informations dans une base de données installée avec l'application.	`SQL` – une chaîne de commande SQL valide. `parameterArray` – un tableau des valeurs utilisées par la commande SQL dans le cas d'une instruction préparée.

moteur WebKit dans les applications hybrides pour l'iPhone. L'utilisation du moteur de base de données intégré au moteur se révèle appropriée lorsque toutes les données sont générées par l'application après son installation.

```
var sampleDatabase = new DataAccessObject("sampleDB", "1.0", "a sample database",2000);
```

Le code précédent crée une base de données nommée sampleDB, ou l'ouvre si elle existe déjà. Puisque WebKit prend en charge la création et l'utilisation de cette base de données, il a besoin de deux éléments d'information. Le premier est le numéro de version de la base. Vous pouvez choisir n'importe quelle valeur ; dans notre exemple, il est fixé à `1.0`. Le second indique la taille maximale prévue de la base de données ; dans notre exemple, il est fixé à `2000` octets.

La valeur retournée par l'appel au constructeur est un `DataAccessObject` connecté à la base de données SQLite sous-jacente et prêt à être utilisé. La modification de la base de données est ensuite possible, comme le montre le code suivant, qui utilise la méthode `setData` de `DataAccessObject`.

```
sampleDatabase.setData("CREATE TABLE IF NOT EXISTS score ('player_name' VARCHAR
➥PRIMARY KEY, 'score' INTEGER);");
```

La commande SQL correspond à la création d'une table, mais la méthode `setData` est employée pour toutes les requêtes SQL qui modifient les tables de la base, ajoutent ou suppriment des données, ou modifient la base d'une manière ou d'une autre.

Nous l'avons décrit au Chapitre 2, les fonctions de contrôle métier (BCF, *Business Control Function*) peuvent être utilisées pour obtenir des données à partir de la base. C'est précisément le rôle de la BCF `getScoresBCF` (voir ci-après) ; vous la trouverez dans le fichier *functions.js*.

Cette BCF se fonde sur la méthode `getData` de `DataAccessObject` pour obtenir des enregistrements à partir de la table `score`. L'appel correspondant se trouve à la ligne 3.

```
1 function getScoresBCF(parameters){
2     var SQL = 'SELECT * FROM score';
3     sampleDatabase.getData(SQL);
4 }
```

La Figure 7.2 illustre les données obtenues *via* cette BCF. Notez que `getScoresBCF` ne retourne aucune valeur, car la méthode `getData` est asynchrone. Le framework QuickConnectiPhone prend en charge la réception des données résultantes de la requête SQL et les passe aux objets de contrôle associés à la même commande que la BCF. La fonction de contrôle de l'affichage `displayScoresVCF`, définie dans le fichier *functions.js*, est associée à la même commande que `getScoresBCF`. Par conséquent, le framework lui passe les informations dès qu'elles sont disponibles.

L'insertion de données dans la table se fait de manière comparable. Pour insérer l'enregistrement de la personne nommée Jane, la méthode `setData` doit être appelée de la manière suivante :

```
sampleDatabase.setData("INSERT INTO score VALUES('Jane', 250)");
```

Les valeurs ne sont généralement pas figées dans les instructions SQL. Le plus souvent, elles sont saisies par l'utilisateur. Bien qu'il soit possible de construire une instruction SQL comme la précédente à partir des informations fournies par l'utilisateur, cette solution est dangereuse. Le code suivant montre comment éviter ce danger à l'aide des instructions préparées.

```
function setScoresBCF(parameters){
    var name = document.getElementById('nameField').value;
    var score = document.getElementById('scoreField').value;
    var statementParameters = [name, score];
    var SQL = "INSERT INTO score VALUES(?, ?)";
    sampleDatabase.setData(SQL, statementParameters);
}
```

Openmirrors.com

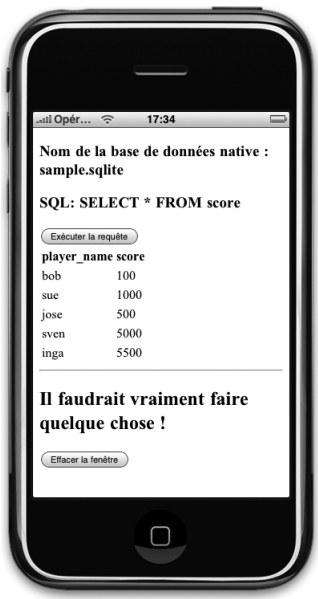


Figure 7.2
L'application BrowserAccessExample après avoir touché le bouton Exécuter la requête.

Vous remarquerez qu'il diffère du code qui appelle `setData` pour créer la table `score`. Dans la chaîne SQL, des points d'interrogation ont remplacé le nom et le score qui étaient supposés apparaître. Il s'agit de paramètres substituables employés dans les instructions préparées pour indiquer où doivent être insérées les données du tableau `statementParameters`. Le programmeur se charge d'ajouter au tableau les valeurs qui seront placées, dans l'ordre, dans la chaîne SQL. C'est pourquoi la variable `name` est ajoutée au tableau `statementParameters` avant la variable `score`.

Instructions préparées

Les instructions préparées améliorent énormément la sécurité des instructions SQL servant à enregistrer les données saisies par l'utilisateur. En réalité, elles constituent un bon moyen d'arrêter les attaques par injection SQL et devraient être utilisées plus qu'elles ne le sont actuellement.

Supposons que vous vouliez ajouter un enregistrement à une table nommée `preferences_utilisateur`, qui comprend les champs `lieu`, `couleur` et `chanson`. Vous pourriez procéder de la manière suivante :

```
"SELECT * FROM preferences_utilisateur WHERE nom = "+unNom
```

Cette manière de composer les instructions SQL est mauvaise, vraiment très mauvaise. Elle ouvre la porte aux attaques par injection SQL sur l'intégralité de la base de données.

Ces attaques permettent de pénétrer dans les bases de données. Vous ne devez pas composer les instructions SQL de cette manière.

Pour créer une chaîne SQL, voici la solution sécurisée :

```
"SELECT * FROM preferences_utilisateur WHERE nom = ?)"
```

Vous devez également créer un tableau qui contient les valeurs qui seront utilisées à la place des points d'interrogation. Puisqu'elles se fondent sur une instruction préparée, les appels à `setData` et `getData` s'occupent du reste.

Lorsqu'une instruction préparée est utilisée, la chaîne SQL est analysée avant que les points d'interrogation ne soient remplacés par les valeurs. Autrement dit, toute commande SQL malveillante placée dans les variables est détectée et rejetée ; pour la base de données, l'instruction SQL semble contenir une chaîne étrange. Un appel avec des points d'interrogation, tel que le précédent, ne retourne aucun résultat. En revanche, en composant l'instruction SQL par concaténation, un hacker peut facilement obtenir tout ce qu'il souhaite à partir d'une table.

Les instructions préparées protègent les bases de données des intrusions connues sous le nom attaques par injection SQL. Même s'il peut sembler idiot de protéger notre petite base de données contre les intrusions, ce n'est pas le cas. Si quelqu'un réussit à obtenir un accès non autorisé à un appareil ou au code qui s'y exécute, il trouvera toujours le moyen de causer des dommages. Tous les développements doivent se faire de manière sécurisée.

La suppression de données dans la base suit une procédure comparable. Puisque la base de données est modifiée, la méthode `setData` est employée. Puisque l'utilisateur saisit le nom à supprimer, une instruction préparée est utilisée.

```
function deleteScoreBCF(parameters){
    var name = document.getElementById('nameField').value;
    var statementParameters = [blogName];
    database.setData('DELETE FROM score where name = ?',
            statementParameters);
}
```

Section 3 : utilisation de bases de données SQLite natives

Outre la possibilité de créer et d'utiliser des bases de données dans le moteur WebKit disponible dans toute application hybride pour l'iPhone, il est également possible d'utiliser des bases de données natives. Grâce à ces bases de données, les développeurs d'applications peuvent inclure des données existantes dans l'application installée. Par exemple,

Openmirrors.com

vous pourriez inclure dans votre application tout un ensemble de localisations GPS standard afin d'aider les utilisateurs à trouver des produits sur un marché de vente au détail. Au lieu de télécharger et d'enregistrer les données au démarrage de l'application, vous pouvez fournir un fichier de base de données SQLite avec votre application.

D'un point de vue fonctionnel, l'application nativeDBAccess est identique à BrowserDBAccess, mais elle utilise un fichier de base de données SQLite nommé *sample.sqlite*. La Figure 7.3 montre la présence de ce fichier dans le groupe `Resources` du projet Xcode.

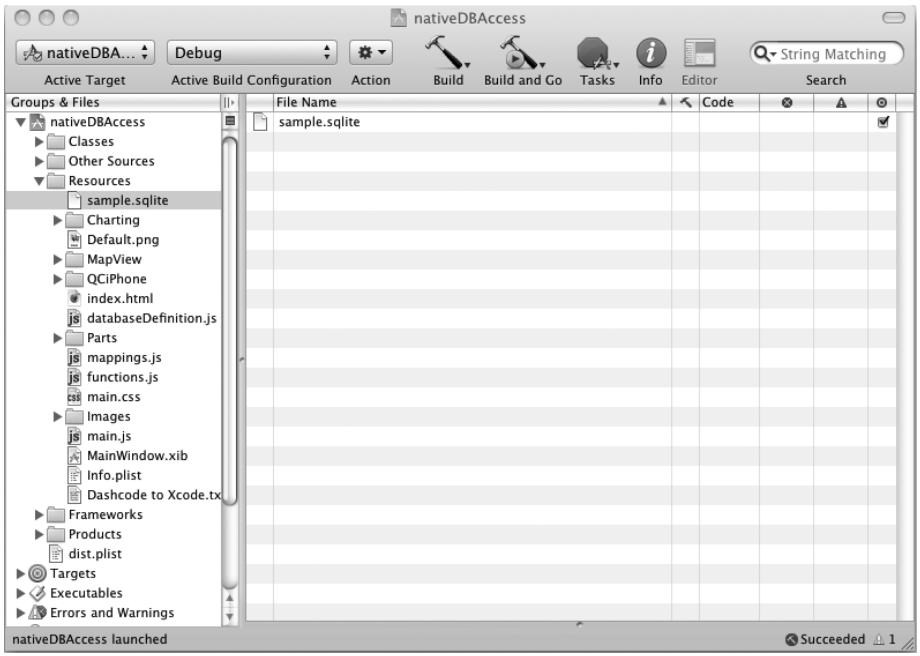


Figure 7.3
Les ressources de l'application nativeDBAccess.

Le code JavaScript nécessaire à la création d'un `DataAccessObject` qui enveloppe les appels à une base de données native diffère de celui de la Section 2 employé avec les bases de données du moteur WebKit. Si la définition de la base de données se fait toujours dans le fichier *databaseDefinition.js*, dans le cas des bases de données natives seul le nom du fichier de base de données est indiqué. Les paramètres de version et de taille sont superflus, car les bases de données natives ont une taille illimitée (avec raison) et la gestion des versions fait partie du processus de construction et de distribution de l'application.

```
var sampleDatabase = new DataAccessObject("sample.sqlite");
```

Notez qu'aucun appel ne crée une table, car la table `score` existe déjà dans le fichier *sample.sqlite*. Bien qu'il soit possible de créer des tables dans des bases de données natives, cette opération est inhabituelle. Les tables sont généralement ajoutées aux fichiers de base de données avant la distribution de l'application sur l'App Store.

Pour obtenir des données depuis une base native, la méthode est pratiquement identique à celle mise en place pour les bases de données WebKit. La seule différence, illustrée dans le code suivant, est que la méthode `getData` est remplacée par la méthode `getNativeData`. Pour le développeur de l'application, ces deux méthodes ont un comportement identique.

```
function getScoresBCF(parameters){
    debug('Obtention des scores depuis la base de données');
    var SQL = 'SELECT * FROM score';
    sampleDatabase.getNativeData(SQL);
}
```

De même que `getData` décrite à la Section 2, la méthode `getNativeData` est asynchrone. Par conséquent, dès que les données sont disponibles, le framework les passe aux objets de contrôle associés à la même commande que la BCF, dans ce cas `displayScoresVCF`. La Figure 7.4 présente la page générée par cette VCF dans l'application nativeDBAccess.



Figure 7.4
L'application nativeDBAccess après avoir touché le bouton d'exécution de la requête.

Openmirrors.com

L'ajout et la suppression des données, ainsi que la modification de la base de données, se font exactement de la même manière qu'à la Section 2, excepté que la méthode `setNative-Data` est invoquée à la place de la méthode `setData`. En tant que développeur de l'application, les différences de comportements vous sont transparentes.

Section 4 : utilisation de *DataAccessObject* avec les bases de données du moteur WebKit

Les sections précédentes ont expliqué comment utiliser `DataAccessObject` pour manipuler des bases de données SQLite WebKit ou natives, sans avoir à connaître leurs détails de fonctionnement internes. Cette section révèle ces détails et montre comment les utiliser. Si vous souhaitez simplement exploiter `DataAccessObject`, sans volonté d'en connaître le fonctionnement, vous pouvez sauter cette section.

Grâce à la classe `DataAccessObject`, définie dans le fichier *DataAccessObject.js* du groupe `QCiPhone`, le programmeur n'est pas obligé de posséder des connaissances détaillées quant à SQLite. Elle a été conçue avec des méthodes et des constructeurs comparables à ceux de l'enveloppe AJAX `ServerAccessObject` décrite au Chapitre 8. En simplifiant l'API, les programmeurs qui ne sont pas familiers de SQLite ont la possibilité d'enregistrer des données sans passer par une longue phase d'apprentissage.

Le constructeur de `DataAccessObject` est la plus simple de toutes ces méthodes. Il fixe et définit les méthodes de l'objet comme des fonctions anonymes. Autrement dit, aucun attribut n'est nécessaire pour enregistrer les paramètres passés au constructeur.

Fonctions anonymes

Lorsque quelqu'un est anonyme, cela signifie que vous ne connaissez pas son nom. Lorsqu'une fonction est anonyme, cela signifie qu'elle n'a pas de nom. En JavaScript, une fonction standard a un nom déclaré à l'aide d'une ligne semblable à la suivante :

```
function aboyer(){}
```

Dans ce cas, la fonction se nomme `aboyer`. Lorsqu'une fonction doit être passée à une autre fonction, il est fréquent d'employer des fonctions sans nom déclaré. Ces fonctions sont anonymes.

Pour passer une fonction anonyme à `aboyer`, nous utilisons le code suivant :

```
aboyer(new function(){
   // Faire quelque chose ici.
});
```

Notez que les opérateurs de portée de la fonction anonyme, `{}`, sont contenus dans les opérateurs de paramètre de la fonction `aboyer`, `()`. La fonction `aboyer` peut invoquer ou enregistrer cette fonction comme bon lui semble.

Les fonctions anonymes ont accès aux variables locales définies dans la fonction dans laquelle elles ont été déclarées. Par conséquent, le code suivant est valide :

```
String type = 'doberman';
aboyer(new function(){
    if(type == 'boxer'){
   }
  else if(type == 'doberman'){
  }
  ...
});
```

Cette possibilité est pratique lorsque des fonctions ou des méthodes prennent des fonctions en paramètres, comme nous le verrons plus loin dans cette section.

À l'instar de nombreux outils puissants, il est important de ne pas faire un usage excessif des fonctions anonymes si elles ne sont pas nécessaires.

Nous avons vu au Tableau 7.1 et aux Sections 2 et 3 que `DataAccessObject` comprend deux principaux groupes de méthodes. L'un prend en charge les données enregistrées ou obtenues avec une base de données WebKit, tandis que l'autre s'occupe des transferts entre la partie JavaScript de l'application et un fichier de base de données SQLite fourni. Les méthodes fondées sur WebKit utilisent principalement du code JavaScript et sont examinées en premier.

Les méthodes `getData` et `setData` sont des façades. Elles contiennent peu de code et se fondent sur une troisième méthode pour réaliser la plus grande partie du travail. Leur seule tâche véritable consiste à assembler les valeurs enregistrées dans la variable `passThroughParameters`.

```
this.getData = function(SQL, preparedStatementParameters){
    var passThroughParameters = generatePassThroughParameters();
    this.dbAccess(SQL, preparedStatementParameters,
            false, passThroughParameters);
}
```

Puisque le comité de normalisation du W3C responsable des spécifications HTML 5 exige que tous les appels aux fonctionnalités de base de données du moteur WebKit soient asynchrones, certaines informations concernant l'état courant de l'application doivent être

Openmirrors.com

passées avec les requêtes pour qu'elles puissent être utilisées par la suite. La fonction `generatePassThroughParameters`, définie dans *QCUtilities.js*, réunit ces valeurs.

Le code suivant montre que cela inclut la commande courante pour laquelle des objets de commandes sont invoqués, le nombre de BCO déjà appelés, les paramètres passés à toutes les fonctions de contrôle, comme `globalParamArray`, et un tableau contenant tous les résultats déjà générés par les appels à d'autres BCF, comme `globalBCFResults`.

```
function generatePassThroughParameters(){
    var passThroughParameters = new Array();
    passThroughParameters.push(window.curCmd);
    passThroughParameters.push(window.numFuncsCalled);
    passThroughParameters.push(globalParamArray);
    passThroughParameters.push(window.globalBCFResults);
    return passThroughParameters;
}
```

Ces valeurs servent ensuite au framework pour garantir que les autres fonctions de contrôle associées à la commande `curCmd` dans le fichier *mappings.js* sont exécutées comme si tous les appels étaient synchrones. Pour de plus amples informations concernant cette procédure, consultez le Chapitre 2.

