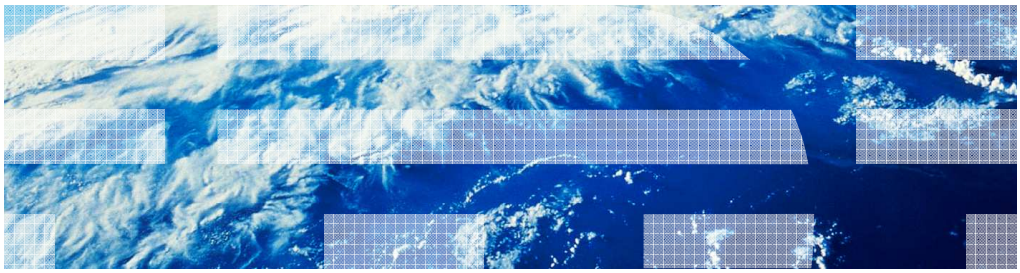
WebSphere Commerce V7 Feature Pack 4

Mobile application architecture



WebSphere software

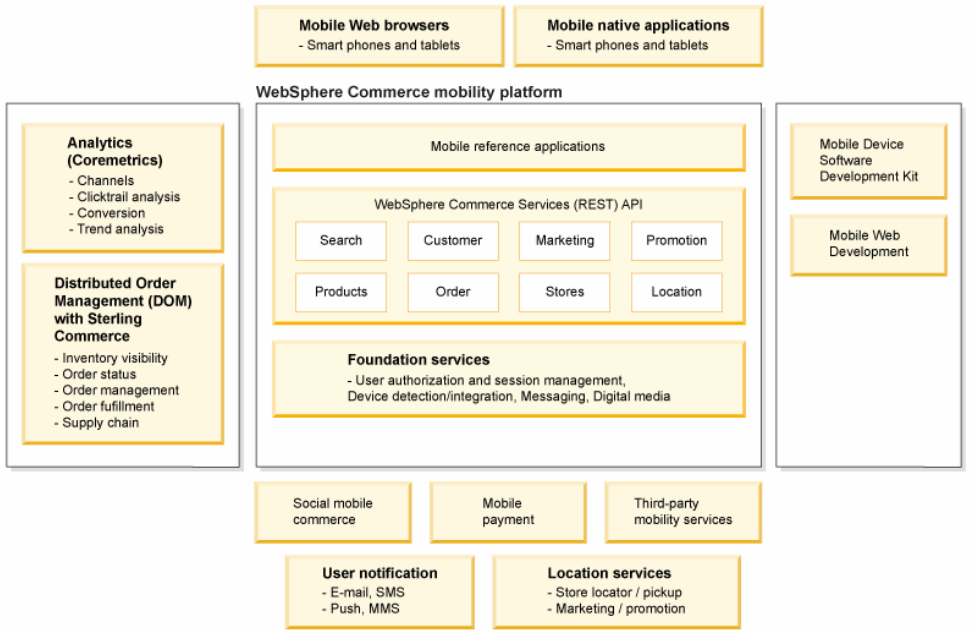
This presentation provides an introduction to the architecture of the device-specific applications for mobile commerce. If you are not familiar with the mobile commerce solution in Feature Pack 4, you should view the mobile solution overview before beginning this presentation.

Table of contents

This presentation begins with a brief overview of the mobile commerce solution for Feature Pack 4. It then focuses on the solution architecture, particularly for the device-specific applications. Following the architecture is a short discussion on how to build Android applications. The presentation concludes with some problem determination tips.

This section provides an overview of the mobile commerce solution.

Mobile solution overview for Feature Pack 4



This chart summarizes the mobile solution in Feature Pack 4. The top of the diagram shows the various ways smart phones and tablets can access a WebSphere Commerce store. New store layouts are provided for viewing the store from a smart phone or tablet browser. There are also device-specific sample applications for Android and iOS devices. Collectively, the mobile web stores and applications are referred to as the mobile reference applications. Similar to the traditional starter stores, the device-specific applications make use of services to access the WebSphere Commerce application. A new REST service layer has been added that is used by the mobile native applications. The web-based mobile stores use the existing OAGIS services through getData tags. In some cases data beans are still being used.

The left side of the diagram lists WebSphere Commerce capabilities that can be added to the mobile reference applications such as analytics and distributed order management (DOM) integration. For example, cross channel inventory availability and orders submitted through the mobile channel can be processed by your back-end DOM system. The right side of the diagram lists the technologies used in building the mobile reference applications. Finally, the bottom of the diagram highlights other integrations that are possible with this solution such as user notification capabilities, social mobile commerce, mobile payment, location services and other vendor mobility services.

