12

Exécuter des applications sur plusieurs plates-formes et appareils iOS

Le SDK d'iOS a été annoncé publiquement en février 2008. À l'heure de son lancement, seuls deux appareils l'utilisaient : l'iPhone et l'iPod touch. Apple n'a cessé de le mettre à jour depuis et a introduit en 2010 un autre membre de la famille, l'iPad. Toujours en 2010, un autre appareil tournant sous iOS a été présenté : l'Apple TV. Qui sait ce que l'avenir nous réserve ? Certaines rumeurs font même état d'une incursion d'Apple dans le domaine des téléviseurs, avec un premier modèle capable d'exécuter les applications de l'App Store. On évoque également la possibilité de contrôler des jeux tournant sur l'Apple TV, contrôlables à l'aide d'un iPhone ou d'un iPod touch.

Chaque année, une nouvelle version du SDK accompagne la sortie de deux ou trois nouveaux modèles d'appareils, bien souvent équipés de capteurs supplémentaires. La puce GPS a été introduite avec l'iPhone 3G, le magnétomètre (un capteur qui indique la direction du nord magnétique, à la manière d'une boussole) est apparu avec l'iPhone 3GS et le gyroscope (pour des jeux plus immersifs), avec l'iPhone 4. L'iPad a introduit une toute nouvelle interface utilisateur, avec un écran bien plus grand que celui de l'iPhone mais sans caméra. Il aura fallu attendre la seconde version de la tablette pour profiter des deux capteurs que l'on connaît aujourd'hui avec l'iPad 2.

Dans le même esprit, chaque nouvelle version du SDK s'accompagne de fonctionnalités inédites : les achats In App, le service des notifications push, Core Data et MapKit avec iOS3 ; le multitâches, les blocs et Grand Central Dispatch avec iOS 4 ; iCloud, l'intégration de Twitter et les storyboards avec iOS 5, pour ne citer que les principales nouveautés. Lorsque vous exploitez ces fonctionnalités, vous pouvez vous demander comment préserver la compatibilité avec les utilisateurs disposant d'une version plus ancienne du système d'exploitation. Gardez à l'esprit, toutefois, que si vous utilisez une fonctionnalité très récente du SDK, vous devrez alors oublier vos anciens utilisateurs (une solution à éviter) ou développer du code qui s'adapte à chaque cas de figure. Dans ce dernier cas, vous devrez intégrer une fonctionnalité

équivalente pour les anciens utilisateurs, ou leur indiquer qu'une mise à jour leur permettrait de profiter d'options inédites.

En tant que développeur, vous devez être capable de développer du code qui s'adapte facilement à chaque type d'appareil et de plate-forme. Pour servir un tel objectif, il vaut mieux profiter des API du framework Cocoa afin de détecter les capacités de chaque appareil, plutôt que développer du code en partant du principe qu'un certain capteur est accessible. En clair, les développeurs ont tout intérêt à ne pas présumer des possibilités matérielles des appareils qu'ils ciblent.

Dans ce chapitre, nous allons parcourir une série de stratégies qui vous aident à développer du code qui s'adapte facilement à de multiples plates-formes et appareils, en utilisant les différentes API du framework Cocoa. Au cours de ce chapitre, vous développerez une extension de catégorie pour la classe `UIDevice`, en ajoutant des méthodes visant à détecter les fonctionnalités qui ne sont pas forcément exposées par le framework.

Développer pour de multiples plates-formes

iOS a été introduit avec la version 2.0 du SDK, et la cinquième mouture du système correspond ainsi à la quatrième version disponible pour les développeurs. L'un des principaux atouts d'iOS par rapport aux plates-formes concurrentes tient au délai des mises à jour : les utilisateurs n'ont pas besoin d'attendre que les opérateurs "approuvent" ces mises à jour et, par leur gratuité, elles sont adoptées par une majorité d'entre eux (plus de75 %) moins d'un mois après leur sortie. En tant que développeur iOS, il est donc parfaitement acceptable de ne supporter que les deux dernières versions en date du SDK. Dans le courant de l'année 2010 et 2011, on pouvait donc se contenter de supporter iOS 4 et iOS 3. Aujourd'hui, en 2012, vous pouvez vous restreindre à iOS 5 et iOS 4. Cela simplifie considérablement la vie des développeurs.

Configurer les paramètres cibles : le SDK de base et la cible de déploiement

Pour personnaliser les fonctionnalités sur lesquelles s'appuie votre application et les versions de l'OS et les appareils supportés, Xcode vous présente deux types de paramètres à configurer, en fonction de la cible que vous compilez : le SDK de base et la cible de déploiement.

Configurer le paramètre Base SDK

Le paramètre Base SDK constitue le premier élément à modifier. Vous le configurez en modifiant votre cible. Pour cela, effectuez les opérations suivantes :

1. Ouvrez votre projet et sélectionnez le fichier de projet à travers le navigateur.

2. Dans le volet de l'éditeur, sélectionnez la cible puis reportez-vous à l'onglet Build Settings. Le paramètre Base SDK figure généralement à la troisième position, mais vous pouvez directement le retrouver à l'aide de la barre de recherche.