Le tableau `passThroughParameters` est transmis à la méthode `dbAccess`, avec un indicateur qui précise si les instructions contenues dans la variable `SQL` doivent être traitées comme une modification des données dans la base. Dans la méthode `getData`, cet indicateur est à `false`.

La méthode `dbAccess` est au cœur de la classe `DataAccessObject` pour les bases de données du moteur WebKit. Elle effectue tout le travail demandé par les appels à `getData` et à `setData`.

Pour comprendre cette méthode, il faut tout d'abord comprendre l'API JavaScript sous-jacente de SQLite. Cette API fait partie du standard HTML 5 à venir et est implémentée dans le moteur WebKit employé par le `UIWebView` de toutes les applications hybrides pour l'iPhone et le navigateur Mobile Safari. La dernière version de ce standard est disponible à l'adresse **http://www.w3.org/html/wg/html5/#sql**. Ce document décrit plusieurs objets et méthodes recensés dans les tableaux suivants.

`Database` constitue l'élément de base de cette API. Le Tableau 7.2 décrit une fonction associée à cet objet et l'une de ces méthodes. `openDatabase` est une fonction de fabrique qui instancie un objet `Database` à notre place.

Selon le document de normalisation, tous les paramètres de `openDatabase` sont facultatifs. Toutefois, il est peu judicieux de ne pas déclarer un nom de base de données. Si vous utilisez plusieurs bases de données dans différentes applications, chacune doit posséder un

Tableau 7.2 : API de `Database`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`openDatabase (dbName, dbVersion, dbDescription, dbSize)`	Objet `Database`	Une fonction de fabrique qui crée un objet `Database` en vue de son utilisation ultérieure.	`dbName` – une chaîne d'identification unique employée pour désigner la base de données. `dbVersion` – une chaîne, généralement donnée sous forme d'un nombre à virgule flottante. `dbDescription` – une chaîne précisant l'objectif de la base de données. `dbSize` – la quantité minimale d'espace disque alloué à la base de données (en octets). Si ce paramètre est absent ou si `null` est passé, la taille par défaut est utilisée (5 Mo).
`transaction(executionCallback, errorCallback, successCallback)`	Objet `SQLTransaction`	Cette méthode crée un objet `SQLTransaction` utilisé pour les mises à jour et les requêtes sur la base de données.	`executionCallback` – une fonction qui contient le code nécessaire à l'exécution du SQL. `errorCallback` – une fonction facultative qui est appelée lorsque la transaction a échoué. L'annulation des opérations sur la base de données n'est pas effectuée dans cette méthode, car cette procédure est automatique en cas d'échec. `successCallback` – une fonction facultative qui est appelée lorsque la transaction a réussi.

Openmirrors.com

nom distinct. Si `dbName` reste vide, il est possible qu'une même base de données soit partagée par toutes les applications. Elle risque donc de subir les opérations malencontreuses des applications écrites par d'autres programmeurs. Pour protéger vos bases de données, vous devez utiliser une règle comparable à celle de la *même origine* qui contraint les appels AJAX dans un navigateur.

Cette règle de protection appliquée aux bases de données stipule que seules les applications ayant la même origine peuvent accéder à une base de données. Si vous utilisez des bases de données dans des applications web hébergées sur `www.monsite.fr` et sur `web.monsite.fr`, ces bases sont accessibles par les différentes applications web. Autrement dit, toutes les applications qui proviennent de `monsite.fr` peuvent accéder à toutes les bases de données qui ont cette origine, quelle que soit l'application qui a créé la base de données, même dans un sous-domaine de `monsite.fr`.

Dans les applications hybrides, l'objet `UIWebView` remplace le navigateur web et, par conséquent, il n'existe aucune *règle de même origine* qui limite les requêtes AJAX. Cela n'est pas encore établi, mais nous pouvons supposer que la restriction de même origine sur les bases de données n'est pas plus effective. C'est pourquoi la seule protection de votre base de données contre les accès par d'autres applications réside dans l'utilisation d'un nom et d'une version inconnus des autres programmeurs. Vous devez donc donner un nom et une version à votre base de données.

La méthode `transaction` est employée par tous les appels SQL. En réalité, il est impossible d'exécuter des instructions SQL sur des bases de données WebKit sans passer par un objet `SQLTransaction` créé par la méthode `transaction`. Par conséquent, dans les applications hybrides pour l'iPhone, tous les appels JavaScript concernant une base de données sont automatiquement transactionnels.

Les développeurs sont souvent préoccupés par l'annulation des opérations sur la base de données. En général, cette annulation est effectuée lorsque le code écrit par le programmeur détecte l'échec d'une transaction. Lorsque vous utilisez les fonctionnalités JavaScript de base de données, ces annulations ne constituent pas un problème car les transactions s'en occupent automatiquement en cas d'échec.

Vous ne devez pas exécuter une instruction SQL `ROLLBACK` en cas d'échec d'une transaction. Cette opération a déjà été effectuée et cela risque de provoquer des dysfonctionnements. La fonction `errorCallBack` passée à la méthode `transaction` n'est pas utilisée dans ce but. Elle sert uniquement à signaler les échecs.

L'objet `SQLTransaction` n'offre qu'une seule méthode, `executeSQL` (voir Tableau 7.3). Elle accepte toute instruction SQL que vous lui passez, mais il est imprudent d'assembler une instruction SQL et de l'exécuter. Vous devez utiliser à la place les instructions

préparées. Pour de plus amples informations concernant les instructions préparées, consultez la Section 2.

Tableau 7.3 : API de `SQLTransaction`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`executeSQL(sqlStatement, arguments, successCallback, errorCallback)`	Aucune	Cette méthode exécute une chaîne d'instruction SQL quelconque.	`sqlStatement` – une chaîne contenant une instruction SQL valide. Elle peut inclure des points d'interrogation (`?`) si elle doit être traitée comme une instruction préparée. `arguments` – un tableau facultatif des valeurs utilisées pour remplacer les points d'interrogation (`?`) dans les instructions préparées. `successCallback` – une fonction facultative qui est appelée lorsque l'exécution de `sqlStatement` a réussi. `errorCallback` – une fonction facultative qui est appelée lorsque l'exécution de `sqlStatement` a échoué.

Outre l'instruction SQL, le paramètre facultatif `arguments` est passé. Il s'agit d'un tableau de chaînes de caractères qui vont remplacer les points d'interrogation inclus dans l'instruction SQL. Ils correspondent aux variables de l'instruction préparée créée par l'appel à `executeSQL`. Tous les appels à `executeSQL` créent une instruction préparée, même lorsqu'il n'y a aucun paramètre substituable. Son utilisation apporte donc plus de sûreté, sans changer la rapidité de l'application.

Les deux derniers paramètres désignent les fonctions de rappel exécutées lorsque l'instruction, non la transaction, réussit ou échoue. La fonction de succès reçoit un objet `SQLResultSet` de la part de la méthode `executeSQL`. Le Tableau 7.4 présente l'API de `SQLResultSet`. La fonction d'échec reçoit un objet `SQLError` ; son API est décrite au

Openmirrors.com

Tableau 7.6. Pour traiter les résultats de l'exécution de l'instruction SQL, vous devez implémenter ces fonctions et les indiquer dans l'appel à la méthode `executeSQL`.

L'objet `SQLResultSet` contient les informations que vous avez demandées au travers de l'instruction SQL. Il inclut également deux éléments intéressants : les attributs `insertID` et `rowsAffected`.

Lorsque la clé primaire d'une table est à incrémentation automatique et qu'un enregistrement est ajouté à cette table, l'attribut `insertID` du `SQLResultSet` contient la valeur générée pour la clé. Ce fonctionnement est très utile lorsque la clé est requise pour ajouter des données associées dans d'autres tables.

Après l'exécution réussie d'une instruction SQL d'insertion ou de mise à jour, l'attribut `rowsAffected` contient le nombre de lignes qui ont été ajoutées ou modifiées. Cela permet de valider le comportement des instructions SQL complexes.

Tableau 7.4 : API de `SQLResultSet`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`insertID`	Aucune	Un attribut entier en lecture seule qui contient l'identifiant de l'enregistrement ajouté si le champ correspondant est à incrémentation automatique.	Aucun
`rowsAffected`	Aucune	Un attribut entier en lecture seule qui contient le nombre de lignes ajoutées ou modifiées par une opération de type mise à jour.	Aucun
`rows`	Aucune	Un attribut `SQLResultSetRowList` qui contient toutes les lignes retournées par une instruction de type requête.	Aucun

Le troisième attribut de `SQLResultSet` se nomme `rows`. Il contient un `SQLResultSetRowList`, qui représente un tableau d'enregistrements. Le Tableau 7.5 montre que son attribut

`length` précise le nombre d'enregistrements obtenus par une instruction de type `SELECT`. Il correspond au nombre de lignes contenues dans le tableau.

Tableau 7.5 : API de `SQLResultSetRowList`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`length`	Aucune	Un attribut en lecture seule qui contient le nombre d'enregistrements obtenus par une instruction de type requête.	Aucun.
`item(index)`	Tableau	Une méthode qui retourne un enregistrement sous forme de tableau associatif ou de mappe JavaScript.	`index` – l'indice de l'enregistrement dans le jeu des résultats à retourner.

Il offre également la méthode `item` pour accéder à chaque ligne des résultats. Grâce à cette méthode et à l'attribut, il est facile d'itérer sur les lignes et leurs valeurs à l'aide de boucles `for`. Voici la solution généralement employée pour effectuer ces itérations :

```
1 for( var i = 0; i < aResultSet.length; i++){
2     var aRow = aResultSet.item(i);
3     for(key in aRow){
4         var aValue = aRow[key];
5         // Exploiter la clé et la valeur.
6     }
7 }
```

Bien que le code précédent puisse être considéré par beaucoup comme une solution standard, son exécution n'est pas optimale car, à la ligne 1, la taille du jeu de résultats est obtenue à chaque tour de boucle externe. Un autre gaspillage est généré par la répétition de la boucle `for-each` à la ligne 3.

Les boucles JavaScript `for-each` sont particulièrement gourmandes en cycles processeur, notamment lorsqu'elles sont employées dans d'autres boucles. Plus loin dans cette section, la description de la méthode `dbAccess` montrera comment optimiser ce code.

Si l'exécution de l'instruction SQL provoque une erreur, quelle qu'en soit la raison, un objet `SQLError` est généré à la place de l'objet `SQLResultSet`. De même qu'un `SQLResultSet` est passé à la fonction de réussite indiquée en argument de la méthode `executeSQL`, un `SQLError` est passé à la fonction d'échec.

Openmirrors.com

La fonction de succès ne reçoit jamais un `SQLError` et la fonction d'erreur ne reçoit jamais un `SQLResultSet`. Ainsi, chacune de ces fonctions remplit un seul objectif et devient donc plus facile à créer, à écrire et à maintenir.

L'objet `SQLError` contient deux attributs qui correspondent à un numéro de code d'erreur et à un message d'erreur générés par SQLite (voir Tableau 7.6).

Tableau 7.6 : API de `SQLError`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`code`	Aucune	Un attribut en lecture seule qui contient le numéro du code de l'erreur. Les codes possibles sont : `0` – la transaction a échoué pour une raison inconnue. `1` – L'instruction a échoué pour une raison inconnue. `2` – L'instruction a échoué car la version attendue de la base de données ne correspond pas à sa version réelle. `3` – L'instruction a échoué car le nombre de données retournées était trop important. Essayez d'utiliser le modificateur SQL `LIMIT`. `4` – L'instruction a échoué car les limites mémoire ont été atteintes et l'utilisateur n'a pas accepté de les augmenter. `5` – L'instruction a échoué en raison d'un échec du verrouillage dans la transaction. `6` – Une instruction `INSERT`, `UPDATE` ou `REPLACE` a échoué car elle ne respecte pas une contrainte, comme un cas de clé unique dupliquée.	Aucun.
`message`	Aucune	Un attribut en lecture seule qui contient un message d'erreur approprié.	Aucun.

`code` et `message` sont utiles aux programmeurs, mais ne doivent pas être présentés aux utilisateurs, car ils ne sauront pas ce qu'ils représentent. Ces attributs doivent servir à journaliser l'erreur et à afficher une information utile aux utilisateurs.

La méthode `dbAccess` de `DataAccessObject` les utilise. Nous l'avons mentionné précédemment dans cette section, cette méthode prend cinq arguments et réalise le travail des

méthodes de façade `getData` et `setData`. Son quatrième argument indique si la requête modifie ou consulte la base de données. Sa valeur est fixée par la fonction de façade qui invoque `dbAccess`.

En examinant la méthode `dbAccess`, vous constaterez qu'elle emploie plusieurs fonctions anonymes. La première est passée à la ligne 3 en premier et seul paramètre de la méthode `transaction`. Il pourrait sembler plus facile de la définir comme une fonction normale ailleurs dans le code et de la passer en premier argument, mais, dans ce cas, le code restant serait difficile, voire impossible, à implémenter car des variables accessibles grâce aux fonctions anonymes ne le seraient alors plus. La ligne 2 du code suivant instancie l'objet `QueryResult`, qui est utilisé comme valeur de retour de la méthode `dbAccess`.

Pour qu'un objet `QueryResult` soit véritablement utile, il doit être disponible depuis l'intérieur de la fonction `executeSQL` de la transaction. Sans les fonctions anonymes, la seule manière de procéder consiste à le rendre global.

S'il est déclaré de manière globale, un seul accès à la base de données ne peut avoir lieu à la fois. Puisque tous les accès à la base de données sont asynchrones, ce fonctionnement est impossible à garantir. Par conséquent, la meilleure approche consiste à opter pour des fonctions anonymes. Cette même logique s'applique aux fonctions passées à la méthode `executeSQL` de la transaction elle-même.

La méthode `executeSQL` de l'objet `Transaction` peut recevoir deux fonctions en paramètres. Les bonnes pratiques stipulent qu'elles doivent toujours être passées. Le second paramètre correspond à la fonction qui traite les résultats lorsque la requête se passe parfaitement (voir Tableau 7.6). La déclaration de cette fonction commence à la ligne 10 du code ci-après.

Dans la fonction de réussite, tout identifiant généré par l'instruction SQL est enregistré dans le `QueryResult` instancié à la ligne 7. Si des lignes de la base de données sont modifiées, leur nombre est également indiqué, mais uniquement si le paramètre `treatAsChangeData` de la méthode `dbAccess` est fixé à `true`.

Si des données n'ont pas été modifiées, une requête a dû être exécutée. Les lignes 23 à 51 prennent en charge ce cas. Dans ces lignes, un tableau JavaScript à deux dimensions est créé et associé au `QueryResult`. Toutes les valeurs qui se trouvent dans le jeu de résultats `SQLResult` y sont enregistrées. En transférant les données dans un tableau JavaScript standard, vous pouvez accéder à ces données dans un autre code, sans connaître la structure de l'objet `SQLResult` ni les champs renvoyés par la requête. Si vous souhaitez connaître les noms des champs, ils sont également présents dans l'objet `QueryResult`. Les noms et les valeurs des champs conservent l'ordre dans lequel ils se trouvent dans le `SQLResult`.

```
1  this.dbAccess = function(SQL, preparedStatementParameters,
2          treatAsChangeData, passThroughParameters){
3      if(!this.db){
```

Openmirrors.com

```
        this.db = openDatabase(dbName, dbVersion,
                dbDescription, dbSize);
    }
    var queryResult = new QueryResult();
    this.db.transaction(function(tx) {
       tx.executeSql(SQL, preparedStatementParameters,
                function(tx, resultSet) {
            if(treatAsChangeData){
                try{
                    queryResult.insertedID = resultSet.insertId;
                    queryResult.rowsAffected =
                            resultSet.rowsAffected;
                }
                catch(ex){
                    // Il doit s'agir une mise à jour.
                    queryResult.rowsAffected =
                            resultSet.rowsAffected;
                }
            }
            else{
                // La base de données n'a pas été modifiée.
                // Il doit s'agir d'une requête.
                queryResult.numRowsFetched =
                        resultSet.rows.length;
                var dataArray = new Array();
                queryResult.numResultFields = 0;
                queryResult.fieldNames = new Array();
                if(queryResult.numRowsFetched > 0){
                   // Obtenir les identifiants des champs dans le jeu de résultats.
                    firstRecord = resultSet.rows.item(0);
                    var numFields = 0;
                    for(key in firstRecord){
                        queryResult.fieldNames.push(key);
                        numFields++;
                    }
                    queryResult.numResultFields = numFields;
                    var numRecords =
                            queryResult.numRowsFetched;
                    for(var i = 0; i < numRecords; i++){
                        var record = resultSet.rows.item(i);
                        var row = new Array();
                        dataArray.push(row);
                        for(var j = 0; j < numFields; j++){
                            row.push(
                                    record[queryResult.fieldNames[j]]);
                        }
                    }
```

```
51                  }
52                  queryResult.data = dataArray;
53              }
54              if(window.callFunc){
55                  var theResults = new Array();
56                  theResults.push(queryResult);
57                  theResults.push(passThroughParameters);
58                  requestHandler(passThroughParameters[0],
59                          passThroughParameters[2], theResults);
60              }
61          }// Fin de la fonction de rappel en cas de succès de l'exécution.
62          , function(tx, error) {
63              queryResult.errorMessage = error.message;
64              if(window.callFunc){
65                  var theResults = new Array();
66                  theResults.push(queryResult);
67                  theResults.push(passThroughParameters);
68                  requestHandler(passThroughParameters[0],
69                          passThroughParameters[2], theResults);
70              }
71          }// Fin de la fonction de rappel en cas d'échec de l'exécution.
72          );// Fin de l'appel principal à executeSql.
73      });// Fin de la fonction de rappel de la transaction.
74  }// Fin de la méthode dbAccess.
```

Quel que soit le type de l'instruction SQL exécutée, modification ou interrogation de la base de données, l'objet `QueryResult` créé au début de la fonction est envoyé par le framework à toutes les fonctions de contrôle restantes non invoquées. Cela se passe aux lignes 58 et 59 à l'aide de la fonction `requestHandler` décrite au Chapitre 2.