Mobile solution terminology

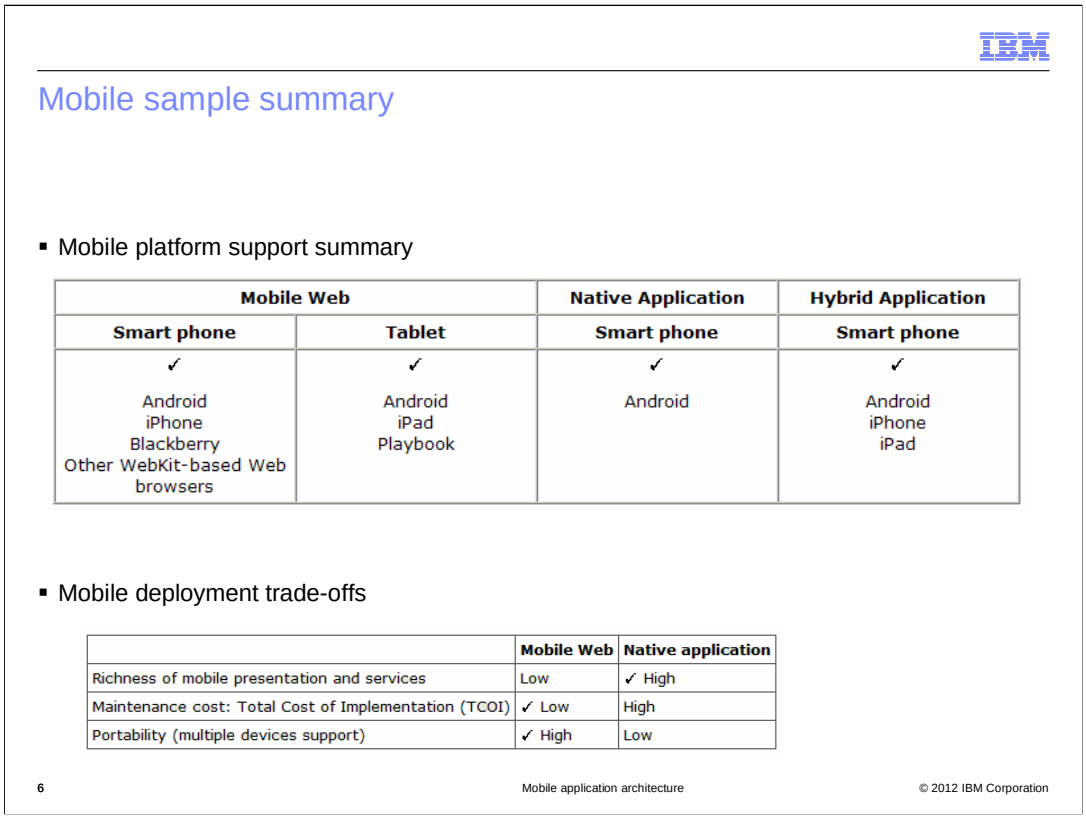
- Mobile web
  - Smart phone
  - Tablet
- Native application
- Hybrid application

The terms shown on this slide are used throughout the presentation to describe different pieces of the mobile commerce solution. Mobile web refers to any store that is accessed through the web browser on a mobile device. The mobile device can be either a smart phone or a tablet.

A native application is device-specific and is downloaded and installed from an application store or marketplace. Once installed, the application provides direct access to the online store. It also allows access to other device features such as the address book or camera. A native application is built to run on a single platform. It uses REST services to communicate with WebSphere Commerce.