Vous pouvez changer sa valeur à "Latest iOS SDK" (dernière version du SDK d'iOS) ou à toute version du SDK installée sur votre machine de développement. Ce paramètre indique au compilateur la version du SDK à utiliser pour compiler et bâtir votre application : il exerce donc un contrôle direct sur les API disponibles au sein de votre application. Par défaut, les nouveaux projets créés avec Xcode se basent toujours sur la dernière version du kit de développement et Apple gère la rétrocompatibilité avec les API. À moins d'avoir de bonnes raisons, conservez toujours cette valeur par défaut.

Configurer le paramètre Deployment Target

Le second paramètre concerne la cible de déploiement, qui correspond à la version minimale de l'OS nécessaire pour utiliser votre application. Si vous l'assignez à une valeur précise, par exemple 5.0, l'application App Store empêche automatiquement les utilisateurs disposant d'une version antérieure du système d'exploitation de télécharger ou d'installer votre application. Afin de renforcer son audience potentielle, nous vous conseillons d'assurer la compatibilité avec au moins une version antérieure de l'OS. Par exemple, si iOS 5 est la dernière version, vous devriez au moins supporter iOS 4. Vous définissez le paramètre Deployment Target au même onglet que le paramètre Base SDK.

Lorsque vous exploitez une fonctionnalité du SDK d'iOS 5 mais que vous souhaitez malgré tout assurer le support d'anciennes versions, le paramètre Base SDK doit correspondre au dernier SDK (ou iOS 5) et Deployment Target doit valoir iOS 4. Toutefois, lorsque votre application s'exécute sur des appareils iOS 4, certains frameworks et fonctionnalités risquent de manquer à l'appel. Il en va de votre responsabilité, en tant que développeur, d'adapter votre application afin qu'elle fonctionne sans planter.

Considérations sur le support de plusieurs SDK : les frameworks, les classes et les méthodes

Le support de plusieurs versions du SDK présente des conséquences sur trois éléments principaux : les frameworks, les classes et les méthodes. Au cours des sections qui suivent, vous découvrirez toutes les solutions permettant de les gérer au mieux.

Disponibilité des frameworks

Bien souvent, une nouvelle version du SDK introduit un tout nouveau framework, ce qui signifie qu'il sera absent des anciennes versions du système d'exploitation. Le framework `EventKit.framework` introduit avec iOS 4 illustre parfaitement ce cas de figure. Ce framework n'est accessible qu'aux utilisateurs disposant d'iOS 4 et

supérieur. Vous faites face à deux choix. Soit vous assignez la cible de déploiement à iOS 4 et vous ne développez votre application que pour des utilisateurs disposant de cette version du système, soit vous vérifiez si le framework correspondant est bien présent sur le système de l'utilisateur et vous masquez éventuellement les éléments de l'interface qui nécessitent un appel vers ce framework dans le cas contraire. Clairement, cette seconde solution est le meilleur choix.

Lorsque vous utilisez un symbole défini dans un framework qui n'est pas disponible dans les anciennes versions, votre application ne va pas se charger. Pour éviter pareille situation et charger un framework de manière conditionnelle, vous devez l'associer avec un lien faible. Pour cela, ouvrez la fenêtre des paramètres de la cible dans l'éditeur de projet. Rendez-vous ensuite à l'onglet Build Phases et déroulez la quatrième section (*Link Binary With Libraries*). Vous verrez alors une liste des frameworks actuellement liés à votre cible. Si vous n'avez pas changé le moindre paramètre, tous ces frameworks sont marqués comme Required (nécessaires), par défaut. Cliquez sur le menu correspondant et passez la valeur à Optional, afin d'associer un framework avec un lien faible.

Lorsque vous effectuez cette opération, les symboles manquants deviennent automatiquement des pointeurs `null`, ce qui vous permet d'activer ou de désactiver des éléments d'interface au moyen d'une boucle conditionnelle.

Sur iOS 5, `Twitter.Framework` constitue un bon exemple de cette pratique. Lorsque vous utilisez le framework Twitter intégré pour envoyer des tweets, vous avez intérêt à l'associer avec un lien faible, tout en vérifiant au moment de l'exécution s'il est bien disponible. Si ce n'est pas le cas, vous devez afficher vos propres éléments d'interface pour autoriser l'utilisateur à composer des tweets.

INFO

Lorsque vous liez un framework qui n'est présent que sur la toute dernière version du SDK, mais que vous indiquez une cible de déploiement vers un SDK plus ancien, votre application ne se lance pas et plante quasi immédiatement. Elle serait ainsi rejetée par Apple. Lorsque vous recevez un rapport de plantage de la part de l'équipe d'Apple vous indiquant que votre application plante immédiatement à son lancement (sans vous présenter des dumps significatifs), c'est le premier élément à vérifier.