La ligne 62 débute la déclaration de la fonction de traitement des erreurs pour l'objet `Transaction`. Cette fonction doit insérer le message d'erreur généré par SQLite dans le `QueryResult` et le transmettre aux fonctions de contrôle restantes. Pour de plus amples informations concernant les fonctions de contrôle, consultez le Chapitre 2.

Section 5 : utilisation de *DataAccessObject* avec les bases de données natives

Pour accéder aux bases de données natives, il faut moins de code JavaScript que pour accéder aux bases de données du moteur WebKit. Une grande partie du travail est effectuée du côté Objective-C du framework.

Openmirrors.com

À l'instar de `getData` et de `setData`, les méthodes `getNativeData` et `setNativeData` sont des façades. Toutefois, elles appellent non pas la méthode `dbAccess` de `DataAccessObject`, mais la fonction `getDeviceData` définie dans le fichier *QCUtilities.js*.

```
this.getNativeData =
        function(SQL, preparedStatementParameters){
            getDeviceData(dbName, SQL, preparedStatementParameters);
        }
this.setNativeData =
        function(SQL, preparedStatementParameters){
            setDeviceData(dbName, SQL, preparedStatementParameters);
        }
```

La fonction `getDeviceData` est constituée de deux parties. La première, représentée par les lignes 4 à 16, construit un tableau des informations nécessaires à l'exécution de la requête en Objective-C. Cela comprend notamment les paramètres généraux décrits à la section précédente. Ces paramètres sont requis, car, comme dans le traitement des requêtes aux bases de données du moteur WebKit, la requête est asynchrone. Pour de plus amples informations concernant les appels asynchrones, consultez le Chapitre 2.

Le tableau créé contient le nom de la base de données, l'instruction SQL à exécuter, les paramètres de l'instruction SQL préparée et les paramètres généraux. La section précédente de ce chapitre a donné les explications concernant les instructions préparées.

```
 1 function getDeviceData(dbName, SQL,
 2         preparedStatementParameters, callBackParams){
 3     if(dbName && SQL){
 4         var dataArray = new Array();
 5         dataArray.push(dbName);
 6         dataArray.push(SQL);
 7         if(preparedStatementParameters){
 8             dataArray.push(preparedStatementParameters);
 9         }
10         else{
11             // Placer un paramètre substituable.
12             dataArray.push(new Array());
13         }
14         var callBackParameters =
15                 generatePassThroughParameters();
16         dataArray.push(callBackParameters);
17
18         var dataString = JSON.stringify(dataArray);
19         makeCall("getData", dataString);
20     }
21     return null;
22 }
```

Lorsque les données sont prêtes, elles sont converties en une chaîne JSON passée à la fonction `makeCall`. Nous l'avons vu au Chapitre 4, cette fonction active le côté Objective-C du framework. Il s'agit là de la deuxième partie importante de la fonction `getDeviceData`. Sans elle, les bases de données natives seraient inaccessibles. Pour de plus amples informations concernant JSON, consultez l'Annexe A.

À l'instar des fonctions du Chapitre 4 qui accèdent aux données natives, des objets de contrôle Objective-C sont nécessaires à l'obtention des données depuis la base. Ces deux objets, `SetDataBCO` et `GetDataBCO`, interagissent avec la base de données, qu'elle soit modifiée ou consultée. Ce fonctionnement est comparable à celui des fonctions JavaScript `getData` et `setData`.

```
+ (id) doCommand:(NSArray*) parameters{
    if( [parameters count] >= 3){
        NSString *dbName = [parameters objectAtIndex:1];
        NSString *SQL = [parameters objectAtIndex:2];
        NSArray *perparedStatementValues = nil;
        if([parameters count] == 4){
            perparedStatementValues = [parameters objectAtIndex:3];
        }
        SQLiteDataAccess *aDBAccess =
                [SQLiteDataAccess getInstance:dbName isWriteable:YES];
        return [aDBAccess getData:SQL withParameters:perparedStatementValues];
    }
    return nil;
}
@end
```

Par ailleurs, comme les fonctions JavaScript `getData` et `setData`, le code de traitement dans ces BCO est court. La commande `doCommand` permet d'obtenir le nom de la base de données, l'instruction SQL et les paramètres de l'instruction préparée provenant de la requête JavaScript.

Lorsque tous ces éléments d'informations sont disponibles, la méthode `getData` de l'objet `SQLiteDataAccess` est invoquée. Cet objet est essentiellement un clone de l'objet JavaScript `DataAccessObject`. Il offre les méthodes `getData` et `setData`, mais, contrairement à la version JavaScript, la version Objective-C est un singleton. Pour de plus amples informations concernant les singletons, consultez le Chapitre 4.

Les méthodes `getData` et `setData` de `SQLiteDataAccess` sont des façades pour la méthode `dbAccess`. De même que la méthode JavaScript `dbAccess`, la version Objective-C est particulièrement complexe. Cela provient de la complexité de l'API SQLite définie

Openmirrors.com

par la bibliothèque dynamique *libsqlite3.0.dylib* fournie avec chaque iPhone et iPod Touch (voir Tableau 7.7).

Tableau 7.7 : API de SQLite3

Objet/fonction	*Valeur de retour*	*Description*	*Paramètres*
`sqlite3`	Aucune	Un objet qui représente la base de données SQLite en mémoire.	Aucun.
`sqlite3_open (filePath, &aDatabase)`	`int SQLITE_OK` en cas de succès de l'ouverture	Ouvre le fichier de base de données indiqué par `filePath` et enregistre le pointeur dans `aDatabase`.	`filePath` – le chemin complet du fichier de base de données SQLite sur la machine. `aDatabase` – une référence vers un pointeur `SQLite3` qui représente la base de données en mémoire.
`sqlite3_close (aDatabase)`	`void`	Ferme les connexions au fichier de base de données SQLite.	`aDatabase` – un pointeur `SQLite3` fixé dans la fonction `sqlite3_open`.
`sqlite3_errmsg (aDatabase)`	`const char *`	Obtient la dernière erreur générée.	`aDatabase` – un pointeur `SQLite3` vers la base de données dont on souhaite obtenir un message d'erreur.
`sqlite3_stmt`	Aucune	Une seule instruction SQL préparée.	Aucun.
`sqlite3_prepare_v2 (aDatabase, SQLChar, -1, &statement, NULL)`	`int SQLite_OK` en cas de succès	Interprète l'instruction SQL.	`aDatabase` – un pointeur `SQLite3` vers la base de données. `SQLChar` – une chaîne `const char *` de caractères UTF8 qui représente l'instruction SQL.
`sqlite3_column_count (statement)`	`int`	Compte le nombre de champs dans le jeu de résultats d'une requête.	`statement` – un pointeur `sqlite3_stmt`.

Tableau 7.7 : API de SQLite3 *(suite)*

Objet/fonction	*Valeur de retour*	*Description*	*Paramètres*
`sqlite3_column_name (statement, i)`	`const char *`	Obtient le nom d'un champ.	`statement` – un pointeur `sqlite3_stmt`. `i` – un entier qui représente le numéro du champ.
`sqlite3_changes (database)`	`int`	Obtient le nombre d'enregistrements affectés par la modification d'une table.	`aDatabase` – un pointeur `SQLite3` vers la base de données.
`sqlite3_step (statement)`	`int SQLITE_ROW` lorsqu'une ligne est disponible	Déplace le curseur vers l'enregistrement suivant dans le jeu de résultats.	`statement` – un pointeur `sqlite3_stmt`.

`sqlite3_column_type (statement, i)`	`int`
	
	`Valeurs possibles :`
	
	`SQLITE_INTEGER`
	
	`SQLITE_FLOAT`
	
	`SQLITE_BLOB`
	
	`SQLITE_NULL`
	
	`SQLITE_TEXT`
		Obtient le type d'un champ (colonne) dans le jeu de résultats d'une instruction.	`statement` – un pointeur `sqlite3_stmt`.
			
			`i` – le numéro du champ dont on souhaite obtenir les données.

`sqlite3_column_int (statement, i)`	`int`	Obtient la valeur d'un champ de type entier.	`statement` – un pointeur `sqlite3_stmt`. `i` – le numéro du champ dont on souhaite obtenir les données.
`sqlite3_column_double (statement, i)`	`double`	Obtient la valeur d'un champ de type réel.	`statement` – un pointeur `sqlite3_stmt`. `i` – le numéro du champ dont on souhaite obtenir les données.

Openmirrors.com

Tableau 7.7 : API de SQLite3 *(suite)*

Objet/fonction	*Valeur de retour*	*Description*	*Paramètres*
`sqlite3_column_text (statement, i)`	`const unsigned char *`	Obtient la valeur d'un champ de type chaîne de caractères.	`statement` – un pointeur `sqlite3_stmt`. `i` – le numéro du champ dont on souhaite obtenir les données.
`sqlite3_column_blob (statement, i)`	`byte *`	Obtient les octets d'un champ de type BLOB (*binary large object*).	`statement` – un pointeur `sqlite3_stmt`. `i` – le numéro du champ dont on souhaite obtenir les données.
`sqlite3_column_bytes (statement, i)`	`int`	Obtient le nombre d'octets de la valeur d'un champ de type BLOB.	`statement` – un pointeur `sqlite3_stmt`. `i` – le numéro du champ dont on souhaite obtenir les données.
`sqlite3_finalize (statement)`	`void`	libère les ressources associées à l'instruction.	`statement` – un pointeur `sqlite3_stmt`.
`sqlite3_bind_blob (statement, parameterIndex, aVariable, byteLength, transienceKey)`	`int`	Lie un pointeur sur un tableau d'octets à un paramètre substituable d'une instruction préparée.	`statement` – un pointeur `sqlite3_stmt`. `parameterIndex` – l'indice du paramètre substituable dans l'instruction préparée. `aVariable` – un pointeur sur le tableau d'octets à enregistrer dans la base de données. `byteLength` – le nombre d'octets à enregistrer. `transienceKey` – un indicateur précisant si les données insérées doivent être copiées avant l'insertion afin d'éviter leur modification pendant l'enregistrement.

Tableau 7.7 : API de SQLite3 *(suite)*

Objet/fonction	*Valeur de retour*	*Description*	*Paramètres*
`sqlite3_bind_double (statement, parameter-Index, aVariable)`	`int`	Lie une valeur réelle à un paramètre substituable d'une instructionpréparée.	`statement` – un pointeur `sqlite3_stmt`. `parameterIndex` – l'indice du paramètre substituable dans l'instruction préparée. `aVariable` – un nombre réel à enregistrer dans la base de données.
`sqlite3_bind_int (statement, parameter-Index, aVariable)`	`int`	Lie une valeur entière à un paramètre substituable d'une instruction préparée.	`statement` – un pointeur `sqlite3_stmt`. `parameterIndex` – l'indice du paramètre substituable dans l'instruction préparée. `aVariable` – un nombre entier à enregistrer dans la base de données.

Cette bibliothèque C contient toutes les fonctions utilisées pour l'accès aux bases de données SQLite. Elle prend également en charge les transactions et les instructions préparées. Le code ci-après montre comment employer cette API avec les instructions préparées. Il provient de la méthode `dbAccess` de l'objet `SQLiteDataAccess`. Comme le stipule l'API, la fonction `sqlite3_prepare_v2` attend des pointeurs sur la base de données, sur l'instruction SQL à exécuter et sur un pointeur vers une variable `sqlite3_stmt`. En cas d'erreur au cours de l'exécution de l'instruction SQL, `sqlite3_prepare_v2` retourne un code numérique qui représente l'erreur.

```
int numResultColumns = 0;
sqlite3_stmt *statement = nil;
const char* SQLChar = [SQL UTF8String];
if (sqlite3_prepare_v2(database, SQLChar, -1,
        &statement, NULL) == SQLITE_OK) {
    ...
}
```

Openmirrors.com

Dans le code précédent, la variable `sqlite3_stmt` passée a été fixée au cours de l'exécution de la fonction `sqlite3_prepare_v2` et contient le `sqlite3_stmt` qui est ensuite utilisé pour obtenir chaque élément de données du jeu de résultats de la requête SQL exécutée. Une suite d'appels à la fonction `sqlite3_step` déplace un curseur qui désigne la position de la ligne dans le jeu des résultats. Ainsi, lorsque le jeu de résultats est vide, ou ne contient aucune ligne, `sqlite3_step` retourne non plus `SQLITE_ROW` mais `SQLITE_DONE`. Cela permet d'utiliser une instruction `while` pour extraire les données de la ligne indiquée par le curseur.

```
while (sqlite3_step(statement) == SQLITE_ROW) {
    ...
}
```

Dans cette boucle `while`, plusieurs `NSMutableArrays` sont créés ; voir la ligne 218 du fichier *SQLiteDataAccess.m* et la ligne suivante. Chacun de ces tableaux représente une ligne du jeu de résultats et se nomme donc `row`.

```
NSMutableArray *row = [[NSMutableArray alloc] initWithCapacity:numResultColumns];
```

Pour obtenir les valeurs des différents champs des enregistrements du jeu de résultats, il faut appeler les fonctions associées au type approprié. Le Tableau 7.7 recense ces fonctions et le code suivant montre comment les utiliser.

```
int type = [[[theResult columnTypes]
        objectAtIndex:i] intValue];
if(type == SQLITE_INTEGER){
    NSNumber *aNum = [[NSNumber alloc]
            initWithInt: sqlite3_column_int(statement, i)];
    [row addObject:aNum];
    [aNum autorelease];
}
else if(type == SQLITE_FLOAT){
    NSNumber *aFloat = [[NSNumber alloc]
            initWithFloat :sqlite3_column_double(statement, i)];
    [row addObject:aFloat];
    [aFloat autorelease];
}
else if(type == SQLITE_TEXT){
    NSString *aText = [[NSString alloc]
            initWithCString:sqlite3_column_text(statement, i)
            encoding:NSASCIIStringEncoding];
    [row addObject:aText];
    [aText autorelease];
}
else if(type == SQLITE_BLOB){
```

```
        NSData *aData = [[NSData alloc]
                dataWithBytes:sqlite3_column_blob(statement, i)
                length:sqlite3_column_bytes(statement,i)];
        [row addObject:aData];
        [aData autorelease];
    }
    else{//SQLITE_NULL
        [row addObject:@"null"];
    }
```

Pour appeler la fonction appropriée, le type du champ concerné doit être déterminé. Pour cela, le type du champ est examiné et enregistré avant l'appel ; voir les lignes 199 à 205 dans le fichier *SQLiteDataAccess.m* et le code suivant.

```
NSMutableArray *columnTypes = [[NSMutableArray alloc]
        initWithCapacity:0];
for(int i = 0; i < numResultColumns; i++){
    NSNumber * columnType = [NSNumber numberWithInt:
            sqlite3_column_type(statement,i)];
    [columnTypes addObject:columnType];
}
[theResult setColumnTypes:columnTypes];
```

Le type de chaque colonne est obtenu à partir du jeu de résultats de l'instruction en invoquant la fonction `sqlite3_column_type`. Le pointeur sur l'instruction et le numéro du champ concerné sont passés à la fonction, qui retourne un indicateur numérique du type de ce champ. Voici les types reconnus :

- `SQLITE_INTEGER` ;
- `SQLITE_FLOAT` ;
- `SQLITE_BLOB` ;
- `SQLITE_NULL` ;
- `SQLITE_TEXT`.

Les méthodes `dbAccess`, `setData` et `getData` retournent un pointeur sur un `DataAccessResult`. Cet objet contient les résultats de l'exécution d'une instruction SQL sur une base de données SQLite. Le code suivant est extrait du fichier *DataAccessResult.h* et montre les champs utilisés pour enregistrer les résultats d'une exécution SQL.

```
@interface DataAccessResult : NSObject {
    NSArray *fieldNames;
    NSArray *columnTypes;
    NSArray *results;
```

Openmirrors.com

```
    NSString *errorDescription;
    NSInteger rowsAffected;
    NSInteger insertedID;
}

@property (nonatomic, retain) NSArray *fieldNames;
@property (nonatomic, retain) NSArray *columnTypes;
@property (nonatomic, retain) NSArray *results;
@property (nonatomic, retain) NSString *errorDescription;
@property (nonatomic) NSInteger rowsAffected;
@property (nonatomic) NSInteger insertedID;

- (NSString*) JSONStringify;

@end
```

Le Chapitre 4 l'a expliqué, puisque le framework prend en charge le passage des données, tous les objets `DataAccessResult` générés par des appels à `SQLiteDataAccess` sont inclus dans les paramètres transmis aux objets de contrôle de l'affichage (VCO, *View Control Object*). Puisqu'un appel JavaScript est effectué pour obtenir ou fixer les données dans une base de données "native", les résultats de la requête doivent être renvoyés à l'application JavaScript. Cette opération est réalisée par l'objet `SendDBResultVCO`.