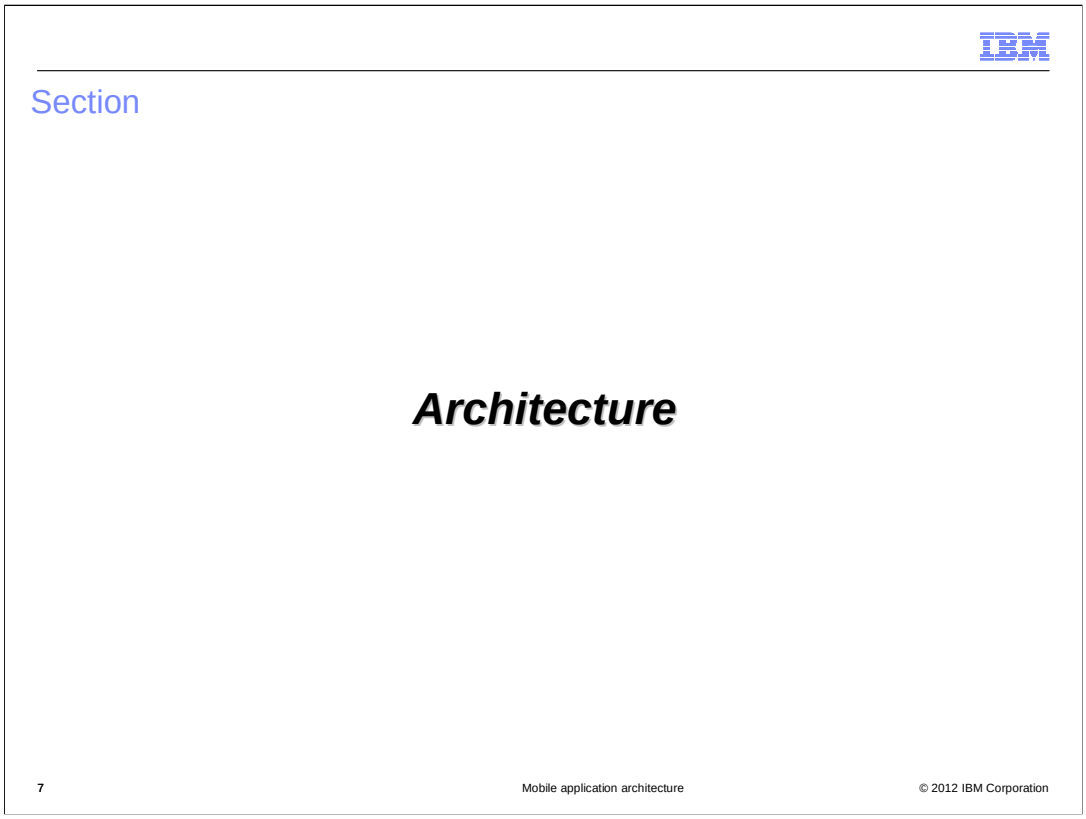
A hybrid application shares the same characteristics as a native application with one important difference. Instead of being purely device-specific, the storefront browsing is done through a WebKit browser embedded within the application. This allows the hybrid application to access device features such as the address book or camera but it can also reuse the JSP pages built for the mobile web application.

These concepts are described further in the next section.



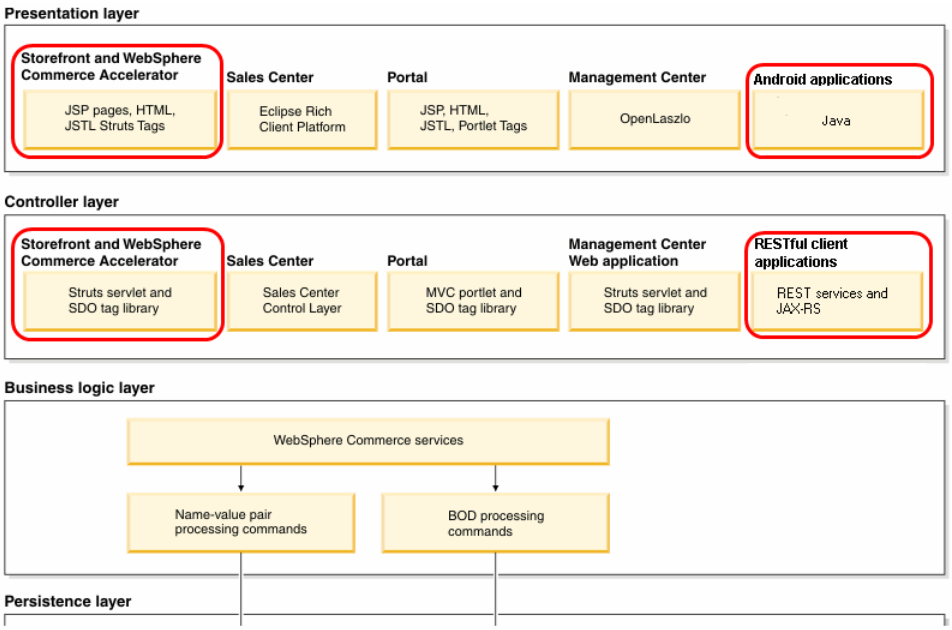
This slide summarizes the type of mobile reference applications available for each device type. The mobile web stores for smart phones and tablets are available for the widest range of devices. The web applications require WebKit based browser support such as is available in iOS, Android, and Blackberry 6 devices. For older smart phones, the mobile web store released in WebSphere Commerce V7 can still be used. Two types of device-specific applications are available for mobile devices. The native application is available only for Android phones. A hybrid application is available for both Android phones and iOS devices such as iPhones and iPads. The iOS hybrid application is available as a separate download. It is not included with Feature Pack 4.

There are several factors to consider when choosing which mobile solution to deploy for your store. The fully native application provides the richest experience for the shopper but results in higher maintenance costs and is not portable across devices. The mobile browser solution provides the best portability and lowers maintenance costs but lacks the richness of a purpose-built application. The hybrid application helps to balance these two extremes. The native application frame provides the application experience and the use of the built in browser to display most of the store content means the bulk of the functionality is portable and centrally maintained.



This section covers the mobile commerce solution architecture.