Disponibilité des classes

Parfois, un nouveau SDK introduit de nouvelles classes au sein d'un framework existant. Cela signifie que même si le framework a bien été associé tous les symboles ne sont pas nécessairement accessibles aux anciennes versions du système d'exploitation. Avec iOS 4, la classe `UILocalNotification` d'`UIKit.Framework` en constitue un exemple. Ce framework est lié à toutes les applications iOS et vous devez donc vérifier la présence de cette classe en instanciant un objet à l'aide de la méthode `NSClassFromString`. Si elle retourne `nil`, cela signifie que la classe n'est pas présente sur l'appareil cible. Avec iOS 5, on retrouve le même problème avec le contrôle `UIStepper`. Si vous utilisez cette classe, vérifiez sa présence.

Une autre solution pour vérifier la disponibilité d'une classe consiste à utiliser la méthode `class` au lieu de `NSClassFromString`, comme le montre le code suivant :

```
Vérifier la disponibilité du contrôle UIStepper
if ([UIStepper class]) {
   // On crée une instance et on l'ajoute à la sous-vue
} else {
   // On crée une instance d'un contrôle équivalent
   // et on l'ajoute à la sous-vue
}
```

INFO

Pour utiliser la méthode `class`, vous devez employer le compilateur LLVM Clang et définir la cible de déploiement à 3.1 ou supérieure.

Disponibilité des méthodes

Dans certains cas, de nouvelles méthodes sont ajoutées à une classe existante lors de l'apparition d'un nouveau SDK. Avec iOS 4, un exemple classique concerne le support du multitâches. La classe `UIDevice` dispose d'une méthode baptisée `isMultiTaskingAvailable`. Le code suivant vérifie la présence de cette classe :

```
On vérifie si une méthode est disponible dans une classe
if ([[UIDevice currentDevice]
   respondsToSelector:@selector(isMultitaskingSupported)]) {
   if([UIDevice currentDevice].isMultitaskingSupported) {
      // Le code pour supporter le multitâches
   }
}
```

Pour vérifier si une méthode est disponible dans une classe donnée, utilisez la méthode `respondsToSelector:`. Si elle retourne `YES`, vous êtes libre d'utiliser la méthode que vous avez vérifiée.

Si la méthode que vous vérifiez est une fonction C globale, vous devez la tester avec `NULL`, comme le montre le code suivant :

```
On vérifie la disponibilité d'une fonction C
if (CFunction != NULL) {
   CFunction(a);
}
```

INFO

Vous devez explicitement tester le nom de la fonction avec `NULL`. Vous ne devez pas supposer de manière implicite que les pointeurs valent `nil` ou `NULL`.

Vérifier la disponibilité de frameworks, classes et méthodes

Même s'il est relativement facile de se souvenir de la date d'apparition de chaque framework, il est bien plus complexe de retenir la version minimale requise pour chaque classe et méthode. Il est tout aussi complexe de parcourir les centaines de pages de la documentation officielle d'iOS pour retrouver les caractéristiques d'une méthode. Nous vous conseillons deux solutions pour vérifier la disponibilité d'un framework, d'une classe ou d'une méthode.

Documentation des développeurs

La solution la plus simple pour vérifier la disponibilité de symboles ou de frameworks consiste à vous reporter à la section Availability correspondante, dans la documentation des développeurs. La Figure 12-1 présente une capture d'écran d'un tel cas de figure, afin de vérifier la présence des fonctions de multitâches.

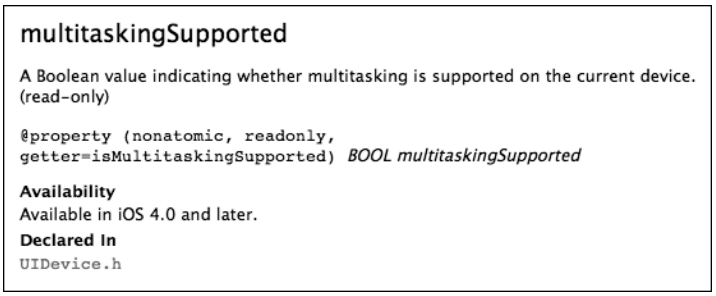

Figure 12-1
On vérifie la disponibilité des fonctions de multitâches dans la documentation des développeurs.

Macros dans les fichiers d'en-tête d'iOS

Une autre solution pour vérifier la disponibilité d'une méthode ou d'une classe consiste à parcourir les fichiers d'en-tête. Cette méthode est encore plus simple : cliquez sur un symbole de votre code source en appuyant sur la touche Cmd, afin d'ouvrir le fichier d'en-tête correspondant dans Xcode. Les méthodes les plus récentes présentent au moins l'une des macros apparaissant à la Figure 12-2.

```
Macros de disponibilité
UIKIT_CLASS_AVAILABLE
__OSX_AVAILABLE_STARTING
__OSX_AVAILABLE_BUT_DEPRECATED
```

```
@property(nonatomic,readonly,getter=isMultitaskingSupported) BOOL multitaskingSupported __OSX_AVAILABLE_STARTING
    (__MAC_NA,__IPHONE_4_0);
```

Figure 12-2
On vérifie la présence du multitâches à travers le fichier d'en-tête.