À l'instar de tous les objets de commande, `SendDBResultVCO` offre une méthode `doCommand`. Puisque les données doivent être renvoyées au code JavaScript, elles sont converties en une chaîne JSON (lignes 8 à 11 du code suivant). Chaque résultat est d'abord converti en une chaîne JSON et ajouté au tableau `retVal` de type `NSMutableArray`. Ce tableau est ensuite converti en une chaîne JSON. En raison des limitations de la bibliothèque JSON d'Objective-C, il est impossible de convertir les tableaux d'objets en une chaîne JSON en un seul appel. La bibliothèque ne parcourt pas les objets des tableaux pour effectuer les conversions appropriées. Un appel supplémentaire à `JSONStringify` est donc nécessaire pour chaque objet `DataAccessResult`.

```
1  + (id) doCommand:(NSArray*) parameters{
       ...
2      NSArray *results = [parameters subarrayWithRange:aRange];
3      int numResults = [results count];
4      NSMutableArray *retVal = [[NSMutableArray alloc] init];
5      for(int i = 0; i < numResults; i++){
6          DataAccessResult * aResult =
7                  (DataAccessResult*)[results objectAtIndex:i];
8          NSString* resultString = [aResult JSONStringify];
```

```
 9          [retVal addObject:resultString];
10      }
11      [retVal addObject:[parameters objectAtIndex:4]];
12
13      SBJSON *generator = [SBJSON alloc];
14      NSError *error;
15      NSString *dataString = [generator stringWithObject:retVal error:&error];
16      [generator release];
17      dataString = [dataString
18              stringByReplacingOccurrencesOfString:@"'" withString:@"\\'"];
19      NSString *jsString = [[NSString alloc]
20              initWithFormat:
21              @"handleRequestCompletionFromNative('%@')
22              , dataString];
23      QuickConnectViewController *controller =
24              [parameters objectAtIndex:0];
25      [controller.webView
26              stringByEvaluatingJavaScriptFromString:jsString]);
27      return nil;
28  }
```

Comme au Chapitre 4, où les résultats étaient convertis en une chaîne JSON, un appel permet de les passer à la partie JavaScript de l'application en vue de leur traitement. Il s'agit de l'appel à `stringByEvaluatingJavaScriptFromString` vu précédemment. Il exécute la fonction JavaScript `handleRequestCompletionFromNative` pour s'assurer que tous les autres BCO et VCO associés à la commande d'origine soient exécutés. Pour de plus amples informations concernant ce fonctionnement, consultez le Chapitre 5.

En résumé

Le module JavaScript `DataAccessObject` facilite énormément les interactions avec les bases de données SQLite prises en charge par le moteur WebKit. Puisque ce module accepte uniquement les types JavaScript standard, comme les chaînes de caractères et les tableaux, il est plus facile à employer que les fonctions JavaScript SQLite de WebKit. `DataAccessObject` est également une façade utilisée pour accéder aux fichiers de base de données "natives" que vous pouvez livrer avec votre application. Avec des appels JavaScript, il est possible d'accéder aux données présentes dans la base. De cette manière, l'application JavaScript est une application complète qui offre les mêmes fonctionnalités que l'application équivalente écrite en Objective-C.

À l'instar du `DataAccessObject` JavaScript, la classe Objective-C `SQLiteDataAccess` facilite l'extraction ou l'enregistrement des données dans des bases SQLite livrées avec

Openmirrors.com

l'application. Puisqu'elle suit la même conception que la classe JavaScript `DataAccessObject`, il est plus facile de comprendre son rôle, même sans maîtriser Objective-C.

Puisque tous les résultats des requêtes effectuées sur les bases de données du moteur WebKit ou les bases de données "natives" sont des objets qui contiennent des types JavaScript standard, il est inutile de connaître les détails internes du fonctionnement de SQLite avec WebKit ou en natif.

`DataAccessObject` et `SQLiteDataAccess` se fondent sur les transactions pour éviter que les appels asynchrones ne perturbent les requêtes. Vous pouvez effectuer plusieurs appels concurrents à la base de données sans vous inquiéter d'éventuelles perturbations sur les données dans la base. Le chapitre suivant explique comment réaliser une enveloppe pour les appels AJAX qui ressemble et se comporte de manière comparable à `DataAccessObject`.

Openmirrors.com



8

Données distantes

Votre application peut avoir besoin d'accéder à des données qui se trouvent dans une base de données distante ou qui sont fournies par des services web. Vous pourriez même vouloir synchroniser les données du téléphone avec des données enregistrées à distance.

Les applications hybrides pour l'iPhone facilitent la mise en œuvre de ces comportements. Puisqu'elles sont hybrides, elles disposent d'un accès total à l'objet `XMLHttpRequest` depuis le code JavaScript. Ce chapitre explique comment obtenir des données à partir d'un flux RSS et les afficher dans votre application.

Comme pour l'accès aux bases de données examiné au Chapitre 7, une enveloppe simple d'emploi pour l'objet `XMLHttpRequest` est décrite dans la première section. Les sections suivantes expliquent comment cette enveloppe a été créée et son fonctionnement interne.

Section 1 : application browserAJAXAccess

Au Chapitre 7, l'application nativeDBAccess a illustré l'enregistrement et la récupération des données à partir d'une base de données SQLite sur un appareil. Ce chapitre décrit une application semblable qui permet d'interagir avec des services et des serveurs web pour obtenir des données. La Figure 8.1 montre l'application browserAJAXAccess en cours d'exécution.



Figure 8.1
L'application browserAJAXAccess affiche le contenu du blog de TetonTech.

Tous les blogs WordPress proposent un flux RSS qui fournit les dix derniers billets du blog (voir Figure 8.2). Bien que ce flux ne semble pas proposer l'ensemble des billets du blog, en réalité ce n'est pas le cas. L'application browserAJAXAccess affiche uniquement les intitulés, mais il est très facile de l'étendre pour enregistrer les intitulés et les billets à la manière décrite au Chapitre 7.

Heureusement, les flux RSS se moquent du client. Un flux reçoit une demande pour les billets du blog et les envoie au format XML, quel que soit le demandeur. Puisque `UIWebView` intègre le moteur WebKit, il peut envoyer des requêtes au flux et interpréter le contenu XML reçu. Pour cela, l'outil utilisé prend la forme de l'objet `XMLHttpRequest` et la méthode employée se nomme AJAX.

Openmirrors.com

Figure 8.2
Le flux RSS de TetonTech affiché dans Safari.



AJAX n'a rien de grec

L'*Iliade* est l'un des plus grands textes de tous les temps. Cette épopée d'Homère raconte la guerre entres les Grecs, ou Achéens, et les Troyens.

L'un des héros grecs se nomme Ajax. Il bat constamment ses adversaires et, lors d'un épisode, il sauve à lui seul la flotte grecque de la destruction.

À l'instar de la flotte grecque, le développement traditionnel des pages web a subi et subit des attaques. On lui reproche d'être trop lent, trop difficile à employer et pas assez flexible. Une fois encore, AJAX vient à la rescousse, mais il ne s'agit plus du héros grec. Ce nouvel AJAX signifie *Asynchronous JavaScript and XML*.

L'idée derrière AJAX est simple : améliorer l'expérience utilisateur en ne rechargeant pas les pages à chaque requête. À la place, des données sont envoyées ou obtenues, et les résultats sont présentés en employant des techniques issues du HTML dynamique. Tout cela peut être réalisé dans une page à l'aide de JavaScript.

En conjuguant l'objet `XMLHttpRequest` et du code JavaScript simple pour manipuler une page web, votre application hybride pour l'iPhone peut exploiter des données distantes comme si elles étaient locales. Vous obtenez alors le meilleur des deux mondes : l'application

peut s'exécuter en mode autonome, s'exécuter en mode réseau et synchroniser les données lorsqu'une connexion est disponible. Le framework QuickConnectiPhone fournit une enveloppe AJAX simple d'emploi : `ServerAccessObject`.

Section 2 : utilisation de *ServerAccessObject*

L'enveloppe AJAX `ServerAccessObject` vous permet d'accéder facilement à des données distantes sans connaître les détails de l'API de `XMLHttpRequest`. L'API de `ServerAccessObject` est quasiment identique à celle de `DataAccessObject` examinée au Chapitre 7. Elle comprend un constructeur et deux méthodes. Le constructeur enregistre l'URL du serveur distant et configure les méthodes de l'objet. Les deux méthodes, `getData` et `setData`, permettent ensuite d'obtenir et d'envoyer des données au serveur distant défini dans le constructeur. Le Tableau 8.1 décrit l'API de `ServerAccessObject`.

Tableau 8.1 : API de `ServerAccessObject`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`ServerAccessObject (URL)`	`ServerAccessObject`	Crée un `ServerAccessObject` lorsqu'elle est appelée avec le mot clé `new`.	`URL` – l'URL du serveur à contacter.
`getData(dataType, refresh, parameterSequence, HTTPHeaders)`	`void`	Cette méthode permet d'obtenir des informations depuis un serveur distant. La requête est de type `GET`. Cette méthode est sûre vis-à-vis des threads.	`dataType` – le type des données à obtenir : `ServerAccessObject.XML` ou `ServerAccessObject.TEXT`. `refresh` – un booléen qui indique si une mise à jour forcée des données depuis le serveur doit être effectuée. `parameterSequence` – les paramètres à ajouter à l'URL. Le point d'interrogation initial (`?`) n'est pas inclus. `HTTPHeaders` – un tableau associatif contenant les noms et les valeurs des entêtes à envoyer avec la requête.

Openmirrors.com

Tableau 8.1 : API de `ServerAccessObject` *(suite)*

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`setData(dataType, parameterSequence, data, HTTPHeaders)`	`void`	Cette méthode est utilisée pour modifier ou créer des informations sur un serveur distant, ou pour passer des types de paramètres sécurisés. La requête est de type `POST`. Cette méthode est sûre vis-à-vis des threads.	`dataType` – le type des données à obtenir : `ServerAccessObject.XML` ou `ServerAccessObject.TEXT`. `parameterSequence` – les paramètres à ajouter à l'URL. Le point d'interrogation initial (`?`) n'est pas inclus. `data` – les données à inclure dans l'envoi. Il peut s'agir d'une grande quantité de caractères, d'un fichier, etc. `HTTPHeaders` – un tableau associatif contenant les noms et les valeurs des entêtes à envoyer avec la requête.

Le fichier *ServerAccessObject.js*, qui se trouve dans le groupe `QCiPhone` des modèles Dashcode et Xcode, contient l'enveloppe `ServerAccessObject`. Il est automatiquement inclus dans le fichier *index.html* de l'application par les deux modèles.

La fonction `getSiteDataBCF`, définie dans le fichier *functions.js*, utilise `ServerAccessObject`. Ce fichier JavaScript contient toutes les fonctions de contrôle pour l'application browserAJAXAccess. Pour de plus amples informations concernant ces types de fonctions et la manière de les créer, consultez le Chapitre 2.

L'objectif de la fonction `getSiteDataBCF` est d'obtenir les billets à partir du blog de l'auteur. Ce comportement est facile à mettre en œuvre à l'aide de l'objet `ServerAccessObject`, comme le montrent les lignes 2 à 5 du code suivant :

```
1 function getSiteDataBCF(parameters){
2     var site = new ServerAccessObject(
3             'http://tetontech.wordpress.com/feed/');
4     site.getData(ServerAccessObject.XML,
```

```
5             ServerAccessObject.REFRESH);
6     // Puisque les appels AJAX sont asynchrones,
7     // cette BCF ne retourne aucune valeur.
8
9 }
```

Les lignes 2 et 3 construisent le `ServerAccessObject`. Il reçoit l'URL du flux RSS, dans ce cas celle du blog de l'auteur (**http://tetontech.wordpress.com/feed**). Pour accéder à un autre blog WordPress, vous devez simplement remplacer cette URL.

Les lignes 4 et 5 utilisent ensuite ce nouvel objet nommé `site` pour obtenir les données. Puisqu'il s'agit d'un flux RSS et que le serveur renvoie des données XML à l'application browserAJAXAccess, on indique que les données sont de type XML. Le second paramètre force une actualisation des données, au lieu qu'elles soient prises dans un cache. Ce fonctionnement est indispensable si vous souhaitez garantir que les données reçues contiennent toutes les modifications enregistrées sur le serveur.

Une utilisation inconsidérée de l'actualisation peut conduire à des effets secondaires négatifs. Si l'actualisation est demandée alors qu'elle n'est pas requise, le serveur et le réseau peuvent se trouver surchargés dans le cas où l'application devient populaire. C'est pourquoi il faut déterminer si les toutes dernières données doivent être obtenues ou si elles peuvent être légèrement obsolètes.

À l'instar de `DataAccessObject`, tous les appels à `ServerAccessObject` sont asynchrones. Plusieurs appels peuvent être effectués en parallèle, tant qu'un nouveau `Server-Access-Object` est créé pour chacun d'eux. Avec le framework QuickConnectiPhone, il est inutile de définir une fonction de rappel comme cela est requis lorsqu'on utilise AJAX. Le framework s'assure de l'exécution des fonctions de contrôle métier (BCF, *Business Control Function*) et des fonctions de contrôle de l'affichage (VCF, *View Control Function*) associées à la même commande que la BCF qui invoque `ServerAccessObject`.

Dans le fichier *mappings.js*, les fonctions `getSiteDataBCF` et `displaySiteDataVCF` sont associées à la commande `sampleQuery` (voir Chapitre 2). Autrement dit, les données demandées par `getSiteDataBCF` sont certaines d'être passées à `displaySiteDataVCF` par le framework.

Le code de `displaySiteDataVCF` est donné ci-après. Le paramètre `results` passé à cette fonction est un tableau d'objets JavaScript, un pour chaque BCF associée à la commande `sampleQuery`. La méthode `getData` de `ServerAccessObject` conduit à la création d'un objet `QueryResult`, comme c'était le cas dans la méthode `getData` de `DataAccess-Object` ; pour de plus amples informations concernant l'objet `QueryResult`, consultez le Chapitre 7.

Openmirrors.com

Après avoir effacé les éléments affichés et représentés par la variable `container` dans le code suivant, l'objet `QueryResult` contenant les résultats de l'appel AJAX est obtenu (ligne 9). Puisque `getSiteDataBCF` est la première BCF associée à la commande `sampleQuery`, les résultats générés par son invocation se trouvent dans le premier objet du tableau `results` passé en paramètre.

Nous l'avons vu au Chapitre 7, les objets `QueryResult` possèdent un attribut nommé `data`. Dans le cas des requêtes XML, comme celle effectuée dans `getSiteDataBCF`, cet attribut contient le document XML résultant.

Puisque ce document est comparable à un document utilisé habituellement avec du contenu HTML dynamique, il est traité de manière semblable. Les mêmes types de méthodes sont disponibles, comme `getElementById` et `getElementsByTagName`. Il est également constitué d'objets `Node`, avec des relations de parent, d'enfants et de frères. Vous pouvez employer toutes les méthodes et solutions classiques pour interpréter les données.

La fonction utilitaire `parseWordPressFeed`, définie dans le fichier *RSSUtilities.js*, est appelée à la ligne 9. Elle emploie la méthode standard pour obtenir un tableau en deux dimensions, nommé `entries`, qui contient les billets de blog. Chaque entrée du tableau comprend la date de publication, le contenu et l'intitulé du billet.

```
1 function displaySiteDataVCF(results, parameters){
2     var container =
3             document.getElementById('queryResults');
4     // Effacer le conteneur.
5     while(container.lastChild){
6         container.removeChild(container.lastChild);
7     }
8     // Utiliser un parseur wordpress pour créer les objets des billets.
9     var entries = parseWordPressFeed(results[0].data);
10    var numEntries = entries.length;
11    // Pour chaque billet, ajouter l'intitulé et la date
12    // dans le <div> conteneur.
13    for (var i = numEntries-1; i >= 0; i--){
14        var entry = entries[i];
15        var publishDate = entry.date;
16        var title = entry.title;
17
18        var titleElement = document.createElement('h2');
19        titleElement.innerText = entry.title;
20        container.appendChild(titleElement);
21
22        var dateElement = document.createElement('h3');
```

```
23          dateElement.innerText = entry.date;
24          container.appendChild(dateElement);
25
26          var hardRule = document.createElement('hr');
27          container.appendChild(hardRule);
28      }
29 }
```

Pour chaque entrée générée par la fonction `parseWordPressFeed`, les lignes 13 à 28 créent des objets HTML `Element` et les insèrent dans le conteneur. L'intitulé et la date de chaque billet de blog sont ainsi affichés. Une ligne est ensuite ajoutée pour séparer chaque billet affiché.

Cet exemple montre comment gérer les flux RSS, mais d'autres types d'accès sont aussi simples, si ce n'est faciles. Vous pouvez effectuer une requête de type `TEXT` et obtenir le contenu HTML à insérer dans l'interface utilisateur de votre application. Toutefois, cette approche est déconseillée pour des questions de sécurité. Vous pouvez également effectuer un appel de type `TEXT` pour obtenir des données au format JSON.