Architecture for mobile solutions



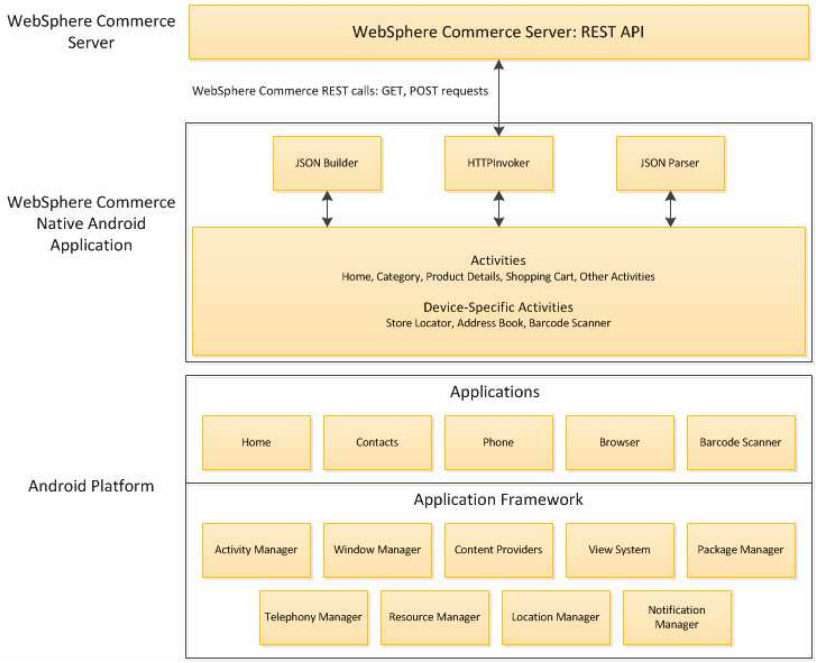
The architecture for the mobile solutions falls into two categories. The browser-based mobile web stores and the browser-based portion of the hybrid application use the existing JSP and Struts architecture common to the Madisons desktop store. The native Android application and the native shell of the hybrid application are built using Java. The purely native Android application uses the new REST services to control communication with the WebSphere Commerce server.

Smart phone and tablet web stores

- Standard storefront customization
  - JSP
  - JavaScript
  - CSS
  - Dojo widgets
- Smartphone web files
  - Stores\WebContent\Madisons\mobile20
- Tablet web files
  - Stores\WebContent\Madisons\tablet

Extending the smart phone and tablet web stores follows the same customization methods as other starter stores. The tablet store uses some Dojo mobile widgets that are part of the Dojo 1.7 release. These widgets provide a more native user experience for shoppers. Documentation on the widgets used can be found in the WebSphere Commerce Information Center. The JSP, JavaScript and CSS files for the smart phone and tablet stores can be found in subdirectories under your base Madisons directory. If your store is published under a different name, the path is different than the one shown on this slide.

Native application architecture



This slide shows the architecture of the native Android application. The API invoker and JSON Builder and Parser components provide the communication with the WebSphere Commerce server using the REST service API. The activities provide the store interface and page flow. Each page in the store has its own activity. You can customize the store by adding or changing activities. If your customization involves new or modified REST services, you can extend the JSON Builder and Parser to support the new data. The native application also has the ability to communicate with other applications on the phone. The sample Madisons applications makes use of the barcode scanner, if one exists, GPS data for store location and the phone's address book for contact information.

Native business objects

- Java classes that define native representation of nouns
  - Address
  - Cart
  - ClientLocation
  - PaymentInstruction
  - Person
  - PointOfInterest
  - Product
  - Store
  - UserBilling
  - UserShipping
- Do not contain all information found in the corresponding noun

The native application contains Java classes that represent the business objects accessed by the store. These business objects are built by the JSON parser when a REST service response contains business object data. For example, the cart object is populated with the response from a get cart request. The default business objects do not necessarily include the full content of the REST response. In particular, user data is not included in the default objects because it is not used in the sample application. You can extend the business objects or add new ones as needed to store custom data for your store.

Resource definitions

- Define resource strings
  - Madisons-AndroidNativeMobile\res\values\strings.xml

```
<string name="registerBenefits">Registering provides you with personalized services
    including: Quick checkout, Wish List, Order Status, SMS messages and
    Promotions.</string>
<string name="register">Register</string>
<string name="logonId">Logon ID</string>
<string name="password">Password</string>
```

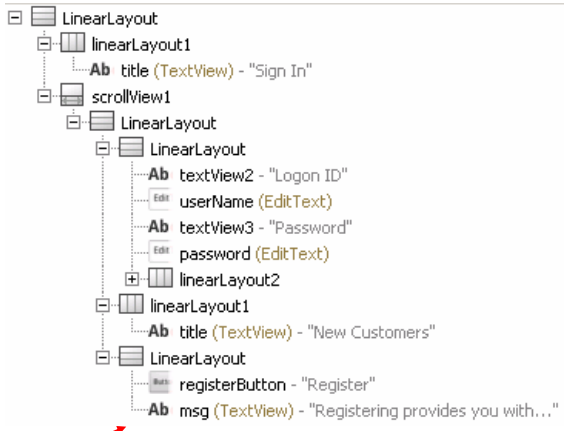
- R.java
  - Easy way to reference all your resources
    - R.string.<identifier>
    - R.id.<identifier>
    - R.layout.<identifier>

Similar to web-based stores, the native application defines resource strings to support translation of the store text strings. Resource strings are defined in an XML file at the location shown on the slide. The code snippet shows an example of some resource strings being defined.