Cette seconde solution est généralement plus rapide. Mais toutes les méthodes n'ont pas de macros associées : si ce n'est pas le cas, vous devez vous reporter à la documentation des développeurs.

INFO

Si une méthode ne dispose pas de macros, cela signifie probablement que la méthode a été ajoutée depuis longtemps au SDK et que vous n'avez pas à vous préoccuper de sa présence dans les deux derniers SDK en date.

Maintenant que vous savez comment supporter de multiples versions du SDK, concentrons-nous sur l'objectif essentiel de ce chapitre : assurer la compatibilité avec plusieurs types d'appareils. Dans la section suivante, vous découvrirez leurs subtiles différences et apprendrez à vérifier la disponibilité d'une fonctionnalité spécifique. En parallèle, vous développerez une classe d'extension de catégorie sur `UIDevice` qui ajoute des méthodes et des propriétés visant à vérifier des fonctionnalités qui ne sont pas exposées par le framework.

Détecter les capacités de l'appareil

Par le passé, alors qu'il n'existait que deux types d'appareils (l'iPhone et l'iPod touch), l'une des principales erreurs que commettaient les développeurs consistait à détecter le nom du modèle et à vérifier s'il s'agissait d'un "iPhone". Ils se contentaient alors de supposer les capacités accessibles. Si cette technique fonctionnait bien aux balbutiements de la plate-forme, l'apparition de nouveaux appareils et de capteurs inédits a conduit à de nombreux problèmes. Par exemple, la version d'origine de l'iPod touch n'intégrait pas de microphone ; mais, après la mise à jour de l'iPhone OS 2.2, les utilisateurs ont eu la possibilité de connecter un microphone externe ou un micro-casque. Si votre code définit les capacités de chaque appareil en fonction de son nom de modèle, l'application risque de ne pas fonctionner correctement.

Détecter les appareils et supposer leurs capacités

Étudiez le code source suivant, qui présume les fonctions de l'iPhone :

```
Détecter la présence d'un microphone de manière incorrecte
if(![[UIDevice currentDevice].model isEqualToString:@"iPhone"]) {
   UIAlertView *alertView = [[UIAlertView alloc]
                             initWithTitle:@"Erreur"
                             message:@"Microphone absent"
                             delegate:self
                             cancelButtonTitle:@"Masquer"
                             otherButtonTitles: nil];
   [alertView show];
}
```

Le problème avec ce code tient au fait que le développeur a envisagé, de manière vague, que seuls les iPhones disposent d'un microphone. À l'origine, ce code se révélait tout à fait fonctionnel. Mais, avec la mise à jour 2.2 d'iOS, lorsque Apple a ajouté la possibilité de connecter un microphone externe à l'iPod touch, le code précédent empêche ces utilisateurs d'exploiter l'application. Par ailleurs, ce code renvoie une erreur pour tous les appareils introduits par la suite, en particulier l'iPad.

Vous devez utiliser une autre méthode pour détecter la disponibilité de composants ou de capteurs spécifiques et ne pas faire la moindre assomption. Hélas, ces méthodes sont dispersées à travers une grande variété de frameworks. Commençons à bâtir une meilleure solution pour vérifier les capacités d'un appareil, en regroupant les méthodes correspondantes dans une même catégorie de la classe `UIDevice`.

Détecter le hardware et les capteurs

Au lieu de supposer les capacités de chaque appareil, vous devez véritablement vérifier la présence des composants et capteurs précis dont vous avez besoin. Par exemple, au lieu de partir du principe que seuls les iPhones disposent d'un microphone, utilisez des API afin de vérifier la présence réelle d'un microphone. Le code suivant présente le suprême intérêt de fonctionner automatiquement avec tous les prochains appareils, tout en vérifiant la présence de microphones externes.

Un autre atout ? Le code ne tient qu'en une ligne.

La solution correcte pour vérifier la disponibilité du microphone

```
- (BOOL) microphoneAvailable {
   AVAudioSession *session = [AVAudioSession sharedInstance];
   return session.inputIsAvailable;
}
```

Dans le cas du microphone, vous devez également prendre en compte les notifications de changement d'entrée de l'appareil. Ainsi, vous afficherez un bouton d'enregistrement sur votre interface utilisateur lorsque ce dernier branche un microphone, en appelant `viewDidAppear`. Voici la méthode appropriée :

On détecte le branchement d'un microphone

```
void audioInputPropertyListener(void* inClientData,
                                AudioSessionPropertyID inID,
                                UInt32 inDataSize, const void *inData) {
   UInt32 isAvailable = *(UInt32*)inData;
   BOOL micAvailable = (isAvailable > 0);
   // Mettez à jour l'interface utilisateur ici
}

- (void)viewDidLoad {
   [super viewDidLoad];
   AudioSessionAddPropertyListener(
                               kAudioSessionProperty_AudioInputAvailable,
                               audioInputPropertyListener, nil);
}
```

Il vous suffit ici d'ajouter un écouteur sur la propriété `kAudioSessionProperty_AudioInputAvailable` et sur le callback pour vérifier la valeur.