JSON n'a rien de grec, lui non plus

JSON se prononce "Jaison", comme le Jason (avec l'accent anglais) avec ses Argonautes. Il est intéressant de remarquer que le long voyage de Jason et de ses compagnons avait pour objectif de ramener la Toison d'or.

JSON (*JavaScript Object Notation*) est utilisé pour faire voyager les objets JavaScript sur de longues distances dans les réseaux. Le code suivant montre comment créer un objet JavaScript qui enregistre un nom et un prénom :

```
var unObjet = new Object();
unObjet.prenom = 'Paul';
unObjet.nom = 'Martin';
```

Le code suivant illustre une autre syntaxe :

```
var unObjet = {prenom:'Paul', nom:'Martin'};
```

Lequel est correct ? Les deux réalisent la même chose. La seconde solution est probablement plus rapide, mais, lorsque les objets sont grands, la première est plus lisible et facile à modifier.

La seconde syntaxe a pour avantage de pouvoir être passée sous forme d'une chaîne de caractères :

```
"{prenom:'Paul', nom:'Martin'}"
```

Openmirrors.com

Cette chaîne peut ensuite être convertie en un objet en la passant à la fonction JavaScript standard `eval`, mais cette méthode est dangereuse. Pour une solution plus sûre et simple d'emploi, consultez l'Annexe A.

Est-ce une coïncidence si, tels les deux héros grecs, AJAX et JSON sont là pour sauver le développement de type web ? À vous de décider.

Quel que soit le type des données à obtenir, `ServerAccessObject` facilite cette opération. Il suffit simplement d'instancier cet objet et d'appeler `getData`, pour une requête de type `GET`, ou `setData`, pour une requête de type `POST`. Dans tous les cas, le framework QuickConnectiPhone passe les informations aux autres objets de contrôle associés à la commande.

Section 3 : *ServerAccessObject*

La section précédente a montré comment l'utilisation de `ServerAccessObject` permettait d'exploiter les possibilités d'AJAX et de l'API de `XMLHttpRequest` sans avoir besoin de maîtriser ces derniers. Cette section s'intéresse à ces deux aspects et montre comment les utiliser. Si vous souhaitez simplement vous servir de `ServerAccessObject` sans en comprendre le fonctionnement, vous pouvez sauter cette section.

Grâce à `ServerAccessObject`, défini dans le fichier *ServerAccessObject.js* du groupe `QCiPhone`, le programmeur n'a pas besoin d'une grande connaissance d'AJAX pour en exploiter les possibilités. Cette classe propose des méthodes et des constructeurs semblables à ceux de l'enveloppe JavaScript `DataAccessObject` décrite au Chapitre 7. Puisqu'il dispose d'une API simplifiée, le programmeur qui n'est pas familier d'AJAX peut envoyer et obtenir des données distantes sans passer par une longue phase d'apprentissage. Si vous connaissez l'API de l'un de ces objets d'accès, vous pouvez employer l'autre.

Le constructeur de `ServerAccessObject` est la plus simple de ses méthodes. Il enregistre l'URL du serveur distant et définit ensuite les méthodes de l'objet. La ligne suivante, extraite du constructeur, montre que l'URL est enregistrée dans l'attribut `URL` de l'objet en vue d'une utilisation ultérieure.

```
this.URL = URL;
```

Outre cet attribut `URL`, auquel le programmeur n'accède jamais directement, il faut également prendre en compte la méthode privée `makeCall`.

`makeCall` est au cœur de `ServerAccessObject`. Elle effectue tout le travail requis par les accès au serveur. Elle est invoquée par les deux méthodes de façade `getData` et `setData`. Le Tableau 8.2 décrit son API et son utilisation de base.

Comme vous le verrez par la suite, l'une des nombreuses solutions standard pour affecter des méthodes à des objets est employée pour ces façades. Elle consiste à créer un objet de fonction à l'aide du constructeur `function` et à affecter le résultat à un attribut de l'objet courant représenté par le mot clé `this`.

Les deux méthodes de façade `getData` et `setData` sont quasiment identiques. Elles prennent quatre arguments et les passent, ainsi que trois autres, à la méthode `makeCall`. Les paramètres supplémentaires sont le premier, le cinquième et le septième dans l'appel à `getData` et le premier, le troisième et le septième dans l'appel à `setData`. Le cinquième paramètre de la méthode `makeCall` précise les données qui doivent être passées au serveur à l'aide d'une requête de type `POST`. Il est évidemment inutile pour la méthode `getData` et est donc remplacé par `null`.

```
this.getData = function(dataType, refresh,
        parameterSequence, HTTPHeaders){

    var passThroughParameters = generatePassThroughParameters();
    this.makeCall('GET', dataType, refresh,
            parameterSequence, null, HTTPHeaders, passThroughParameters);
}

this.setData = function(dataType,
        parameterSequence, data, HTTPHeaders){

    var passThroughParameters = generatePassThroughParameters();
    this.makeCall('POST', dataType, true,
            parameterSequence, data, HTTPHeaders, passThroughParameters);
}
```

L'appel à la fonction `generatePassThroughParameters` assemble toutes les informations qui permettent au framework de poursuivre le traitement des BCF et VCF associées à la BCF en utilisant `ServerAccessObject`. Pour de plus amples informations concernant cette fonction qui se trouve dans le fichier *QCUtilities.js*, consultez le Chapitre 5.

Le troisième paramètre de la méthode `makeCall` est un indicateur booléen qui précise si le cache du client, dans ce cas le `UIWebView`, doit être utilisé ou non. Pour la méthode `setData`, la mise en cache étant évidemment une mauvaise idée, la valeur de ce paramètre est figée à `true` de manière à désactiver le cache.

En incluant ces paramètres dans la signature de la méthode `makeCall`, elle peut encapsuler efficacement l'obtention et l'envoi des données à un serveur distant. Ce principe de fonction utilitaire servant de façade est souvent employé lorsque, comme dans notre cas, une

Openmirrors.com

grande partie du code de deux fonctions/méthodes ou plus est presque identique et que, sans la façade, le code exhiberait une redondance importante.

Tableau 8.2 : API de `makeCall`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`makeCall(callType, dataType, refresh, parameterSequence, data, HTTPHeaders, passThroughParameters)`	`void`	Cette méthode doit être considérée comme privée. Elle effectue les appels AJAX au serveur et prend en charge les résultats.	`callType` – `GET` ou `POST`. `dataType` – `TEXT` ou `XML`. `refresh` – un booléen qui indique si une mise à jour forcée des données depuis le serveur doit être effectuée. `parameterSequence` – une chaîne qui contient les paramètres de la requête ajoutés à l'URL. `data` – les données textuelles ou binaires envoyées avec la requête. Ce paramètre est utilisé avec les requêtes `POST`. `HTTPHeaders` – un tableau associatif contenant les noms et les valeurs des en-têtes à envoyer avec la requête. `passThroughParameters` – un tableau des valeurs requises par le framework pour poursuivre le traitement après la réception des données depuis le serveur.

Pour comprendre la méthode `makeCall`, il est nécessaire de comprendre l'API de l'objet JavaScript `XMLHttpRequest` sous-jacent. Cette API est mise en œuvre par la classe `UIWebView` utilisée dans les applications hybrides et la version mobile de Safari.

Le seul objet défini par cette API est `XMLHttpRequest` (voir Tableau 8.3).

Tableau 8.3 : API de `XMLHttpRequest`

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`XMLHttpRequest()`	`XMLHttpRequest`	Le constructeur de l'objet.	Aucun.
`abort()`	`void`	Termine la requête.	Aucun.
`getAllResponseHeaders()`	`String`	Cette méthode retourne une chaîne qui contient les noms et les valeurs de tous les en-têtes de la réponse.	Aucun.
`getResponseHeader(aHeaderName)`	`String`	Cette méthode retourne une chaîne qui contient la valeur de l'en-tête de nom indiqué ou `null` si cet en-tête n'existe pas.	`aHeaderName` – le nom de l'en-tête HTTP de la réponse dont la valeur est recherchée.
`open(type, URL, asynch, userName, password)`	`void`	Ouvre et prépare une connexion au serveur.	`type` – une chaîne contenant la valeur `GET` ou `POST`. `asynch` – un booléen qui précise si la requête doit être asynchrone. Ce paramètre doit toujours avoir la valeur `true`. `userName` – un paramètre facultatif qui précise le nom d'utilisateur permettant d'accéder au fichier ou au répertoire indiqué dans l'URL. `password` – un paramètre facultatif qui précise le mot de passe de l'utilisateur indiqué pour accéder au fichier ou au répertoire indiqué dans l'URL.
`send(data)`	`void`	Cette méthode associe des données textuelles ou binaires à une requête. Elle est utilisée dans les requêtes de type `POST`, par exemple pour l'envoi de fichiers.	`data` – les informations associées à la requête.

Openmirrors.com

Tableau 8.3 : API de `XMLHttpRequest` *(suite)*

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`SetRequestHeader(name, value)`	`void`	Une méthode qui peut redéfinir les valeurs des en-têtes standard ou ajouter des en-têtes personnalisés avec leur valeur à une requête.	`name` – une chaîne qui représente l'identifiant de l'en-tête. `value` – une chaîne qui contient la valeur associée au nom.
`onreadystatechange`		Un attribut auquel est affecté une fonction. Cette fonction est invoquée lorsque l'événement `onreadystatechange` est déclenché, c'est-à-dire à chaque modification de `readyState`.	
`readyState`		Un entier qui représente l'état de la requête effectuée. Voici les valeurs possibles : `0` – la méthode `send` n'a pas été invoquée. `1` – La requête est en cours d'envoi au serveur. `2` – La requête a été reçue par le serveur. `3` – Une partie de la réponse envoyée par le serveur a été reçue. `4` – L'intégralité de la réponse envoyée par le serveur a été reçue.	
`responseText`		Les données envoyées par le serveur sous forme de texte, sans les en-têtes HTTP de la réponse.	
`responseXML`		Les données envoyées par le serveur au format DOM XML. Si les données ne sont pas du XML valide, cet attribut est `null`.	

Tableau 8.3 : API de `XMLHttpRequest` *(suite)*

Attribut/méthode	*Valeur de retour*	*Description*	*Paramètres*
`status`		Un nombre envoyé par le serveur pour indiquer le succès ou l'échec d'une requête. Les plus fréquents sont 404 (non trouvé) et 200 (succès). La liste complète est disponible à l'adresse **http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html**.	
`statusText`		Une chaîne générée par le serveur qui contient un message correspondant au code d'état.	

De cette API, les méthodes les plus employées sont le constructeur, `open` et `send`. Le code suivant donne un exemple simple d'utilisation de ces méthodes et d'autres attributs. Il demande la page principale du projet open-source WebKit sous forme textuelle.

Deux points sont à remarquer dans cet exemple. Tout d'abord, l'objet `request` est global. Il peut être utilisé dans la fonction `handleResponse`, qui est appelée automatiquement par le moteur du navigateur lorsque `readyState` change. Le code est ainsi plus simple, mais cela risque de poser un problème si deux requêtes sont émises en parallèle.

```
var request = new XMLHttpRequest();
request.onreadystatechange = handleResponse;
request.open('GET','http://webkit.org/', true);
request.sent('');

function handleResponse(){
    if(request.readyState == 4){
        if(request.status == 200){
            var result = response.responseText;
            // Utiliser result.
        }
    }
}
```

En raison de la portée globale de la variable `request`, cet exemple simple n'est pas sûr vis-à-vis des threads. Puisque les requêtes sont asynchrones, il est possible, et fort probable, que des requêtes se perturbent. `ServerAccessObject` encapsule cette variable globale de manière à résoudre ce problème.

Openmirrors.com

Le deuxième point concerne l'envoi de la requête à une URL complète. Dans un navigateur, un objet `XMLHttpRequest` peut demander des données uniquement au serveur dont il provient. L'objet `UIWebView` utilisé dans les applications hybrides ne souffre pas de cette restriction. Il peut demander des données à n'importe quel serveur car il n'est pas un navigateur. Cela représente à la fois une aubaine et un fléau.

Les navigateurs sont restreints sur ce point pour éviter les attaques XSS (*cross-site scripting*). Elles peuvent se produire lorsque du JavaScript malveillant est inséré dans du contenu HTML sinon innocent demandé par votre application. Puisque `UIWebView` n'est pas un navigateur, vous êtes responsable de la protection de votre application contre ces attaques. Par chance, le framework QuickConnectiPhone permet d'associer des fonctions de contrôle de la sécurité (SCF, *Security Control Function*), qui sont invoquées par `ServerAccessObject` avant que les résultats des requêtes ne soient passés aux VCF.

La création de ces SCF est comparable à celle des VCF, et leur association se fait avec `mapCommandToSCF`. La Section 4 donnera un exemple de création et d'utilisation des SCF.

L'API de l'objet `XMLHttpRequest` ne permet pas de forcer une actualisation. `ServerAccessObject` offre cette possibilité en utilisant l'un des attributs HTTP standard de la requête.

```
if(refresh){
    /*
     * Si le cache doit être désactivé et l'appel au serveur imposé,
     * l'en-tête 'If-Modified-Since' doit être fixé à une date passée.
     */
    http.setRequestHeader( "If-Modified-Since", "Sat, 1 Jan 2000 00:00:00 GMT" );
}
```

L'en-tête `If-Modified-Since` indique au serveur qu'il doit envoyer les données si la date de modification de l'élément demandé est postérieure à celle précisée par cet en-tête. En fixant la valeur de l'en-tête à une date passée, nous sommes certains que les données dans le cache ne sont pas utilisées.

Par ailleurs, l'API de `ServerAccessObject` ne permet pas à l'utilisateur d'indiquer si la requête doit être synchrone ou asynchrone. Tous les appels AJAX doivent être asynchrones. De cette manière, le moteur web reste réactif et peut traiter les actions suivantes de l'utilisateur. Si les appels sont synchrones et si l'utilisateur pivote l'iPhone, le `UIWebView` affiche un écran blanc. Cela se produit également si l'utilisateur décide de faire défiler la vue pendant qu'une requête au serveur est en cours. Dans les deux cas, l'expérience de l'utilisateur ne sera pas agréable. L'utilisation synchrone de l'objet `XMLHttpRequest` étant donc une mauvaise idée, `ServerAccessObject` impose un traitement asynchrone de toutes les requêtes.

Contrairement à l'exemple simple précédent, `ServerAccessObject` n'utilise pas une fonction standard pour gérer les événements `onreadystatechange`. À la place, il s'appuie sur une fonction anonyme ; pour de plus amples informations concernant les fonctions anonymes, consultez le Chapitre 3. La décision d'opter pour une fonction anonyme a été prise en raison de sa capacité à exister dans la portée de la fonction englobante.

Toutes les variables locales déclarées dans la méthode `makeCall` sont également disponibles à la fonction anonyme `onreadystatechange`. De cette manière, le problème de la variable globale mentionnée précédemment est résolu. En affectant à la variable `http` le nouvel objet `XMLHttpRequestObject` créé, qui sera utilisé dans la méthode `makeCall`, il se trouve automatiquement dans la portée au moment de l'appel à la fonction anonyme `onreadystatechange`.

Ceux qui ne sont pas habitués à la notion de fonction anonyme risquent de trouver cela incongru. Ceux qui les manipulent déjà y verront un moyen de réaliser quelque chose dont ils n'étaient pas capables. Le code suivant contient l'intégralité de cette fonction anonyme.

```
http.onreadystatechange = function(){

    if(http.readyState == ServerAccessObject.COMPLETE){
        // Le représentant standard de tous les types de requête.
        var queryResult = new QueryResult();
        // Les en-têtes d'erreurs personnalisés que vous pouvez
        // envoyer depuis le code du serveur si vous le décidez.
        queryResult.errorNumber =
                http.getResponseHeader('QC-Error-Number');
        queryResult.errorMessage =
                http.getResponseHeader('QC-Error-Message');
        if(http.status != ServerAccessObject.HTTP_OK
                && http.status != ServerAccessObject.HTTP_LOCAL
                && http.status !=
                        ServerAccessObject.OSX_HTTP_File_Access){

            queryResult.errorNumber = http.status;
            queryResult.errorMessage = "Type d'accès erroné.";
        }

        /*
         * Obtenir les données si le serveur indique que
         * le traitement de la requête a réussi ou si
         * la requête concerne directement un fichier
         * sur le disque du serveur.
         * Les obtenir sous forme de texte ou en XML.
         */
        if(queryResult.errorNumber == null){
```

Openmirrors.com

```
29              queryResult.data = http['response'+dataType];
30              if(!dispatchToSCF(passThroughParameters[0],
31                      queryResult.data)){
32                  queryResult.errorNumber =
33                          ServerAccessObject.INSECURE_DATA_RECEIVED;
34                  queryResult.errorMessage =
35                          "Données non fiables reçues.";
36              }
37          }
38          /*
39           * Invoquer la prochaine fonction de contrôle
40           * de la liste en passant resultData.
41           */
42          if(window.callFunc){
43              /*
44               * L'appel peut être effectué depuis l'extérieur
45               * d'une fonction dispatchToBCF. Dans ce cas,
46               * il n'y a pas de fonction callFunc
47               * définie.
48               */
49              var theResults = new Array();
50              theResults.push(queryResult);
51              theResults.push(passThroughParameters);
52              requestHandler(passThroughParameters[0],
53              passThroughParameters[2], theResults);
54          }
55      }
56
57  };
```

Les lignes 30 à 36 du code contiennent les appels aux SCF et aux VCF mentionnées précédemment. La ligne 30 est comparable au code du contrôleur frontal décrit au Chapitre 2. En réalité, elle est pratiquement identique à la fonction `checkValidation` décrite et se comporte de manière équivalente.