To simplify access to various resources in the Java code, Android provides a generated resource file called R.java. This file contains pointers to all the resources available to the application. String resources are accessed with the notation R.string followed by the string's identifier. The identifier is the name attribute you provided when defining the string. Other resources that can be accessed with R.java include named UI elements and UI layouts.

UI definition

- Sample layout: account_signin.xml



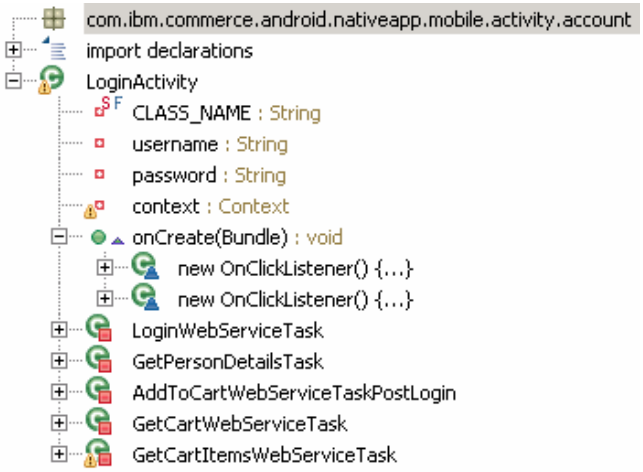

```
<TextView android:id="@+id/msg" style="@style/NormalText"
    android:layout_height="wrap_content"
    android:padding="5dip"
    android:layout_width="wrap_content"
    android:text="@string/registerBenefits">
</TextView>
```

13 Mobile application architecture

The UI layout for the Android application is defined in a series of XML files, one for each screen in the application. The Android development environment in Eclipse provides two views of the layout, a graphical view and an XML view. A separate tab shows a tree representation of the XML structure. On the slide you see the three different representations of the message below the register button. In the XML snippet, you can see where the resource ID for this UI element is defined and where the resource string is accessed to display the message.

Activities - definition

- Java class that defines something an application user can do
  - Login
  - View product details
  - Update billing address



Activities are Java classes that control the actions available to a user of the application. They control the flow between pages in the store, trigger REST services calls to get or update store resources and supply data for the UI elements to display. Each screen in the application has its own activity. A few examples are listed on the slide. On the right side of the slide you see a representation of the LoginActivity class. The onCreate method contains the logic to display the page. This includes event listeners that react to shopper interactions with the screen such as clicking a button. The inner classes perform page-specific tasks such as invoking REST service calls to get data or send updates.

Native application customization

- Receive additional data from REST services
  - JSON Parser
  - Native business object
- Create or modify a store page
  - Layout
  - Activity
  - Resource strings

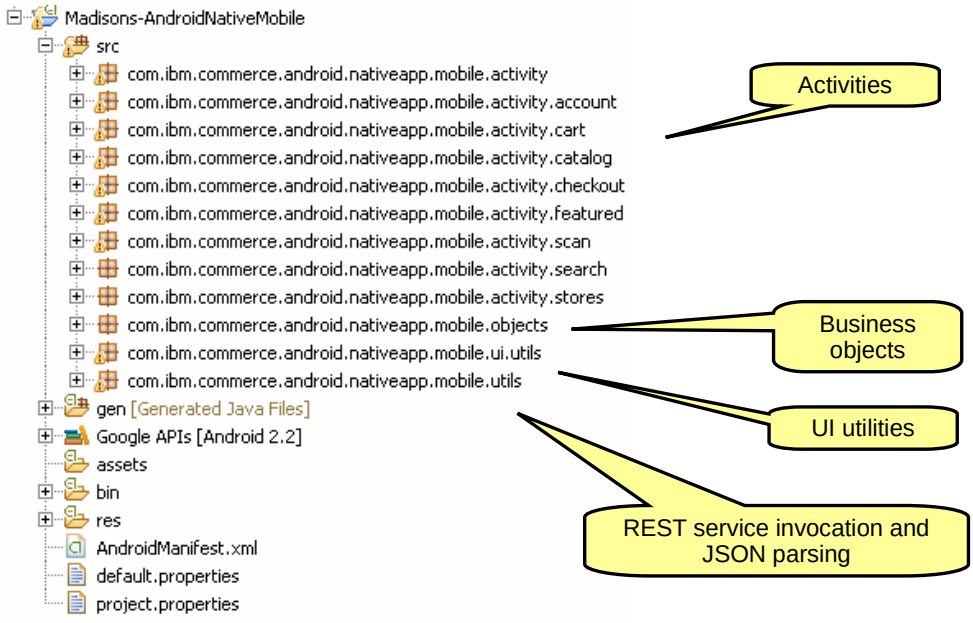
The building blocks discussed in this section so far can all be created or extended to customize your store application. If you need access to information from a REST response that is not currently available in the store, you need to extend the JSON parser and the native business object class where the data is stored. To add or change store pages, you need to create a layout and corresponding activity. The new activity needs to be started from an existing page in the store flow such as by adding a new listener. Any new text displayed in the store should be defined as a resource string.