En quelques lignes de code, vous développez ainsi une version correcte du code de détection. Il vous suffit à présent de l'étendre à d'autres types de composants et de capteurs.

INFO

`AudioSessionPropertyListeners` se comporte de la même manière que l'observation des événements `NSNotification`. Lorsque vous ajoutez un écouteur de propriété dans une classe, vous devez le retirer au bon moment. Dans l'exemple précédent, puisque vous avez ajouté cet écouteur dans `viewDidLoad`, vous devez le retirer dans `viewDidUnload`.

Détecter les types de caméras

À l'origine, l'iPhone ne comptait qu'une seule caméra et le capteur frontal n'est apparu qu'avec l'iPhone 4. L'iPod touch n'a été équipé d'un capteur qu'à sa quatrième génération. Et même si l'iPhone 4 intègre un capteur frontal, le premier iPad en était dépourvu. La seconde version de la tablette a directement profité des deux caméras. Cet exemple illustre parfaitement le danger de développer du code qui ne vérifie pas précisément la présence d'un composant. Il est heureusement très simple d'utiliser directement l'API.

La classe `UIImagePickerController` dispose de méthodes qui détectent la disponibilité d'un type de source :

```
On vérifie la présence d'une caméra
- (BOOL) cameraAvailable {
   return [UIImagePickerController isSourceTypeAvailable:
   UIImagePickerControllerSourceTypeCamera];
}

On vérifie la présence d'un capteur frontal
- (BOOL) frontCameraAvailable
{
   #ifdef __IPHONE_4_0
      return [UIImagePickerController isCameraDeviceAvailable:
                                 UIImagePickerControllerCameraDeviceFront];
   #else
      return NO;
   #endif
}
```

Pour détecter la caméra frontale, votre application doit s'exécuter sur iOS 4 ou supérieur. L'énumération `UIImagePickerControllerCameraDeviceFront` n'est en effet apparue qu'avec iOS 4 car tous les appareils qui présentent un capteur frontal (l'iPhone 4 et l'iPad 2) tournent forcément sur cette version du système, ou plus.

Vous pouvez ainsi utiliser une macro et retourner `NO` si l'appareil tourne sous iOS 3 ou inférieur.

Dans le même esprit, vous pouvez vérifier si le capteur est capable d'enregistrer des séquences vidéo. Seuls les capteurs présents sur l'iPhone 3GS et supérieur présentent de telles capacités. La vérification s'effectue à l'aide du code suivant :

```
On vérifie la présence d'un capteur capable d'enregistrer des vidéos
- (BOOL) videoCameraAvailable {
   UIImagePickerController *picker =
      [[UIImagePickerController alloc] init];
   // Appelez tout d'abord la méthode précédente
   // pour vérifier la présence d'une caméra
   if(![self cameraAvailable]) return NO;
   NSArray *sourceTypes =
      [UIImagePickerControlleravailableMediaTypesForSourceType:
         UIImagePickerControllerSourceTypeCamera];

   if (![sourceTypes containsObject:(NSString *)kUTTypeMovie]){
      return NO;
   }

   return YES;
}
```

Vous énumérez ici les types de médias disponibles pour un capteur donné et vérifiez si la liste comprend `kUTTypeMovie`.

Détecter si la photothèque est vide

Si vous utilisez un capteur, vous aurez presque à tous les coups besoin d'accéder à la photothèque de l'utilisateur. Avant d'appeler `UIImagePicker` pour afficher ses albums photo, vous devez vous assurer qu'ils contiennent des clichés. Vous le vérifiez de la même manière que la détection de la présence de la caméra. Contentez-vous de passer `UIImagePickerControllerSourceTypePhotoLibrary` ou `UIImagePickerControllerSourceTypeSavedPhotosAlbums` en guise de type de source.

