Supporting Multiple Platforms in a Single Codebase

- Embrace the platform
- Cross-platform mechanics
    - Platform identification
    - Platform-specific APIs and properties
    - Platform-specific resources
- Strategies and recommendations
    - Branching
    - Platform-specific JS files
    - References and Further Reading

Objective

In this section, you will explore the ways you can support both Android and iOS within a single Titanium project. Through a combination of platform specific feature abstraction and usage of conditional code branching, Titanium provides you with a powerful mechanism for creating cross platform, native mobile apps with a maximum amount of code reuse.

Contents

Titanium is not a write once, run anywhere framework. It's more aptly referred to as a "*write once, adapt everywhere"* framework. Your business logic, networking, database, and event handling logic will be close to 100% cross-platform compatible. The user interfaces on iOS and Android differ so significantly that in most cases you'll have to do at least a little platform specific coding. That said, it's not uncommon for cross platform apps to reuse 80, 90, or even 100% of their UI code as well.

Embrace the platform

Best of breed, native apps take full advantage of the platforms on which they run. Your Titanium apps should do more than just run on iOS and Android. When running on an iOS device, your app should feel like an iOS app. Your Android app should feel like an Android app. By this, we mean apps that:

- Follow platform UI conventions, such as tabs at the top (Android) or bottom (iOS).
- Use hardware-specific features, such as the Android Menu button.
- Use OS-specific controls, such left and right navigation buttons in title bars on iOS.
- Participate in the platform ecosystem, such as using platform-appropriate notification mechanisms.

The best approach when creating cross-platform apps is to develop and test for both iOS and Android right from the start. Designing and developing your app with multiple platforms in mind right away will be significantly more efficient than developing for one, then porting to the next.

Cross-platform mechanics

Before we get into the strategies you should adopt, let's look at the mechanics of handling cross-platform coding within Titanium. This includes:

- Platform identification
- Recognizing platform-specific APIs and properties
- Handling platform-specific resources

Platform identification

Titanium provides platform-identification properties in the `Ti.Platform` namespace that you can use for conditional branching within your code. These are:

Property	Description	Sample values
`Ti.Platform.name`	Returns the name of the platform returned by the device	`iPhone OS` for iPhone or iPod, `android` for Android, returns the `navigator.userAgent` string on Mobile Web
`Ti.Platform.osname`	Returns an abbreviated identifier of the platform	`iphone` for iPhone or iPod, `ipad` for iPad, `android` for Android, and `mobileweb` on Mobile Web
`Ti.Platform.model`	Returns device model identifier	`iPhone 3GS` or `iPod Touch 2G` or `Droid` (unsupported on Mobile Web)

Platform-specific APIs and properties

Many of the Titanium APIs are separated according to the platform on which they are supported. For example, the Ti.UI.iPhone namespace

includes user interface components that are supported on only on the iOS operating system. The same is true of the Ti.UI.iOS, Ti.UI.iPad, and Ti.UI.Android namespaces. By segmenting such platform-specific functionality, we help make it clear what will and won't work on the various platforms.

As you explore the API docs, you will find also many platform-specific properties and values. For example, the Ti.UI.Window object has an Android-specific property called `softInputMode`. That property's value must be one of the constants in the Ti.UI.Android namespace. These platform-specific properties are labeled as such. To avoid crashing and errors, don't try to use them on other platforms.

Another concern is platform-specific constants. Explore the API docs and you'll see lots of constants that define button appearance, media file type, and so forth. When these are platform specific, they are generally put into their own sub-namespace. (If they're not in a platform-specific namespace, expect that we'll be moving them there as we clear up platform parity issues.) For example, Titanium.UI.iOS.ANIMATION_CURVE_EASE_IN defines an iOS-specific animation property. Don't use one platform's constants on another platform or your code will throw an error.

Platform-specific resources

Titanium gives you various ways to include platform-specific resources, like images, stylesheets, and scripts, in your project. Titanium uses an "overrides" system to make it easy to use platform-specific resources. Any file in the platform-specific Resources directories (`Resources/android`, `Resources/iphone`, or `Resources/mobileweb`) will override, or be used in place of, those in the `Resources` directory. You don't have to use any special notation in your code to specify that these files should be used.

```
var img = Ti.UI.createImageView({
 image: 'logo.png'
 /* Resources/android/logo.png or Resources/iphone/logo.png  or
 *  Resources/mobileweb/logo.png will be used automatically if they exist
 *  when you build for those platforms, respectively
 */
});
```

You can maintain a hierarchy of folders within the Resources or platform-specific folder. For example, let's say you put most of your images into the `Resources/images` folder. To include platform-specific overrides, you would duplicate that folder hierarchy in the platform folders. Thus, you'd need to put the images into the `Resources/android/images`, `Resources/iphone/images`, and `Resources/mobileweb/images` folders.

Here's an example that shows this feature in action with a CommonJS require() module. The code simply calls the base name of the file. Titanium grabs the platform-specific version at build time, which you can see in the Android emulator, iPhone simulator, and Chrome running the Mobile Web preview.

Strategies and recommendations

Branching

Branching in code is useful when your code will be *mostly the same* across platforms, but vary here and there. Long blocks of `if...then` code are difficult to read and maintain. Also, excessive branching will slow your app's execution. If you must use this technique, try to group as much code as you can within a branch and defer loading as much as possible to mitigate the performance penalty of branching.

It's best practice to query the platform value once, then store it in a globally accessible variable. Each time you request one of those properties, Titanium has to query the operating system for the value. This "trip across the bridge" takes a few cycles and if used too frequently could possibly slow your program. Something like the following would be more efficient:

```
// create a JavaScript alias to the platform-specific property
var osname = Ti.Platform.osname;
// Booleans identifying the platforms are handy too
var isAndroid = (osname=='android') ? true : false;

if (isAndroid) {
 // do Android specific stuff
} else {
 // do iOS, mobileweb, or other platform stuff
}
```

You can use JavaScript's ternary operator when you need to branch on a specific property, like this:

```
var isAndroid = (Ti.Platform.osname=='android) ? true : false;
var win = Ti.UI.createWindow({
 softInputMode: (isAndroid) ? Ti.UI.Android.SOFT_INPUT_ADJUST_PAN : null
});
```

Keep in mind the growing list of supported platforms and don't fall prey to coding in an if/else relationship that won't support new platforms. For example, don't do the following:

Anti-pattern!

```
var osname = Ti.Platform.osname;
if (osname != 'android') {
   // don't assume this means iOS! It could be mobile web or some future-supported
platform.
}
```

Platform-specific JS files

Using platform-specific JS files is likely to be most useful when your code is *mostly different* across platforms. This removes long if...then blocks from your main code. Separating platform-specific code reduces the chances of an error that comes from accidentally using the wrong platform's API or property. However, you'll have to remember to apply changes and fixes to *each* of the platform-specific files. So this approach could increase your work rather than reduce it.

```
var label2 = require('ui').label;
// will include /android/ui.js on Android
// and /iphone/ui.js on iOS
// there doesn't even need to be a /ui.js file!
```

References and Further Reading

- Ti.Platform reference
- Android UI scaling

Summary

In this section, you learned how to support both Android and iOS within a single set of files. You learned programming strategies as well as the ways Titanium eases the process of working with platform-specific resources. In the final section of this chapter, we'll see how we can use Titanium's integrated internationalization capabilities to make out app's global accessible.