Tout comme un utilisateur peut saisir des données erronées, un serveur peut envoyer des données corrompues. À l'instar de `checkValidation`, `checkSecurity` est une fonction de type contrôleur d'application. Elle invoque les SCF qui ont été associées dans le fichier *mapping.js* à la commande qui est associée à la BCF en utilisant `ServerAccessObject`. Vous pouvez ainsi appliquer tous les contrôles de sécurité que vous souhaitez aux données reçues depuis un serveur et contrer plus facilement les attaques XSS.

Après avoir vérifié les données, elles sont ajoutées à un objet `QueryResult` pour que l'application puisse poursuivre leur traitement. Cela passe par un appel à la fonction `requestHandler`. `passThroughParameters` est utilisé ici car il contient la commande qui

a déclenché l'appel à la BCF et la fonction de contrôle doit être invoquée ensuite. En appelant `requestHandler`, la méthode `makeCall` de `ServerAccessObject` garantit que toutes les fonctions de contrôle sont invoquées dans l'ordre où elles ont été associées dans le fichier *mappings.js*. Puisque `theResults` est également passé, il est disponible à toutes les autres fonctions de contrôle. Autrement dit, vous pouvez utiliser les données comme bon vous semble.

Puisque la fonction anonyme `onreadystatechange` comprend ces deux appels à des fonctions de type contrôleur d'application et puisqu'il s'agit de la seule fonction à laquelle un serveur envoie des données, elle joue le rôle d'un contrôleur frontal supplémentaire pour l'application. Autrement dit, en utilisant `ServerAccessObject` pour tous les accès à des données distantes, vous bénéficiez des avantages des communications distantes examinées au Chapitre 2 pour les applications standard.

Ce modèle de double contrôleur frontal apporte la sécurité nécessaire aux applications, tout en donnant la possibilité d'obtenir des données depuis n'importe quel serveur. La Figure 8.3 montre comment ces contrôleurs frontaux protègent votre application et vos données.

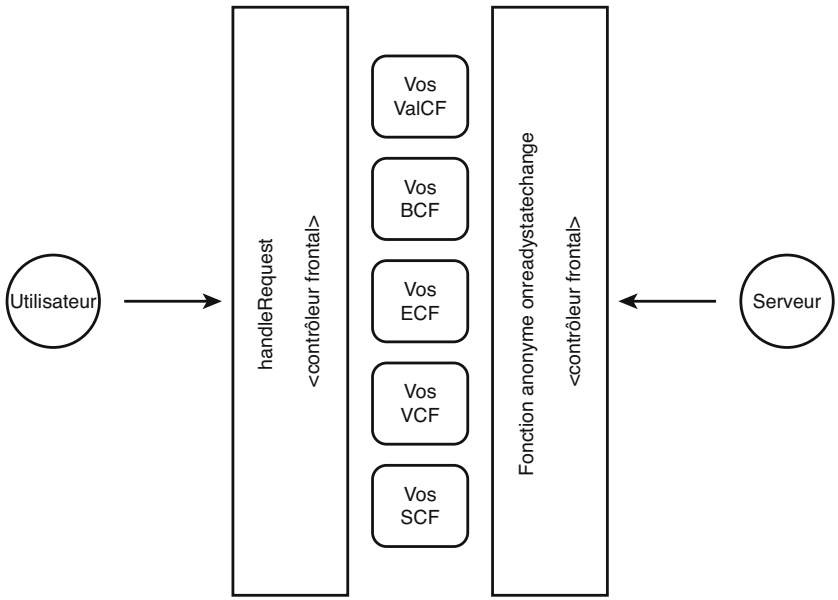


Figure 8.3
Les utilisateurs et les serveurs peuvent fournir des données erronées à l'application. Le modèle du double contrôleur frontal protège le code de l'application des données potentiellement néfastes.

Openmirrors.com

Section 4 : fonctions de contrôle de la sécurité

Les SCF sont invoquées par `ServerAccessObject` pour vérifier la validité des données obtenues depuis un serveur. Elles jouent un rôle équivalant aux ValCF décrites au Chapitre 2. Elles suivent également le même schéma que toutes les autres fonctions de contrôle.

Pour qu'une SCF puisse être invoquée, elle doit être associée à une commande. Par exemple, l'application peut imposer à l'utilisateur d'ouvrir une session afin qu'elle renvoie les données JSON en cas de succès. Voici un exemple de ce type d'association :

```
mapCommandToSCF('login', checkForFunctions);
```

Cela signifie que vous devez également disposer d'une fonction `checkForFunctions` qui peut être appelée par `checkSecurity`.

Puisque la bibliothèque json2 est disponible dans le fichier *json2.js* du groupe `QCiPhone`, cette fonction est simple à écrire.

```
function checkForFunctions(data){
    if(data == JSON.stringify(JSON.parse(data){
        return true;
    }
    return false;
}
```

Au cours de l'analyse des données, la méthode `JSON.parse` ajoute des caractères supplémentaires dans la chaîne si elle rencontre des déclarations ou des appels de fonctions. Cela permet de faire échouer ces définitions ou ces appels de fonctions, apportant ainsi une sécurité supplémentaire contre les attaques XSS.

Vos vérifications peuvent exploiter cela en reconvertissant le nouvel objet JavaScript créé en une chaîne et en la comparant à celle d'origine. Si la comparaison échoue, l'application sait que les données obtenues du serveur contiennent du code malveillant.

Le texte JSON qui vous est envoyé ne peut donc pas contenir les définitions ou des appels de fonctions. Ce point est très intéressant. En effet, si ce n'était pas le cas, vous ne pourriez jamais dire si les fonctions JavaScript contenues dans les données JSON sont de votre fait ou si du code malveillant a été inséré au cours d'une attaque XSS.

En résumé

`ServerAccessObject` constitue une solution facile et sécurisée pour obtenir des données à partir de sources distantes dans une application hybride. Cette classe apporte une API simple d'emploi, comparable à celle de `DataAccess`, qui réduit votre temps d'apprentissage.

Grâce aux SCF, le code obtenu peut être vérifié soigneusement avant que vous ne le déclariez valide.

`ServerAccessObject` peut obtenir des données distantes et les enregistrer par l'intermédiaire de `DataAccessObject` en vue de leur utilisation ultérieure, ou les utiliser directement comme dans le cas de l'application browserAJAXAccess. Il ouvre également des possibilités de synchronisation entre les données enregistrées sur la machine locale et celles présentes sur une machine distante.

`ServerAccessObject` permet de créer du code qui peut signaler automatiquement à un serveur web l'échec du code de l'application. Vous pouvez également l'utiliser pour réaliser des mesures à partir de votre application pour l'iPhone et les envoyer à un serveur de manière à rendre l'application plus rapide et plus simple à utiliser.

Grâce à `ServerAccessObject`, toutes ces possibilités sont désormais facilement utilisables dans votre application hybride installée sur l'iPhone.

Openmirrors.com



A

Introduction à JSON

JSON (*JavaScript Object Notation*) apporte des possibilités très intéressantes. Il permet de convertir des objets et des tableaux JavaScript en chaînes de caractères, qui peuvent ensuite être transmises sur le réseau ou enregistrées dans une base de données. Ces chaînes peuvent ultérieurement être reconverties en objets ou tableaux sur un autre ordinateur ou après avoir été extraites d'une base de données. Cette capacité à sérialiser et à charger des objets et des tableaux JavaScript ouvre de nombreuses possibilités. Cette annexe présente une API pour la bibliothèque JavaScript JSON et fournit quelques exemples de son utilisation.

Section 1 : les fondamentaux

La transmission d'informations d'un système à un autre a toujours représenté un problème. Cela devient particulièrement évident dans le développement d'applications web où un serveur peut être écrit dans n'importe quel langage et s'exécuter sur différents types d'ordinateurs. XML a été l'un des premiers formats indépendants de l'appareil, du système d'exploitation et du langage proposé pour résoudre ce problème d'échange et

certaines de ses utilisations se sont révélées intéressantes. L'utilisation du format XML pour transférer des données peut sembler exagérée, en particulier pour les petits éléments d'information généralement envoyés avec AJAX. Si vous souhaitez envoyer uniquement un petit tableau de nombres ou une mappe de clés-valeurs, XML se révèle un peu verbeux. Ce point a été résolu, non en inventant une nouvelle technologie, mais en employant une fonctionnalité des langages interprétés.

Tous les principaux langages interprétés faiblement typés disposent d'une fonctionnalité d'évaluation, par exemple une fonction `eval`, qui permet d'exécuter des chaînes de caractères comme s'il s'agissait d'un code source. Cette possibilité est puissante, mais dangereuse. En cas de mauvaise utilisation, elle peut totalement ouvrir l'application aux pirates et aux abus. Par ailleurs, tous les principaux langages interprétés faiblement typés ont la possibilité de définir des tableaux et des objets sans passer par un mot clé d'instanciation de type `new`.

Si vous examinez des exemples JavaScript, vous verrez comment créer un tableau. Voici comment procéder dans une approche orientée objet :

```
var tableau = new Array();
tableau.push(5);
tableau.push(13);
tableau.push('Bonjour');
```

La solution suivante n'est pas orientée objet :

```
var tableau = [5,13,'Bonjour'];
```

Ces deux exemples créent des tableaux parfaitement identiques. Le deuxième est intéressant pour notre présentation de JSON. Conjugué aux possibilités JavaScript d'évaluation d'une chaîne comme du code, il permet de bénéficier de JSON.

Le code suivant crée une chaîne de caractères qui correspond à la manière dont un programmeur utiliserait la seconde méthode de création d'un tableau. Il s'agit non pas du tableau en soi, mais d'une description de ce que doit être ce tableau.

```
var uneChaine = "[5,13, 'Bonjour']";
// Évaluer la chaîne.
var tableau = eval(uneChaine);
```

Openmirrors.com

La dernière ligne parse la chaîne en un code source JavaScript, interprète ce contenu JavaScript et l'exécute. Dans cet exemple simple, la chaîne se trouve dans la même application que l'appel à la fonction `eval` et, par conséquent, cette utilisation montre peu d'intérêt. En revanche, si la chaîne provient de la partie Objective-C d'une application QuickConnectiPhone ou, plus classiquement, d'un serveur, l'appel à `eval` a plus de sens.

La création d'objets est comparable. Dans l'approche orientée objet suivante, un objet est créé et des attributs lui sont ensuite ajoutés :

```
var objet = new Object();
objet.largeur = 5;
objet.hauteur = 13;
objet.message = 'Bonjour';
```

Voici la version non orientée objet :

```
var objet = {"largeur":5,"hauteur":13,"message":"Bonjour"};
```

Le code suivant est de type JSON :

```
var uneChaine = '{"largeur":5,"hauteur":13,"message":"Bonjour"}';
// Évaluer la chaîne.
var objet = eval(uneChaine);
```

Bien que cette procédure soit au cœur de JSON, sa mise en œuvre par le programmeur présente un danger. Par exemple, si la chaîne a été envoyée depuis le code JavaScript vers le côté Objective-C de QuickConnectiPhone et si elle contient des instructions complexes, elle peut potentiellement effacer le disque dur.

Des bibliothèques JSON ont été développées pour prendre en charge les problèmes de sécurité. Celle que nous utilisons dans la partie JavaScript se nomme Json2 et est proposée par le fichier *json2.js*. Json2 est l'un des parseurs JSON les plus utilisés en JavaScript. Puisque des données sont régulièrement échangées avec la partie Objective-C des applications fondées sur QuickConnectiPhone, vous devez comprendre l'API de cette bibliothèque.

Section 2 : une API JavaScript pour JSON

L'API de Json2, décrite au Tableau A.1, est simple. Elle est constituée de deux fonctions : une première pour convertir un objet ou un tableau en une chaîne de caractères et une seconde pour convertir des chaînes de caractères en objets.

Tableau A.1 : L'API de Json2

Fonction	*Paramètres*
`JSON.stringify(entity, replacer, space, linebreak)`	Paramètre obligatoire : `entity` – l'objet, le tableau ou le type primitif JavaScript à convertir. Paramètres facultatifs : `replacer` – une fonction ou un tableau qui permet de redéfinir la génération par défaut des chaînes de caractères pour les valeurs associées aux clés des entités JavaScript. `space` – un nombre ou un caractère, comme `'\t'` ou ` `, utilisé pour indenter les entités JavaScript qui sont les valeurs enregistrées avec des clés dans d'autres entités. `linebreak` – un ou plusieurs caractères qui remplacent le caractère `'\n'` par défaut, comme `'\r\n'` ou ` `.
`JSON.parse(string, reviver)`	Paramètre obligatoire : `string` – la chaîne JSON à convertir en un objet ou un tableau de JavaScript. Paramètre facultatif : `reviver` – une fonction qui met en œuvre un comportement inverse à celui de la fonction `replacer` utilisée avec la méthode `stringify`.

La première fonction se nomme `stringify`. Elle prend plusieurs arguments, mais dans le cas de QuickConnectiPhone seul le premier est obligatoire. Il s'agit de l'objet ou du tableau à convertir en chaîne de caractères. En voici un exemple :

```
var chaineJSON = JSON.stringify(objet);
```

Openmirrors.com

La conversion d'une chaîne en objet est tout aussi simple :

```
var objet = JSON.parse(uneChaine);
```

Les tableaux sont pris en charge de la même manière.

Un exemple complet d'utilisation de Json2 pour convertir en chaînes et parser des objets se trouve dans le répertoire *Examples/JSON* de QuickConnectiPhone (*object_JSON_ example.html*). La Figure A.1 illustre la conversion d'un objet en une chaîne, qui est ensuite reconvertie en un objet, ainsi que l'affichage de l'attribut `size` de l'objet.

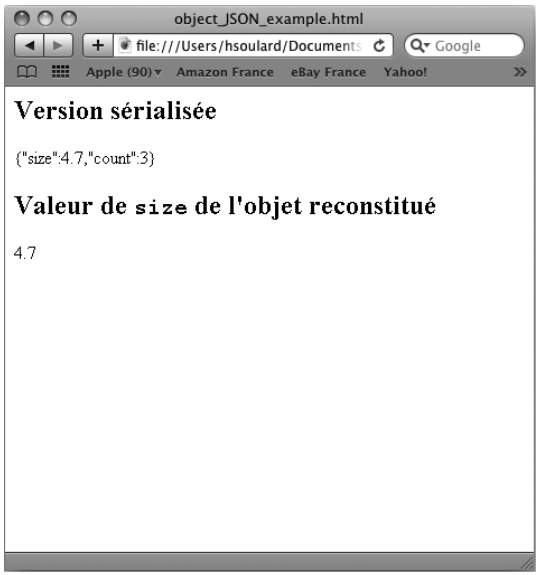


Figure A.1
Convertir un objet en chaîne de caractères et inversement.

Le répertoire *Examples/JSON* contient également l'exemple *array_JSON_example.html*, qui illustre l'utilisation de Json2 avec des tableaux (voir Figure A.2).

Notez que ces deux exemples utilisent les termes standard dans l'industrie pour les fonctions de sérialisation et de reconstruction : `stringify` et `parse`.

La bibliothèque Json2 permet également de passer des types primitifs, comme des nombres. Les chaînes de caractères sont aussi prises en charge. Ce n'est pas le cas de toutes les bibliothèques JSON dans tous les langages. La Figure A.3 illustre ces possibilités.

Grâce à la bibliothèque Json2, vous pouvez convertir en chaîne de caractères n'importe quel type de données, et le reconstruire ensuite. La bibliothèque JSON de la partie Objective-C prend également en charge les types primitifs et les chaînes de caractères.

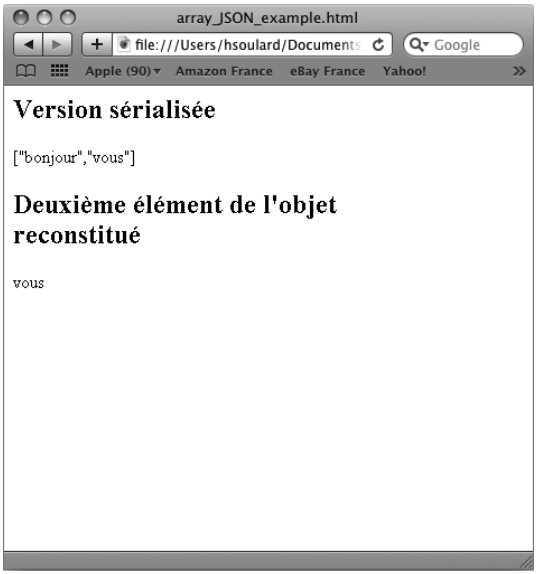


Figure A.2
Convertir un tableau en chaîne de caractères et inversement.