Détecter la présence d'un flash

Jusqu'à présent, seuls l'iPhone 4 et 4S intègrent un flash. Dans les années à venir, bien d'autres appareils devraient en être équipés. On vérifie sa présence à l'aide d'une méthode de la classe `UIImagePickerController` :

```
On vérifie la présence d'un flash
- (BOOL) cameraFlashAvailable {
   #ifdef __IPHONE_4_0
      return [UIImagePickerController isFlashAvailableForCameraDevice:
         UIImagePickerControllerCameraDeviceRear];
   #else
      return NO;
   #endif
}
```

Détecter un gyroscope

Le gyroscope constitue une nouveauté très intéressante de l'iPhone 4. Il permet aux développeurs de mesurer les changements relatifs à la position physique de l'appareil. En comparaison, un accéléromètre ne peut mesurer que la force exercée : il ne prend pas en compte les mouvements de torsion, par exemple. À l'aide du gyroscope, les développeurs de jeux ont la possibilité d'implémenter des contrôles sur six axes, comme ceux que l'on retrouve dans la manette de la PlayStation 3 de Sony ou de la Wii de Nintendo. Vous pouvez détecter la présence d'un gyroscope à l'aide d'une API disponible dans `CoreMotion.framework`.

Code pour détecter la présence d'un gyroscope

```
- (BOOL) gyroscopeAvailable {
   #ifdef __IPHONE_4_0
      CMMotionManager *motionManager = [[CMMotionManager alloc] init];
      BOOL gyroAvailable = motionManager.gyroAvailable;
      return gyroAvailable;
   #else
      return NO;
   #endif
}
```

INFO

Si le gyroscope soutient l'une des fonctionnalités essentielles de votre application, mais que l'appareil cible s'en trouve dépourvu, vous devez concevoir votre application de telle sorte qu'elle propose des méthodes de saisie alternatives. Vous avez aussi la possibilité de les préciser dans la clé `UIRequiredDeviceCapabilities` du fichier info.plist de votre application, empêchant ainsi les appareils dépourvus de gyroscope de l'installer. Vous découvrirez plus en détail cette clé dans la suite de ce chapitre.

Détecter une boussole ou un magnétomètre

On vérifie la présence d'une boussole à l'aide de classe `CLLocationManager` de `CoreLocation.framework`. Appelez la méthode `headingAvailable` de `CLLocationManager` et, si elle retourne `true`, vous pouvez utiliser la boussole dans votre application. Elle est particulièrement utile dans les applications de géolocalisation et de réalité augmentée.

Détecter un écran Retina

En tant que développeur iOS, vous savez déjà qu'il suffit d'ajouter des images deux fois plus grandes qu'à l'accoutumée pour tirer parti d'un écran Retina. Mais, lorsque vous téléchargez ces images depuis un serveur distant, vous devez sélectionner celles qui présentent une résolution double pour les appareils dotés d'un écran Retina.

Un navigateur de photo, comme un client Flickr ou Instagram, constitue un bon exemple de cette pratique. Lorsqu'un utilisateur exécute l'application sur un iPhone 4 ou 4S (les seuls appareils disposant d'un écran Retina à l'heure où nous mettons sous presse), vous devez télécharger les images présentant une résolution

double par rapport à celles sur un écran non Retina. Certains développeurs ont choisi d'ignorer ce cas de figure et de télécharger systématiquement les versions en haute résolution, quel que soit l'appareil, mais ils gâchent ainsi de la bande passante et pourraient souffrir de ralentissements sur une connexion EDGE. Ne téléchargez donc les fichiers de haute résolution qu'après avoir vérifié si l'appareil dispose d'un écran Retina. Cette vérification est simple :

```
Vérification d'un écran Retina
- (BOOL) retinaDisplayCapable {
   int scale = 1.0;
   UIScreen *screen = [UIScreen mainScreen];
   if([screen respondsToSelector:@selector(scale)])
      scale = screen.scale;
   if(scale == 2.0f) return YES;
   else return NO;
}
```

À travers ce code, vous observez l'objet `mainScreen` de l'appareil et vérifiez s'il est capable d'afficher des images en haute résolution propres aux écrans Retina. De cette manière, si Apple introduit un nouvel appareil doté d'un tel écran ou s'il adapte sa gamme d'iPad, votre application fonctionnera toujours sans le moindre changement.

Détecter la présence d'un vibreur

À l'heure actuelle, seuls les iPhone sont capables de vibrer afin d'alerter l'utilisateur. Hélas, il n'existe pas d'API publique permettant de vérifier la présence d'une telle fonction. Heureusement, `AudioToolbox.framework` dispose de méthodes (présentées ci-dessous) qui ne font vibrer que les iPhone.

```
AudioServicesPlayAlertSound(kSystemSoundID_Vibrate);
AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
```

La première méthode fait vibrer l'iPhone et joue un son sur un iPod touch. La seconde méthode se contente de faire vibrer l'iPhone. Sur tous les appareils dépourvus d'un vibreur, elles n'effectuent aucun traitement. Si vous développez un jeu qui fait vibrer l'appareil pour signaler un danger ou un clone de *Labyrinth* dans lequel les vibrations rythment chaque heurt contre un mur, vous devez utiliser la seconde méthode. La première méthode consiste essentiellement à alerter un utilisateur, notamment à travers l'émission d'un bip.