Manifest file – AndroidManifest.xml

- General
  - Minimum SDK level
- Application
  - Register activities
- Permissions
  - Access that must be granted for the application to run
- Instrumentation
  - Profiling information used during development and test

The manifest file is the controlling file for your Android application. It contains several types of information. The general level information includes details such as the version of the application and the minimum SDK level it supports. Each activity in the application must be registered in the manifest. If the application needs certain permissions to be able to operate correctly those are listed. When the application is installed, the device owner is asked whether they are willing to grant those permissions to the application. Permissions might include actions such as accessing the internet or the built-in camera. Instrumentation information can also be included in the manifest. This is only included during development of the application and is removed before it is released.

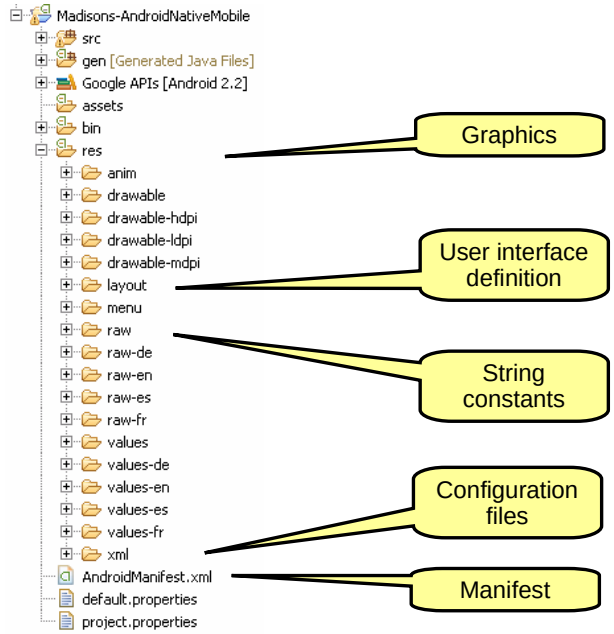
Native application structure – source



17 Mobile application architecture

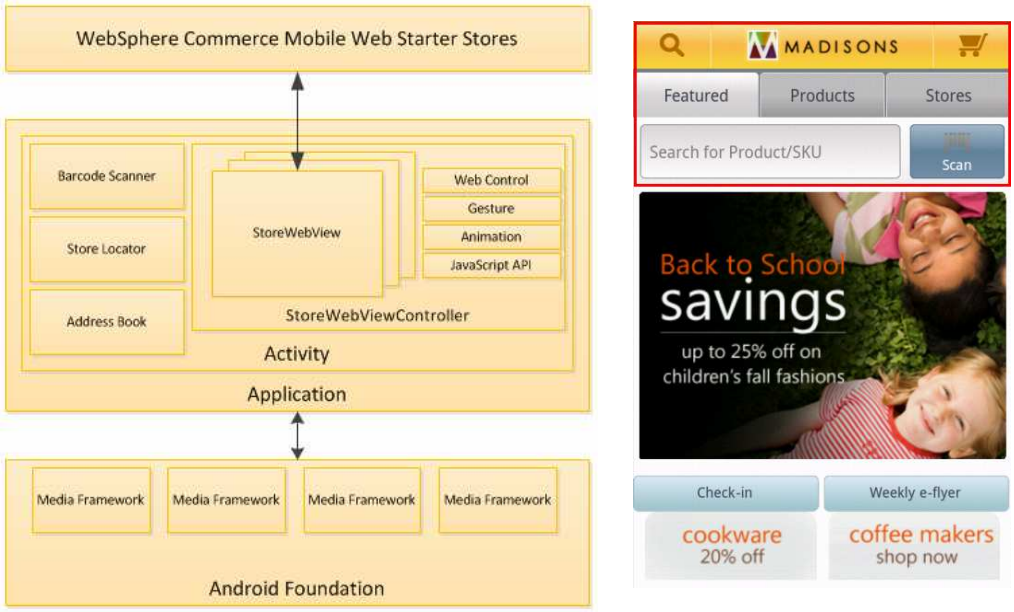
Here you see the file structure of the Java portion of the native application. The activities are separated into different packages based on the type of store function they provide. There are also code packages for the business objects, UI utilities and REST service invocation and JSON parsing code.