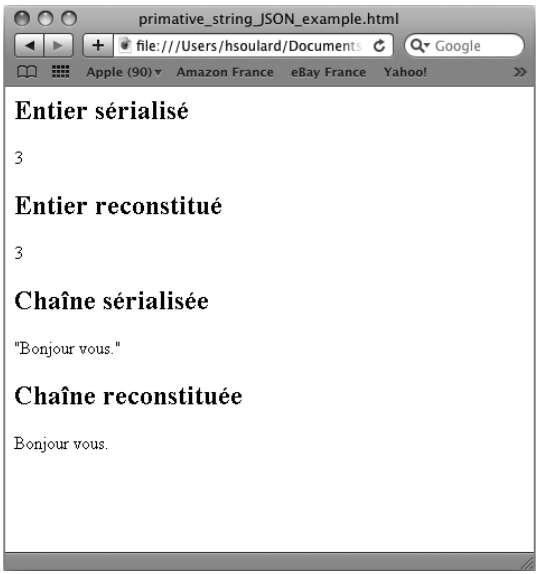


Figure A.3
Convertir des types primitifs et des chaînes de caractères en chaînes de caractères et inversement.

Openmirrors.com

En résumé

JSON constitue une bonne solution pour échanger des informations. Il est indépendant de l'appareil, du système d'exploitation et du langage. Il existe des parseurs open-source gratuits dans tous les langages couramment utilisés, dont certains, comme PHP, les intègrent directement.

La bibliothèque Json2 fournie avec QuickConnectiPhone est facile à utiliser et vous permet d'échanger des données avec le côté Objective-C d'une application hybride.

Openmirrors.com



B

Plan de développement pour QuickConnectFamily

Puisque le développement de QuickConnectiPhone en est à ses débuts, les améliorations sont fréquentes. Il est donc important de suivre de près les mises à jour. Le Tableau B.1 recense les fonctionnalités disponibles au 3 octobre 2009. Vous pouvez consulter la feuille de route du projet à l'adresse **http://quickconnect.pbworks.com/Porting-Roadmap**.

Oui indique que la fonctionnalité est disponible. *En cours* signifie que la fonctionnalité est en cours de développement. *Prévu* signale que la fonctionnalité est envisageable mais que son développement n'est pas commencé. *Impossible* s'applique à une fonctionnalité qui ne peut pas être mise en œuvre sur l'appareil. Les cases qui contiennent un tiret signalent un travail qui n'est pas réalisé au moment de l'écriture de ces lignes.



Bien que le développement soit prévu pour Windows et Windows Mobile, rien n'est encore commencé et le Tableau B.1 omet donc ces systèmes.

Openmirrors.com

Tableau B.1 : Feuille de route de QuickConnectFamily au 3 octobre 2009

	iPhone	*Android*	*Mac*	*Linux*	*Symbian*	*Windows*	*Windows Mobile*	*Palm WebOS*[1]
Géolocalisation	Oui	Oui	Impossible	Impossible	Prévu	Impossible	—	Prévu
Accéléromètre	Oui	Oui	Prévu	Impossible	Prévu	Impossible	—	Prévu
Vibreur	Oui	Oui	Impossible	Impossible	—	Impossible	—	—
Réseau *ad hoc*	Oui	En cours	Oui	Prévu	Prévu	Prévu	Prévu	—
Enveloppe JavaScript pour les bases de données (SQLite)	Oui	Impossible	Oui	Oui	En cours	Prévu	Prévu	Prévu
Enveloppe pour les bases de données natives installées (SQLite)	Oui	Oui	Oui	Oui	En cours	En cours	En cours	—
Enveloppe AJAX	Oui	Impossible[2]	Oui	Oui	En cours	En cours	En cours	—
Bibliothèque de glisser-déposer	Oui	Impossible[3]	Oui	—	—	—	—	Prévu
Réseau par câble de synchronisation	Prévu	—	Prévu	—	—	—	—	—
Accès à l'appareil photo	En cours	—	Oui	—	—	—	—	—
Géolocalisation d'images	Prévu	—	Prévu	—	—	—	—	—
Sons système (lecture)	Oui	Oui	Oui	Prévu	Prévu	En cours	En cours	Prévu
Enregistrement/lecture de fichiers audio	Oui	Oui	Oui	Prévu	—	En cours	—	Prévu
Enregistrement/lecture de fichiers vidéo	En cours	Impossible	Oui	Prévu	—	En cours	—	Prévu
Sélecteurs de date/heure natifs	Oui	Impossible	Impossible	Impossible	Impossible	Prévu	Prévu	Prévu
Cartes Google embarquées	Oui	En cours	Oui	Prévu	Prévu	Prévu	Prévu	Prévu

Tableau B.1 : Feuille de route de QuickConnectFamily au 3 octobre 2009 *(suite)*

	iPhone	*Android*	*Mac*	*Linux*	*Symbian*	*Windows*	*Windows Mobile*	*Palm WebOS*[1]
Bibliothèque pour tableaux et graphiques	Oui	En cours	En cours	En cours	En cours	En cours	En cours	Prévu
Informations sur l'appareil	Oui	Oui	En cours	En cours	En cours	—	—	Prévu
Réseau P2P Bluetooth	En cours	—	—	—	—	—	—	—
Diffusion audio	En cours	—	—	—	—	—	—	—
Balises audio/vidéo de HTML5	Oui	—	Oui	Oui	—	—	—	—
API pour les contacts	En cours (bêta)	—	—	—	—	—	—	—
Notification	En cours (bêta)	—	—	—	—	—	—	—
Lecteur de bibliothèque audio	En cours	—	—	—	—	—	—	—
Sélecteur audio natif	En cours	Oui	—	—	—	—	—	—
Discussion vocale depuis une application	En cours	—	—	—	—	—	—	—
Courrier électronique depuis une application	En cours	—	—	—	—	—	—	—
Détection de secousse	En cours (bêta)	—	—	—	—	—	—	—
Copier-coller	Oui	—	Oui	—	—	—	—	—

1. Cette version ne peut pas être fournie tant que le SDK de WebOS ne fait plus l'objet d'un accord de non-divulgation.
2. Google n'inclut pas l'objet `XMLHttpRequest` dans son objet `WebView`. Il existe une solution, mais elle se révèle très lente.
3. Google n'inclut pas les transitions et les animations CSS nécessaires à la mise en œuvre de cette fonctionnalité. L'utilisation de JavaScript pour le glisser-déposer est une solution trop lente.

Définitions des termes employés au Tableau B.1 :

- Géolocalisation. Obtenir les coordonnées de localisation GPS.
- Accéléromètre. Obtenir les changements d'orientation en x, y et z.
- Vibreur. Déclencher le vibreur de l'appareil.
- Réseau *ad hoc*. Rechercher et communiquer avec les autres appareils du voisinage qui exécutent la même application.
- Enveloppe JavaScript pour les bases de données (SQLite). Utiliser les bases de données HTML 5 intégrées.
- Enveloppe pour les bases de données natives installées (SQLite). Utiliser les bases de données SQLite livrées avec une application.
- Enveloppe AJAX. Une bibliothèque AJAX simple d'emploi pour obtenir des données distantes.
- Bibliothèque de glisser-déposer. Une bibliothèque simple d'emploi pour que l'utilisateur puisse déplacer, faire pivoter et redimensionner des éléments à l'écran.
- Réseau par câble de synchronisation. Accéder à des données et en transférer avec une machine de bureau à l'aide du câble de synchronisation.
- Accès à l'appareil photo. Prendre et enregistrer des photos.
- Géolocalisation d'images. Accéder aux informations de géolocalisation ajoutées aux photos prises avec l'appareil.
- Sons système (lecture). Jouer des sons brefs (d'une durée inférieure à 5 secondes).
- Enregistrement/lecture de fichiers audio. Enregistrer un fichier audio en utilisant l'appareil et jouer ces fichiers ou les fichiers audio livrés avec l'application.
- Enregistrement/lecture de fichiers vidéo. Enregistrer un fichier vidéo en utilisant l'appareil et jouer ces fichiers avec l'application.
- Sélecteurs de date/heure natifs. Afficher et utiliser les sélecteurs écrits en Objective-C à la place des sélecteurs JavaScript limités.
- Cartes Google embarquées. Ajouter des cartes Google personnalisées dans une application à la place des cartes standard ou de l'affichage de l'application de cartographie.
- Bibliothèque pour tableaux et graphiques. Une bibliothèque simple d'emploi permettant d'afficher des graphiques en segments, en barres, en secteurs et autres.
- Informations sur l'appareil. Obtenir les caractéristiques de l'appareil.

Openmirrors.com

- Réseau P2P Bluetooth. Établir un réseau de pair à pair en utilisant la connectivité Bluetooth.
- Diffusion audio. Accéder à des fichiers audio en *streaming*.
- Balises audio/vidéo de HTML 5. Prendre en charge les nouvelles balises de HTML 5.
- API pour les contacts. Accéder aux informations disponibles dans l'application de gestion des contacts.
- Notification. Envoyer des notifications.
- Lecteur de bibliothèque audio. Exploiter une bibliothèque de fichiers audio.
- Sélecteur audio natif. Afficher et utiliser les sélecteurs écrits en Objective-C.
- Discussion vocale depuis une application. Démarrer une conversation vocale sans quitter l'application.
- Courrier électronique depuis une application. Accéder à la messagerie électronique sans quitter l'application.
- Détection de secousse. Détecter si l'utilisateur secoue l'appareil.
- Copier-coller. Prise en charge de cette fonctionnalité.

Openmirrors.com



C

Plan de développement pour PhoneGap

Puisque le développement de PhoneGap en est à ses débuts, les améliorations sont fréquentes. Il est donc important de suivre de près les mises à jour. Le Tableau C.1 vient du wiki PhoneGap. Il recense les fonctionnalités disponibles et prévues, telles qu'indiquées par les développeurs de ce framework. Vous pouvez consulter la feuille de route du projet à l'adresse **http://phonegap.pbworks.com/Roadmap**.

Oui indique que les fonctions JavaScript existent dans PhoneGap au 3 octobre 2009. L'auteur n'est pas en mesure de commenter la disponibilité pour la plate-forme Blackberry. *En cours* signifie que l'équipe PhoneGap travaille sur la fonctionnalité. Les cases qui contiennent un tiret signalent un travail qui n'est pas réalisé et qui n'existe pas encore. *Impossible* concerne les fonctionnalités que les développeurs considèrent indisponibles sur l'appareil.



Bien que le développement soit prévu pour Windows Mobile, rien n'est encore commencé et le Tableau C.1 omet donc ce système.

Tableau C.1 : Feuille de route de PhoneGap au 3 octobre 2009

	iPhone	***Android***	***Blackberry (OS 4.5)***	***Symbian***
Géolocalisation	Oui	Oui	Oui	Oui
Accéléromètre	Oui	Oui	Disponible dans OS 4.7	En cours
Appareil photo	Oui	En cours	En cours	—
Vibreur	Oui	Oui	Oui	Oui
Hors ligne (fichiers locaux)	En cours	Oui	En cours	—
API pour les contacts	Oui	—	Oui	Oui
Enveloppe SQLite	Oui	—	Impossible	
API XMPP	—	En cours	—	—
E/S du système de fichiers	—	En cours	En cours	—
Geste/toucher multiple	Oui	—	—	—
API SMS	—	En cours	—	—
API de téléphonie	En cours	En cours	En cours	—
Copier-coller	—	—	Oui	—
Sons (lecture)	Oui	En cours	—	En cours
Sons (enregistrement)	—	En cours	—	—
Bluetooth	—	—	—	—
Connexion Wi-Fi *ad hoc*	—	—	—	—
Cartes	Oui	En cours	En cours	—
Changement d'orientation	Oui	—	—	En cours
Disponibilité réseau	Oui	—	—	En cours
Magnétomètre	Oui (modèle 3GS)	En cours	—	—

Openmirrors.com

Définitions des termes employés au Tableau C.1 :

- Géolocalisation. Obtenir les coordonnées de localisation GPS.
- Accéléromètre. Obtenir les changements d'orientation en x, y et z.
- Appareil photo. Prendre et enregistrer des photos.
- Vibreur. Déclencher le vibreur de l'appareil.
- Hors ligne (fichiers locaux). Fichiers HTML, CSS et JavaScript installés avec l'application (non sur un serveur web).
- API pour les contacts. Accéder aux informations disponibles dans l'application de gestion des contacts.
- API XMPP. Messageries de type Jabber.
- E/S du système de fichiers. Lire et écrire des fichiers textuels ou binaires.
- Geste/toucher multiple. Utiliser un ou plusieurs doigts pour la saisie de données complexes.
- API SMS. Messagerie instantanée.
- API de téléphonie. Passer des appels téléphoniques.
- Copier-coller. Dupliquer des données saisies.
- Sons (lecture). Jouer des fichiers audio.
- Sons (enregistrement). Enregistrer des fichiers audio à l'aide de l'appareil.
- Bluetooth. Utiliser la connectivité Bluetooth avec d'autres appareils.
- Connexion Wi-Fi *ad hoc*. Rechercher et communiquer avec d'autres appareils.
- Cartes. Utiliser des cartes Google.
- Changement d'orientation. Détecter le passage en modes portrait et paysage.
- Disponibilité réseau. Détecter si l'appareil a accès au réseau.
- Magnétomètre. Obtenir les informations de la boussole intégrée.

Openmirrors.com

Index

Symboles

__(double souligné) 117
__gap, variable 117
__gap_device_model, variable 117

A

abort, méthode 196
accel, commande 94
Accéléromètres
  PhoneGap 130
  QuickConnectiPhone 94
Accès
  aux bases de données
    BrowserDBAccess, application d'exemple 151-153
    dans WebKit 161-172
    natives 172-182
    SQLite avec WebView 153-158
    SQLite natives 158-161
    vue d'ensemble 151
  aux données distantes
    BrowserAJAXAccess, application d'exemple 186-188
    fonction de contrôle de la sécurité 203
    ServerAccessObject 188
      Voir ServerAccessObject
    vue d'ensemble 185
Actualisation de l'affichage visible 36
add, fonction 38
Affichage
  cartes géographiques 133-138
  sélecteurs 97
AJAX 187
Anonymes, fonctions 161
Appareil, activation
  en JavaScript 92-98, 115-122
  en Objective-C 98-106, 122-130
applicationDidFinishLaunching, méthode 22, 24
Applications
  d'exemple
    BrowserAJAXAccess 186-188
    BrowserDBAccess 151-153
  d'immersion 69-70
  hybrides
    boîte d'alerte et 8
    définition 1
  non fondées sur les listes 64-68
Approvisionnement 16
Arrêter la lecture des enregistrements 95
Asynchrone, définition 49
AudioServicesPlaySystemSound, fonction 124

B

Balayement 59
Base de données
  BrowserDBAccess, application d'exemple 151-153
  de WebKit 161-172
    Database, objet 163-165
    dbAccess, méthode 163
    exemple de code 170
    generatePassThroughParameters, fonction 163
    getData, méthode 162
    passThroughParameters, tableau 163
    setData, méthode 162
    SQLError, objet 169
    SQLResultSet, objet 167
    SQLResultSetRowList, objet 167-169
    SQLTransaction, objet 165
  natives 172-182
    API SQLite3 175
    getDeviceData, méthode 173
    getNativeData, méthode 173
    makeCall, fonction 174
    SendDBResultVCO, objet 181
    setNativeData, méthode 173
  SQLite avec WebView 153-158
  SQLite natives 158-161
  terminologie 152
  vue d'ensemble 151
BCF (*Business Control Function*) 36, 38, 42, 50
Boîte d'alerte
  applications hybrides et 8
  PhoneGap 119
BrowserAJAXAccess, application d'exemple 186-188
BrowserDBAccess, application d'exemple 151-153

C

calculateSolutionsBCF, fonction 39
callFunc, fonction 52
Cartes géographiques
  afficher dans une application JavaScript QuickConnect 133-138
  module de QuickConnect 138-149
  zoom 144-148
Chaînes de caractères
  convertir des objets/tableaux en 208
  convertir en objets 209
Champs de base de données 152
changeView, fonction 63
checkNumbersValCF, fonction 41
checkSecurity, fonction 201
Classes
  DataAccessObject
    bases de données de WebKit 161-172
    bases de données SQLite avec WebView 153-158
    bases de données SQLite natives 158-161
    méthodes 154
  GlassAppDelegate 23
  QuickConnectViewControl-ler 22
  singletons 107
  SQLiteDataAccess 172-182
Clés
  étrangères 153
  primaires 153
code, attribut (SQLError) 169
Comportements standard 59
Contenu web, embarquer
  avec PhoneGap 29-30
  avec QuickConnectiPhone 25-29
Contrôleurs
  d'affichage 49-53
  d'application 41
  d'erreur 53-54
  métier 49-53
Conversion
  chaînes de caractères en objets 208, 209
  objets en chaînes de caractères 208
Copie de fichiers 13
Cube, transition 67

