Oracle ADF Mobile

Device Convergence – As simple as 1-2-3

Chris Muir & Frank Nimphius
Oracle Corporation

keywords

ADF, ADF Mobile, Device Integration

Introduction

A very important but often forgotten benefit of the smartphone/tablet world we now live in is that of "device convergence". It's the convenience of carrying around a single device, an iPhone or Android phone for example which does everything in a small form factor that used to require multiple devices. No need to carry around a camera, a GPS, an address book, eBooks, watches, the list goes on. Today your phone and tablet has all of these features built in, for significantly little money.

As consumers we've latched onto these features like it's a new world order. Be it using the device's GPS and Google Maps for finding our way, taking pictures of our kids and emailing them to our family, or one of a thousand other opportunities these "smart" devices afford our personal lives, they certainly have changed our lives in a very small time period indeed, less than 10 years.

Yet in the enterprise space the efficiencies offered by a digital workforce enabled with smart devices having access to all these features has barely yet been realized. We stand on the edge of a revolution in the workforce where warehouse staff are directed to stock by the GPS on their phone rather than printing out a stock registry from the central computer, shop assistants no longer need to take items back to the till for a price check but rather take a photo of the barcode with their phone to tell you the price, country fire fighters watch real-time video telemetry on their tablets from drones circling out of control fires. No longer is the workforce going to be tied to a PC on a desk in some office away from where the real action is.

Yet a distinct challenge for enterprise IT staff on utilizing these device services is the disparate mobile platforms. Developing for Apple's iOS Xcode platform is completely different from Google's Android's offerings. There's different APIs, different programming languages, different skill sets in building for each platform which frankly, most enterprises don't have the time to specialize in, or can afford separate teams to write custom solutions for, whose IT department has an expanding budget these days? Over the last 20 years as an industry we've been converging on a single standardized web platform driven by HTML and CSS, but all this was thrown out the window with the success of the mobile market.

So we have these great opportunities for efficiency to our enterprise, we can see our businesses will radically change (and most definitely our competitors will if we don't), yet the mobile world is also going to hit our IT departments with radical and expensive cost to support the diverged mobile market. Or will it?

Oracle ADF Mobile

Enter Oracle's ADF Mobile, a relatively new 2012 offering from Oracle that not only allows developers to write a single Java code base to run natively on both iOS and Android, not only allows you to use a skill set that you already have in your organization, that of Java and Oracle's ADF platform, but also allows you to access the disparate device features through a single, standard API. Oracle is taking the ideals of Java, the concept of write once deploy anywhere to the mobile world, something the big mobile vendors would rather you didn't do but rather write a custom solution for their own platform. Where's the efficiency in that?

If you'd like to find out more about Oracle's ADF Mobile platform please visit: http://bit.ly/adfmobile

In investigating ADF Mobile and the device service integration it's worthwhile understanding ADF Mobile provides developers the ability to create AMX or custom HTML pages that run on the device.

An AMX page is a mobile page where developers use predefined advanced UI components provided by ADF Mobile beyond regular HTML components, specifically enhanced for the mobile user experience. Supporting touch gestures, compatible and supported by Oracle across mobile platforms, designed to be shown with a native look and feel for each mobile OS, for most ADF Mobile developments AMX pages are the best choice.

ADF Mobile also supports developers wanting to write their mobile pages from the ground up using HTML, CSS and JavaScript. This option provides the most customized development experience, though, it comes with the overhead of writing considerable code to provide the same functionality of AMX pages and the predefined AMX components.

In returning to the article's purpose in describing how developers access the device features, it's worthwhile understanding that both page types have access to the *mini-JVM* that runs on the device to access Java services, or, can integrate with JavaScript too. So it's from here that your ADF Mobile applications would typically access any APIs and programmatic services. Let's explore this further.

API Convergence

Of significant importance to ADF Mobile and you as a developer is how ADF Mobile provides a single unified API for accessing the device's features such as the GPS, camera, contacts, email and SMS without having to learn the nuances of each mobile platform.

The secret sauce in ADF Mobile's mix is that of Apache Cordova (formerly known as PhoneGap), "a set of device APIs that allow a mobile app developer to access native device function such as the camera or GPS from JavaScript." (Source: http://cordova.apache.org) While Oracle could have written its own solution in these regards, Cordova is such a comprehensive and rapidly improving open source product that it only made sense to adopt it. Figure 1 shows where Cordova fits into the overall architecture of ADF Mobile:

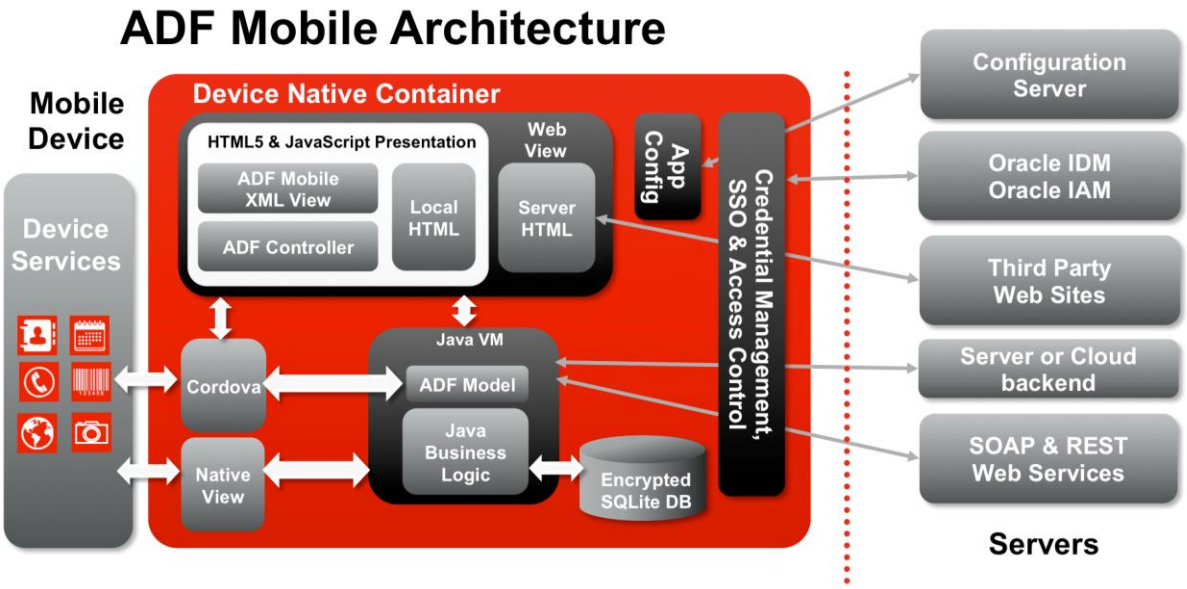


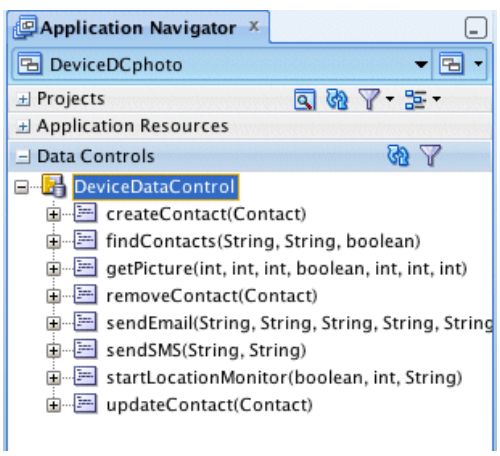
Figure 1 ADF Mobile Architecture

Yet Oracle has taken the solution further by providing programmers who don't want to deal with the APIs at all a set of drag 'n' drop services that will instantly wire up a button to take picture from the phone for example, also a set of Cordova-wrapping Java APIs for ADF Mobile programmers who don't want to call the JavaScript libraries but want the flexibility of invoking services via Java, and even a set of Cordova-wrapping JavaScript APIs for ADF Mobile programmers who don't want to deal with the changing Cordova APIs between versions. Depending on your needs as a developer, where you want to wire up these services with as little programming as possible, or you want the flexibility and sophistication to access and control these services programmatically, Oracle's ADF Mobile has you covered.

For the purposes of the article we'll keep it simple and look at JDeveloper's drag 'n' drop service to wire up the device features, as well as a quick look at the Java implementation to show how easy it is. We'll leave the JavaScript option for developers with a keen interest in 100% customized solutions to explore elsewhere, it's interesting but an edge case most ADF Mobile developers wont bother with.

Device Features Data Control – as Simple as 1-2-3

One of productivity boosters JDeveloper gives to developers is that of the Data Control Palette and the services it provides. Where traditional programmers have to code everything by hand, in many cases JDeveloper provides an easier declarative approach to making use of such services. The trick to this is Oracle identifies the activities that programmers will be doing again and again, and turns these into a common function and page controls which can be accessed via JDeveloper's Data Control Palette. Want to invoke a web service with a button and display the output? Check. Want to show the contents of an array in a scrollable list? Check. Want to take a picture with the device's camera? Check.



For ADF Mobile these device features are accessed via the DeviceDataControl as can be seen in Figure 2. It provides access to numerous services of the mobile device including the camera, location (GPS), contacts, emails and SMS. These can be simply dragged 'n' dropped onto an ADF Mobile AMX page and JDeveloper will do the rest.

It's a little hard to demonstrate this in a document rather than a video, but to say following this articles concept of "as-easy-as-1-2-3" the end result is the ADF Mobile page, the button on the page to call the service, and the service parameters are all configured on behalf of the programmer with 0 lines of code to write. If you look to the end of this document you can follow a link to a number of YouTube videos that show how this is easily done.

Figure 2 Data Control Palette

Java API

Beyond the Data Control drag 'n' drop approach to wiring up pages in a declarative fashion, with nearly no programming involved, ADF Mobile also allows developers who want more programmatic control to call the device services via a set of Java APIs. Admittedly some readers may shun away here from reading the Java code it is worthwhile having a look at how easy this really is.