Native application structure - resources



Within the resources folder are store graphics, the UI layout files, the string resources and additional XML configuration files. The manifest file is found at the root of the project folder.

Hybrid application architecture



19 Mobile application architecture

The hybrid application has the same high level architecture as the native application. It has layout files and activities to implement the device-rendered content. It has string resource files for text labels. The main difference is in the way the application communicates with the WebSphere Commerce server. Instead of communicating through REST services and rendering the store pages within the device, the hybrid application accesses the built-in WebKit browser and makes HTTP requests that are routed to the Struts servlet instead of the REST services. Server-side JSP files are used to generate the page content and an HTML response is returned and rendered by the built-in browser. As long as shoppers are interacting with the storefront, the communication is between the browser and WebSphere Commerce. If the shopper launches a native feature, such as the option menu for the application, then XML layouts and activity classes are used to render the screen content natively. A class called the StoreWebViewController is the main control point between the browser and WebSphere Commerce. This class is discussed more on the next slide.

StoreWebViewController

- Control web content returned from server
  - User Agent in HTTP head specifies hybrid application
  - WebSphere Commerce allocates special device ID (-21)
  - Page headers and footers removed from response
- Animation
  - Mimic native applications with animation between page forward and back
- Gesture
  - Allow shoppers to flip through catalog pages

The StoreWebViewController class acts as the controller for the hybrid application. It makes requests to the WebSphere Commerce server and displays the results within Android activities. A special device ID is used for all requests coming from the hybrid application. This allows the WebSphere Commerce server to differentiate from requests coming from the smart phone web store. Since the hybrid application has a native shell, the standard headers and footers are removed from the HTML response for these requests.

The controller also implements some animation to give the browser-based content a more native appearance. Page transitions are animated and swipe gestures are supported.

JavaScript bridge

- Invoke Android native capabilities through JavaScript
  - JSP calls JavaScript function
  - JavaScript calls Android function
  - Madisons hybrid application implements Android function
  - Android function invokes native capabilities
- Uses
  - Address book integration
  - Store locator map integration
- Implementation of JavaScript bridge
  - Madisons-AndroidHybridMobile\src\com\ibm\commerce\android\hybrid\mobile\web\JavaScriptInterface.java

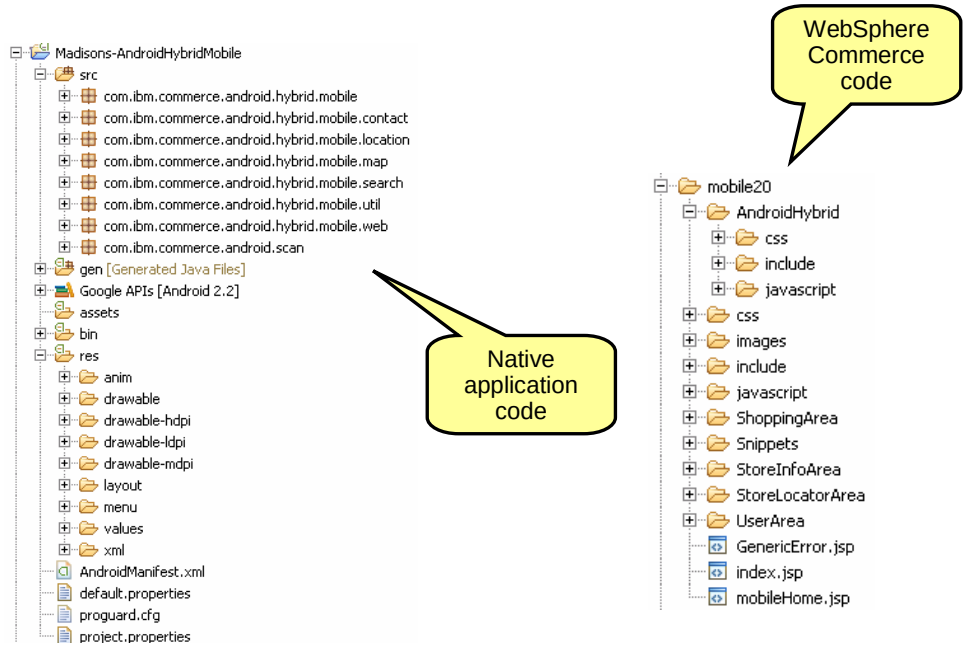
The JavaScript bridge allows the store pages rendered in the browser to access native functions on the device by creating a bridge through the native code in the hybrid application. The store JSP is written to call a JavaScript function to perform an action, such as selecting a contact from the address book. The JavaScript method calls an Android function that is implemented by the hybrid application. Since the application is running natively on the device, it can be granted permission to access native device functions such as the address book. There are only a few places in the store where the JavaScript bridge is used such as accessing the device address book for contacts and GPS for store location. The Android implementation class of the bridge is listed on the slide.