D

Dashcode 8
  modèle QuickConnectiPhone 8-10
  répertoires 14
  transitions 66

Openmirrors.com

DataAccessObject, classe
base de données de WebKit 161-172
Database, objet 163-165
dbAccess, méthode 163
exemple de code 170
generatePassThroughParameters, fonction 163
getData, méthode 162
passThroughParameters, tableau 163
setData, méthode 162
SQLError, objet 169
SQLResultSet, objet 167
SQLResultSetRowList, objet 167-169
SQLTransaction, objet 165
base de données SQLite
avec WebView 153-158
natives 158-161
méthodes 154
DataAccessObject, méthode 154
DataAccessObject.js, fichier 153
Database, objet 163-165
dbAccess, méthode 163, 169
Défiler, transition 66
Délégués 19
deleteScoreBCF, fonction 158
Déplacement 87
Développement, outils
PhoneGap
feuille de route 219
ressources en ligne 5
vue d'ensemble 1-2
QuickConnectiPhone
feuille de route 213
ressources en ligne 5
vue d'ensemble 1-2
Device.exec, fonction 119
Device.init, méthode 117
Device.Location.init, méthode 120
Device.vibrate, méthode 119
Diapositive, transition 67
didAccelerate, méthode 130
didUpdateToLocation, méthode 130
dispatchToBCF, fonction 49, 50
dispatchToECF, fonction 54
dispatchToValCF, fonction 45
dispatchToVCF, fonction 53
displayScoresVCF, fonction 156, 160
displaySiteDataVCF, fonction 190, 191, 192
displaySolutionVCF, fonction 39
Dissolution, transition 66
doCommand, méthode 104, 110, 111, 139, 181
DollarStash, application d'exemple 69
done, méthode 75
Données distantes, accéder
BrowserAJAXAccess, application d'exemple 186-188
fonctions de contrôle de la sécurité (SCF) 203
ServerAccessObject, classe 188-193
displaySiteDataVCF, fonction 190, 191, 192
getData, méthode 188, 193
getSiteDataBCF, fonction 189
makeCall, méthode 193, 195, 196
onreadystatechange, fonction anonyme 200, 201
ServerAccessObject, méthode 188
setData, méthode 189, 193
XMLHttpRequest, objet 196, 198
vue d'ensemble 185
Données, obtenir 36
Double souligné (__) 117
Double-touchers, avec les cartes géographiques 143
dragAndGesture, application d'exemple 79

E

ECF (*Error Control Function*) 39, 43, 53-54
Échanger, transition 67
Embarquer
contenu web
PhoneGap 29-30
QuickConnectiPhone 25-29
Google Maps 133-138
Enregistrements
audio
arrêter 95
lire en JavaScript 95
de base de données 152
entryECF, fonction 39
eval
fonction 42
type 206
executeSQL, méthode 166, 170

F

Faire pivoter, transition 67
Feuilles de route
  PhoneGap 219
  QuickConnectiPhone 213
Fichiers
  copier 13
  DataAccessObject.js 153
  ServerAccessObject.js 189
Fonctionnalités d'une application, étapes de création 54
Fonctions
  anonymes 161
  checkSecurity 201
  d'association, API 40
  de contrôle de l'affichage (VCF) 36
    displayScoresVCF 156, 160
  de contrôle de la sécurité (SCF) 203
  de contrôle de la validation (ValCF) 36
  de contrôle des erreurs (ECF) 39
  de contrôle métier (BCF) 36
  de contrôle, modularité 36
  de rotation 77
  deleteScoreBCF 156-158
  dispatchToVCF 53
  displayScoresVCF 156, 160
  displaySiteDataVCF 190, 191, 192
  generatePassThroughParameters 163
  getSiteDataBCF 189
  makeCall 174
  onreadystatechange 200, 201
  parse 208, 209
  requestHandler 201
  stringify 208
Fondu, transition 67
Frameworks 34
FrontController, API 37

G

generatePassThroughParameters, fonction 163
Gestes 59, 76
getAllResponseHeaders, méthode 196
getData methodgetData, méthode 154, 193
getData, méthode 156, 162, 188
getDeviceData, méthode 173
getGPSLocation, fonction 96
getInstance, méthode 108
getNativeData, méthode 155, 160, 173
getResponseHeader, méthode 196
getSiteDataBCF, fonction 189
GlassAppDelegate, classe 23
Glisser-déposer 59
  API 78
  modules 78-89
  sautillement des éléments 73
goForward, méthode 64
Google Maps, dans une application JavaScript QuickConnect 133-138
goSub, fonction 64
gotAcceleration, fonction 121
GPS
  JavaScript 96
  Objective-C 103, 104, 105
  PhoneGap 120, 126
  QuickConnectiPhone 96, 121
Groupes Xcode 14
Guide de l'interface utilisateur 57

H

handleRequest, fonction 37, 44
handleRequestCompletionFromNative, méthode 182
HIG (*Human Interface Guide*) 57-61
HistoryExample, application d'exemple 61
Hybrides, applications, boîte d'alerte et 8

I

Imagerie médicale, applications 69
Immersion, applications 69-70
InfoWindow, classe 138, 149
initWithContentsOfFile, méthode 127
initWithFrame:andLocations, méthode 138
insertID, attribut (SQLResultSet) 167
Instanciation d'objets en Objective-C 16
Instructions préparées 157

Openmirrors.com

Interfaces
applications fondées sur les vues 63, 64
interfaces fondées sur les listes 61-64
transformations CSS 71-78
vues 64
Interrupteurs 60
isDraggable, attribut 81
item, méthode 168

J

JavaScript
activation de l'appareil 92-98, 115-122
modularité 33-34
exemple QuickConnect 35-43
scroll, fonction 141
JavaScript Object Notation
Voir JSON
JSON (*JavaScript Object Notation*) 95, 192
activation de l'appareil en Objective-C 101
API Json2 208-209
vue d'ensemble 205-207
Json2, API 208-209
JSONStringify, méthode 181

L

Lecture
enregistrements audio en JavaScript 95
sons système en Objective-C 102
length, attribut (SQLResultSetRowList) 168
Listes
applications non fondées sur 64-68
interfaces fondées sur 61-64
loadView, méthode 25

M

makeCall, fonction 92, 93, 174
makeCall, méthode 193, 195, 196
makeChangeable, fonction 79, 81
makeDraggable, fonction 78, 80
Mandant-délégué, relations 19
Mandants 19
Mandataires 19
mapCommands, méthode 102
mapCommandToCO, méthode 112
MapView, classe 138
math, commande 37, 39
message, attribut (SQLError) 169
Méthodes
abort 196
applicationDidFinishLaunching 22
DataAccessObject 154
dbAccess 163, 169
de rappel 127
doCommand 181
executeSQL 166, 170
getAllResponseHeaders 196
getData 154, 156, 162, 188, 193
getDeviceData 173
getNativeData 155, 160, 173
getResponseHeader 196
handleRequestCompletion FromNative 182
item 168
JSONStringify 181
makeCall 193, 195, 196
open 196
openDatabase 164
readyState 197
responseText 197
responseXML 197
send 197
ServerAccessObject 188
setData 154, 155, 156, 162, 189, 193
setNativeData 155, 173
SetRequestHeader 197
sqlite3_bind_blob 177
sqlite3_bind_double 178
sqlite3_bind_int 178
sqlite3_changes 176
sqlite3_close 175
sqlite3_column_blob 177
sqlite3_column_bytes 177
sqlite3_column_count 175
sqlite3_column_double 176
sqlite3_column_int 176
sqlite3_column_name 175
sqlite3_column_text 177
sqlite3_column_type 176
sqlite3_errmsg 175
sqlite3_finalize 177
sqlite3_open 175
sqlite3_prepare_v2 175
sqlite3_step 176
sqlite3_stmt 175

Méthodes (*suite*)
stringByEvaluatingJavaScriptFromString 182
transaction 164, 170
XMLHttpRequest 196
Modèles
QuickConnectiPhone
pour Dashcode 8-10
pour Xcode 12-16
Modularité
fonctions de contrôle 36
JavaScript 33-34
exemple QuickConnect 35-43
implémenter dans QuickConnectiPhone 44-48
Modules
définition 34
glisser-déposer 78-89
redimensionnement 78-89
rotation 78-89
moveX:andY, méthode 147

N

Natives, bases de données
accéder 172-182
API SQLite3 175
getDeviceData, méthode 173
getNativeData, méthode 173
makeCall, fonction 174
SendDBResultVCO, objet 181
setNativeData, méthode 173
SQLite, accéder 158-161
Navigateur, partie 61-64
NSLog, fonction 111

O

Objective-C 16-19
activation de l'appareil 98-106, 122-130
architecture de QuickConnectiPhone 107-113
instancier des objets 16
module de cartographie de QuickConnect 138-149
sélecteurs 106
structure d'une application PhoneGap 23-25
QuickConnectiPhone 19-22
Objets
convertir des chaînes de caractères en 209
convertir en chaînes de caractères 208
créer 207
Database 163-165
instancier en Objective-C 16
SendDBResultVCO 181
ServerAccessObject 188
SQLError 169
sqlite3 175
SQLResultSet 167
SQLResultSetRowList 167-169
SQLTransaction 165
XMLHttpRequest 196, 198
oldScale, attribut 82
ongesturechange, événement 86
onreadystatechange
attribut 197
fonction anonyme 200, 201
ontouchchange, gestionnaire 74
ontouchend, gestionnaire 75
open, méthode 196
openDatabase, méthode 164

P

parse, fonction 208, 209
passThroughParameters, tableau 163
PhoneGap 8, 9
accéléromètres 130
activation de l'appareil
en JavaScript 115-122
en Objective-C 122-130
API JavaScript 118
boîte d'alerte 119
embarquer du contenu web 29-30
feuille de route 219
GPS 120, 126
ressources en ligne 5
signaler un dysfonctionnement à l'utilisateur 119
son système 128
structure Objective-C d'une application 23-25
vibreur 117, 124
vue d'ensemble 1-2
Pin, classe 138, 142
Pincement 59
play, commande 93
playSound
commande 93
méthode pour PhoneGap 122
playTweetSound, fonction 121
Pointeurs 17
prepareDrag, fonction 82

Openmirrors.com

Préparées, instructions 157
prepareGesture, fonction 86
Protocoles 20

Q

QCCommandObject, classe 110
QuickConnectFamily, programme d'installation 8
QuickConnectiPhone
  accéléromètres 94
  activation de l'appareil
    en JavaScript 92-98
    en Objective-C 98-106
  afficher des cartes 133-138
  embarquer du contenu web 25-29
  feuille de route 213
  GPS 96, 121
  implémentation Objective-C 107-113
  implémenter une conception modulaire 44-48
  modèles
    Dashcode 8-10
    Xcode 12
  modularité de JavaScript 35-43
  module de cartographie 138-149
  ressources en ligne 5
  structure Objective-C d'une application 19-22
  vibreur 93, 100
  vue d'ensemble 1-2
QuickConnectViewController, classe 22

R

rangeOfString, méthode 125
readyState, méthode 197
Récursivité, définition 52
Redimensionnement, module 78-89
Répertoires de Dashcode 14
requestHandler, fonction 201
responseText, méthode 197
responseXML, méthode 197
Ressources en lignes
  PhoneGap 5
  QuickConnectiPhone 5
retVal, tableau 181
Rotation, module 78-89
Rotation, transition 67
rows, attribut (SQLResultSet) 167
rowsAffected, attribut (SQLResultSet) 167

S

Saisie textuelle de l'utilisateur 59
Saut des éléments, lors du glisser-déposer 73
SCF (*Security Control Function*) 203
scroll, fonction 141
Sélecteurs
  afficher 97
  Objective-C 106
send, méthode 197
SendDBResultVCO, objet 181
sendloc, commande 105
ServerAccessObject, classe 188-193, 193-202
  displaySiteDataVCF, fonction 190, 191, 192
  getData, méthode 188, 193
  getSiteDataBCF, fonction 189
  makeCall, méthode 193, 195, 196
  onreadystatechange, fonction anonyme 200, 201
  ServerAccessObject, méthode 188
  setData, méthode 189, 193
  XMLHttpRequest, objet 196, 198
ServerAccessObject, méthode 188
ServerAccessObject.js, fichier 189
setData, méthode 154, 155, 156, 162, 189, 193
setMapLatLngFrameWithDescription, méthode 148
setNativeData, méthode 155, 173
SetRequestHeader, méthode 197
setStartLocation, fonction 73
shouldStartLoadWithRequest, fonction 99
showDateSelector, fonction 97
showMap, fonction 136
showPickResults, commande 97
Singletons, classes 107
singleTouch, message 143
Sons système
  JavaScript 93
  jouer en Objective-C 102
  PhoneGap 128
Sous-présentations, liste 62
SQLError, objet 169

SQLite, bases de données
avec WebView 153-158
de WebKit 161-172
Database, objet 163-165
dbAccess, méthode 163
exemple de code 170
generatePassThroughParameters, fonction 163
getData, méthode 162
passThroughParameters, tableau 163
setData, méthode 162
SQLError, objet 169
SQLResultSet, objet 167
SQLResultSetRowList, objet 167-169
SQLTransaction, objet 165
natives 158-161
SQLite3, API 175
sqlite3, objet 175
sqlite3_bind_blob, méthode 177
sqlite3_bind_double, méthode 178
sqlite3_bind_int, méthode 178
sqlite3_changes, méthode 176
sqlite3_close, méthode 175
sqlite3_column_blob, méthode 177
sqlite3_column_bytes, méthode 177
sqlite3_column_count, méthode 175
sqlite3_column_double, méthode 176
sqlite3_column_int, méthode 176
sqlite3_column_name, méthode 175
sqlite3_column_text, méthode 177
sqlite3_column_type, méthode 176
sqlite3_errmsg, méthode 175
sqlite3_finalize, méthode 177
sqlite3_open, méthode 175
sqlite3_prepare_v2, méthode 175
sqlite3_step, méthode 176
sqlite3_stmt, méthode 175
SQLiteDataAccess, classe 172-182
SQLResultSet, objet 167
SQLResultSetRowList, objet 167-169
SQLTransaction, objet 165
status, messages (XMLHttpRequest) 198
statusText, chaîne (XMLHttpRequest) 197, 198
stringByEvaluatingJavaScript FromString, méthode 182
stringify, fonction 208
Synchrone, définition 49

T

Tableaux
convertir en chaînes 208
créer 206
passThroughParameters 163
retVal 181
Tables 152
Téléchargement de Xcode 5
Touch, classe 72
Touchers
événements de 72
images pour indiquer 64
simples, avec les cartes géographiques 143
simples, avec les cartes géographiques 143
zones 58
touchesBegan, méthode 144
touchesMoved:withEvent, méthode 140, 147
transaction, méthode 164, 170
Transformations CSS personnalisées, créer 71-78
Transitions 66
translate, fonction 75
Type eval 206

U

UIWebView
API 27
classe 25, 26
Utilisateur
écrans visibles, actualiser 36
saisie, valider 36

V

ValCF (*Validation Control Function*) 36, 38, 43, 48
Valider la saisie de l'utilisateur 36
VCF (*View Control Function*) 36, 38, 43, 49, 53
Vibreur
PhoneGap 117, 124
QuickConnectiPhone 93, 100

Openmirrors.com

Vues 64
applications fondées sur 63, 64
secondaires 27

W

WebKit 71
bases de données 161-172
Database, objet 163-165
dbAccess, méthode 163
exemple de code 170
generatePassThroughParameters, fonction 163
getData, méthode 162
passThroughParameters, tableau 163
SQLError, objet 169
SQLResultSet, objet 167
SQLResultSetRowList, objet 167-169
SQLTransaction, objet 165
webkitTransform, attribut 71, 74
webMapView, attribut 139
WebView, base de données SQLite 153-158
webView:shouldStartLoadWithRequest:navigationType, méthode 124
webViewDidStartLoad, méthode 122

X

Xcode
groupes 14
modèles
QuickConnect 12-16
télécharger 5
XMLHttpRequest, méthode 196
XMLHttpRequest, objet 196, 198

Z

Zoom, cartes géographiques 144-148

Le Programmeur

Développez des applications pour iPhone avec HTML, CSS et JavaScript

Découvrez la manière la plus simple et la plus rapide de développer des applications iPhone !

Pour créer des applications iPhone, inutile de maîtriser l'Objective-C : vous pouvez recourir aux technologies et aux outils du Web que vous utilisez déjà – JavaScript, HTML et CSS. Cet ouvrage vous explique comment combiner les frameworks QuickConnect et PhoneGap avec le kit de développement d'Apple pour créer des applications sécurisées de grande qualité à destination des iPhone.

L'auteur y détaille le processus de développement, de la création de superbes interfaces utilisateur à la compilation, au déploiement et à l'exécution des applications. Il présente des techniques et des exemples de code conçus pour rationnaliser le développement, supprimer la complexité, optimiser les performances et exploiter toutes les possibilités de l'iPhone, de son accéléromètre et son GPS à sa base de données intégrée.

Grâce à cet ouvrage, les développeurs web pourront rapidement programmer pour l'iPhone en exploitant les outils qu'ils connaissent déjà.

Les codes sources des exemples sont téléchargeables sur le site www.pearson.fr.

- **Développer avec Dashcode et Xcode**
- **Modularité JavaScript**
- **Interfaces utilisateur**
- **GPS, accéléromètre et autres fonctions natives avec QuickConnectiPhone**
- **GPS, accéléromètre et autres fonctions natives avec PhoneGap**
- **Cartes Google**
- **Bases de données**
- **Données distantes**
- **Introduction à JSON**
- **Plan de développement pour QuickConnectFamily**
- **Plan de développement pour PhoneGap**

À propos de l'auteur

Lee S. Barney, expert en développement d'applications mobiles et web, est le créateur du framework Quickconnect, qui permet de développer des applications en JavaScript pour l'iPhone.

Niveau : Intermédiaire

Catégorie : Développement mobile

PEARSON

Pearson Education France
17 rue Soyer 92200 Neuilly-sur-Seine
Tél. : 01 72 74 90 00
Fax : 01 72 74 90 17
www.pearson.fr

ISBN : 978-2-7440-4096-2



Openmirrors.com