Détecter la prise en charge du contrôle à distance

Les applications iOS peuvent gérer les événements de contrôle à distance qui se déclenchent lorsque l'on appuie sur des boutons sur le casque externe. Pour les traiter, utilisez la méthode suivante afin de commencer à recevoir les notifications :

```
[[UIApplication sharedApplication] beginReceivingRemoteControlEvents];
```

Implémentez la méthode suivante dans votre `firstResponder` :

```
remoteControlReceivedWithEvent:
```

Assurez-vous de désactiver les notifications lorsque vous n'avez plus besoin de recevoir cet événement, en appelant :

```
[[UIApplication sharedApplication] endReceivingRemoteControlEvents];
```

Détecter les capacités téléphoniques

Vous pouvez savoir si un appareil est capable d'effectuer des appels téléphoniques en vérifiant s'il est en mesure d'ouvrir des URL de type `tel:`. La méthode `canOpenURL:` de la classe `UIApplication` est très pratique pour vérifier si un appareil dispose d'une application capable de gérer les URL d'un type spécifique. Les URL `tel:` sont traitées à l'aide de l'application Téléphone sur un iPhone. Vous pouvez utiliser la même méthode pour vérifier si une application spécifique, capable de traiter une URL précise, est bien installée sur un appareil.

```
Capacités téléphoniques
- (BOOL) canMakePhoneCalls {
    return [[UIApplication sharedApplication]
    canOpenURL:[NSURL URLWithString:@"tel://"]];
}
```

Quelques détails sur l'ergonomie

Les développeurs ont intérêt à masquer les fonctionnalités spécifiques à la téléphonie sur les iPod touch. Par exemple, si vous développez une application d'annuaire qui dresse la liste de numéros de téléphone à partir d'un service web, n'affichez le bouton permettant de passer un appel que sur les appareils capables de l'effectuer. Ne vous contentez pas seulement de le désactiver (rien ne permettra alors à l'utilisateur de le réactiver) ou affichez une alerte d'erreur. On recense des cas où l'affichage de l'erreur "Ce n'est pas un iPhone" sur un iPod touch conduisait au rejet de l'application par l'équipe d'Apple.

Composition d'e-mails et de SMS au sein d'une application

Même si *In App email* et *In App SMS* ne sont pas techniquement des capteurs ou des éléments matériels, tous les appareils ne sont pas capables d'envoyer des e-mails ou des SMS. Les iPhone n'y échappent pas nécessairement, même ceux qui tournent sous iOS 4 ou supérieur. Même si les contrôleurs `MFMessageViewController` et `MFMailComposeViewController` sont apparus avec iOS 4, et même si votre application définit iOS 4 en tant que cible de déploiement, vous devez connaître les principaux pièges de ces classes.

Par exemple, un appareil iOS sur lequel aucun compte e-mail n'a été configuré ne peut pas envoyer de messages, même s'il en est techniquement capable. Il en va de même pour les SMS/MMS : un iPhone dépourvu de carte SIM ne peut pas envoyer

de textos. Gardez ces détails à l'esprit et vérifiez les capacités de chaque appareil avant d'utiliser ces fonctionnalités.

Cette opération est heureusement facile. Les classes `MFMessageComposeViewController` (pour In App SMS) et `MFMailComposeViewController` (pour In App e-mail) disposent des méthodes `canSendText` et `canSendMail`, qui le vérifient très simplement.

Vérifier le support du multitâches

Il est très simple de vérifier si un appareil supporte le multitâches. Comme vous l'avez vu précédemment dans ce chapitre, vous devez vérifier si la méthode `isMultitaskingSupported` est disponible, comme le montre le code suivant. Si elle retourne `YES`, vous pouvez développer du code relatif aux techniques multitâches. Dans le cas contraire, vous devez enregistrer l'état de votre application lorsqu'on la quitte et le restaurer lorsqu'on la lance à nouveau.

```
Le multitâches est-il supporté ?
if ([[UIDevice currentDevice] respondsToSelector:
   @selector(isMultitaskingSupported)]) {
   if([UIDevice currentDevice].isMultitaskingSupported) {
      // Code lié au multitâches
   }
}
```

Mais ce n'est pas tout ! Sur les appareils ne supportant pas le multitâches, le délégué de votre application ne recevra pas les callbacks suivants :

```
- applicationDidEnterBackground:
- applicationWillEnterForeground:
```

Tout code de démarrage et d'initialisation que vous avez écrit dans la méthode `applicationWillEnterForeground:` doit aussi figurer dans `applicationDidFinishLaunchingWithOptions:`, pour être pris en charge sur les appareils ne supportant pas le multitâches.

Dans le même esprit, le code de fin (notamment les méthodes d'enregistrement du contexte Core Data) que vous avez développé dans `applicationDidEnterBackground:` doit aussi être inscrit dans `applicationWillTerminate:`.

Développer la catégorie UIDevice+Additions

Les extraits de code que l'on a parcourus jusqu'à présent sont disponibles sous forme d'une catégorie complémentaire d'`UIDevice`. Vous la téléchargerez sur le site web officiel de cet ouvrage.