The `DeviceManager` object, packaged in `oracle.adf.model.datacontrols.device` and accessed via a call to the `DeviceManagerFactory.getDeviceManager()` method provides the Java API wrapper for calling the Cordova JavaScript libraries and associated device services under the covers. The following code sample shows an abbreviated JavaDoc for calling the `getPicture()` method to invoke the camera, and what the individual parameters do:

```
01 public class oracle.adf.model.datacontrols.device.DeviceManager {
02
03 /** Provides access to the device's default camera application, which enables
04   * taking a picture or retrieving a previously-save image.
05   *
06   * @param quality         - quality of save image. Range 0 to 100
07   * @param destinationType - Choose the format of the return value.
08   *                        - 0 - base64 encoded string - 1- URI to file:///
09   * @param sourceType      - Where should the picture come from?
10   *                        - 0 - Library - 1 - Camera - 2 - Album
11   * @param allowEdit       - Allow simple editing of image before selection
12   * @param encodingType    - Encoding of saved imaged - 0 - JPG - 1 - PNG
13   * @param targetWidth     - Width in pixels to scale image
14   * @param targetHeight    - Height in pixels to scale image
15   * @returns               - A String of either the base64 image or a file URI
16   */
17   public String getPicture(int quality,
18                            int destinationType,
19                            int sourceType,
20                            boolean allowEdit,
21                            int encodingType,
22                            int targetWidth,
23                            int targetHeight);
```

As can be seen the APIs are fairly easy to use, and can be wired up to a button via a managed bean in the following example where the `takePic()` method in the `MyBean` includes a call to `getPicture()`:

```
01 package view;
02
03 import oracle.adf.model.datacontrols.device.DeviceManager;
04
05 public class MyBean {
06
```

```
07   // Called via a button
08   public void takePic(ActionEvent event) {
09     DeviceManager device = DeviceManagerFactory.getDeviceManager();
10
11     String imageData =
12       device.getPicture(100,
13         DeviceManager.CAMERA_DESTINATIONTYPE_FILE_URL
14         DeviceManager.CAMERA_SOURCETYPE_PHOTOLIBRARY,
15         false,
16         DeviceManager.CAMERA_ENCODINGTYPE_JPEG,
17         0,
18         0);
19   }
20 }
```

With the resulting button call to the `takePic()` method:

```
01 <?xml version="1.0" encoding="UTF-8" ?>
02 <amx:view xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03           xmlns:amx="http://xmlns.oracle.com/adf/mf/amx">
04  <amx:panelPage id="pp1">
05   <amx:commandButton text="Take Pic" actionListener="#{myBean.takePic}"/>
06 </amx:panelPage>
07 </amx:view>
```

Admittedly these examples are rather simplistic and don't necessarily reflect real world situations when running on a smartphone or tablet. For example how do we handle situations when the device doesn't actually have a camera? What can we do to handle this situation?

The `DeviceManager` class also provides a number of methods such as `hasCamera()` which returns a Boolean value indicating if the device actually has a camera to make use of. The previous `MyBean.takePic()` picture can now be easily modified:

```
08   public void takePic(ActionEvent event) {
09     DeviceManager device = DeviceManagerFactory.getDeviceManager();
10
11     if (device.hasCamera() {
12       String imageData =
13         device.getPicture(100,
14           DeviceManager.CAMERA_DESTINATIONTYPE_FILE_URL
15           DeviceManager.CAMERA_SOURCETYPE_PHOTOLIBRARY,
16           false,
17           DeviceManager.CAMERA_ENCODINGTYPE_JPEG,
18           0,
19           0);
20     } else {
21       // Do something else
22     }
23   }
```

Better yet the AMX pages also have access to an implicit EL object `deviceScope` that provides access to the same methods. As such we can modify our original page code so the user cannot invoke the button in the first place:

```
01 <?xml version="1.0" encoding="UTF-8" ?>
02 <amx:view xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
03           xmlns:amx="http://xmlns.oracle.com/adf/mf/amx">
04  <amx:panelPage id="pp1">
05   <amx:commandButton text="Take Pic" actionListener="#{myBean.takePic}"
06                      disabled="#{deviceScope.device.hasCamera}"/>
06 </amx:panelPage>
07 </amx:view>
```

How To Learn More

The examples provided in this article are deliberately brief in order to show you the different mechanisms to access the device features rather than comprehensively looking through every API and all their methods.

If you're interested in a more in-depth look at integrating different device features into ADF Mobile, Oracle's ADF Product Management team has provided five YouTube videos on the ADF Insider Essentials channel for you have a closer look & learn more. All five videos are accessible via the following playlist: http://bit.ly/1cYZc6s

Conclusion

For enterprises the power of mobile devices is just not their mobility and connectivity, but the wealth of services they provide to application's running on the device. Oracle's ADF Mobile goes beyond the proprietary solutions provide by Apple and Google, and provides a cross-platform solution that runs on both iOS and Android with a single code base. In turn the APIs for accessing these services has been standardized leaving the nuances of each vendor's mobile platforms behind, as well as providing a set of APIs that are familiar to most Oracle customers', that of the Java platform. It really is as easy as 1-2-3.

Contact :

Name: Chris Muir & Frank Nimphius

Firma: Oracle Corporation

Internet: www.oracle.com