Elle s'articule autour de deux fichiers : `UIDevice+Additions.h` et `UIDevice+Additions.m`. Vous devez associer les frameworks nécessaires à vos projets afin d'éviter les erreurs de compilation, dans la mesure où cette classe est liée à plusieurs

bibliothèques d'Apple. Mais ne vous inquiétez pas : ils se chargent dynamiquement afin ne pas engorger votre application.

UIRequiredDeviceCapabilities

Jusqu'à présent, vous avez appris à vérifier de manière conditionnelle si un appareil dispose de capacités spécifiques, afin de les utiliser le cas échéant. Dans certains cas, votre application dépend exclusivement de la présence d'un composant particulier, sans quoi elle ne fonctionnera pas. C'est notamment le cas des applications photographiques, comme Instagram ou Camera+. Les fonctionnalités au cœur de ces applications ne seront pas accessibles sans la présence d'une caméra. Dans ce cas, vous devez aller plus loin que vérifier les capacités de l'appareil et masquer des parties spécifiques de votre application. Les appareils dépourvus de caméra ne devraient pas être en mesure de télécharger et d'installer votre application.

Apple offre une solution idéale pour contourner ce type de problème, à travers la clé `UIRequiredDeviceCapabilities` du fichier Info.plist. Cette clé supporte les valeurs suivantes : `telephony`, `wifi`, `sms`, `still-camera`, `auto-focuscamera`, `front-facing-camera`, `camera-flash`, `video-camera`, `accelerometer`, `gyroscope`, `location-services`, `gps`, `magnetometer`, `gamekit`, `opengles-1`, `opengles-2`, `armv6`, `armv7`, `peer-peer`.

Vous pouvez explicitement exiger une capacité spécifique et empêcher l'installation de votre application sur les appareils qui en sont dépourvus. Par exemple, vous pouvez empêcher vos applications de s'exécuter sur des appareils disposant d'une caméra en assignant la clé `video-camera` à `NO`. À l'inverse, vous pouvez exiger la présence de `video-camera` en assignant cette clé à `YES`.

INFO

Apple ne vous permet pas de soumettre la mise à jour d'une application existante qui empêche son exécution sur un appareil spécifique qui était supporté jusqu'alors. Par exemple, si votre application supportait l'iPhone et l'iPod touch dans sa version 1.0, vous ne pouvez pas soumettre une mise à jour qui empêche son fonctionnement sur l'un ou l'autre de ces appareils par la suite. Dans le même ordre d'idée, vous ne pouvez pas exiger la présence d'un composant spécifique au cours du cycle de vie de votre application. Le processus de soumission sur iTunes Connect va alors échouer et vous renvoyer une erreur. L'inverse est toutefois admis. Ainsi, si vous aviez exclus un appareil par le passé, vous pouvez l'autoriser à nouveau par la suite. En d'autres termes, si la version 1 de votre application ne supportait que l'iPhone, vous pouvez soumettre une version 2 compatible avec l'ensemble des appareils.

En ajoutant des valeurs à la clé `UIRequiredDeviceCapabilities`, vous empêchez l'installation de votre application sur les appareils dépourvus des capacités que vous exigez. Si vous indiquez que votre application nécessite les fonctions de téléphonie, les utilisateurs disposant d'un iPod touch ou d'un iPad ne pourront pas la télécharger. Assurez-vous que c'est bien le comportement que vous attendez avant d'utiliser cette clé.

En résumé

Au cours de ce chapitre, vous avez découvert de nombreuses techniques et astuces pour vous assurer que votre application s'exécute correctement sur de multiples plates-formes. Nous nous sommes attardés sur l'ensemble des composants et capteurs disponibles auprès des développeurs iOS, en apprenant à détecter leur présence de manière adéquate. Vous avez ainsi développé progressivement une extension de catégorie sur `UIDevice`, que vous utiliserez pour détecter rapidement les capacités d'un appareil. Enfin, vous avez découvert la clé `UIRequiredDeviceCapabilities`, qui permet d'exclure des appareils dépourvus de composants spécifiques. Nous vous conseillons toutefois de dépendre essentiellement des méthodes détaillées dans ce chapitre et d'utiliser la clé `UIRequiredDeviceCapabilities` avec parcimonie.

Pour aller plus loin

Documentation Apple

Les documents suivants sont disponibles dans la bibliothèque des développeurs iOS, à l'adresse **developer.apple.com**, ou à travers la documentation de Xcode et la table de référence de l'API.

- *Comprendre la clé UIRequiredDeviceCapabilities*
- *Détails sur la configuration des builds iOS*

Autre ressources

- MKBlog, "Tutorial iPhone : une meilleure solution de vérifier les capacités des appareils iOS".

  http://blog.mugunthkumar.com/coding/iphone-tutorial-better-way-to-checkcapabilities-of-ios-devices/

- Github, "MugunthKumar/DeviceHelper".

  https://github.com/MugunthKumar/DeviceHelper