Native features in hybrid application

- Shell
  - Header
  - Tabs
  - Search
  - Option menu
- 2D barcode scanner
  - Requires separate barcode scanning application (such as Zxing)
- Voice search
  - Requires Google Voice Search
- Address book integration
  - Select address from contacts list
- Store locator and Google map integration
  - Requires Google Maps API key

This slide lists the native features included in the hybrid application. The shell provides the frame that surrounds the embedded browser. The other features are those that require interaction with device-specific features or applications.

Hybrid application structure



This slide shows the package structure of the hybrid application. On the left is the application code. It is similar to the native application but since most of the store pages are defined by JSP files there is significantly less Java code. The packages are organized by the store capability they provide. On the right is the server-side code. Most of the store JSP files are shared with the smart phone web store. A few files are specific to the hybrid application and these are contained in the AndroidHybrid subdirectory. The application-specific files include the header and footer JSP fragments that are empty for the hybrid application and the JavaScript bridge functions.

Section

Building Android applications

This section discusses how to set up an Android development environment and build the applications.

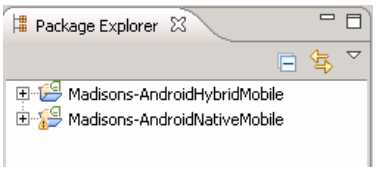
Development environment

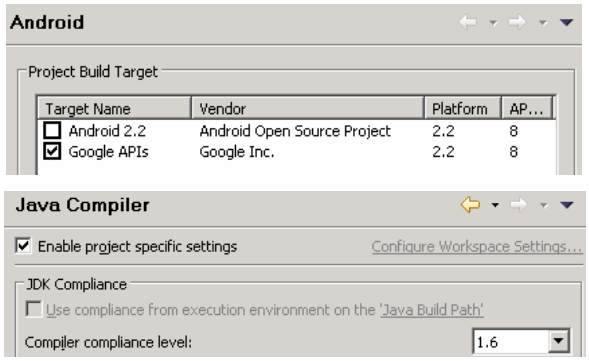
- Eclipse
  - Install a separate Eclipse environment
  - Do not use WebSphere Commerce Developer
- Oracle JDK 6
  - JRE is not enough, you need the JDK
- Android SDK starter package
  - Contains core SDK tools
- ADT plug-in for Eclipse
  - Adds Android development tools to Eclipse
- Android SDK platform
  - Android platform you are developing for
    - Android 2.2 (API 8)
  - Google APIs add-on

Android application development is done in an Eclipse environment separate from your WebSphere Commerce Developer environment. To compile the Android application files, you need to install JDK 6 from Oracle, the JRE is not enough. Once you have Eclipse and a JDK installed, you can install the Android SDK starter package and the ADT plug-in for Eclipse. The starter package gives you the common SDK platform tools and the plug-in gives you Android development tools. The final step is to install one or more Android SDK platforms. The SDK platform is the level of Android you are developing for. The Madisons sample applications are built on Android 2.2 with the Google APIs add-on package for map support.

Import sample application code

- Sample location
  - <ToolkitInstallDir>/components\store-enhancements\samples\Android
- Make sure
  - Oracle JDK is installed
  - Build target is Android 2.2 with Google APIs
  - Java compiler level is 1.6

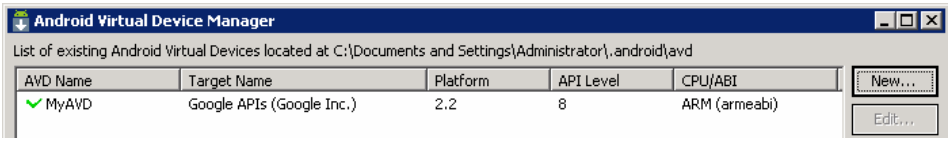




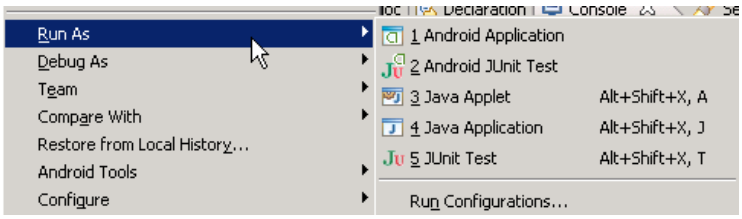
When your Eclipse environment is set up and you've installed an Android SDK, you are ready to create Android projects for the hybrid and / or native applications. The sample code is stored in the components folder at the location shown on the slide. You can create an empty project in Eclipse and import the sample code into your project. If you run into errors when compiling the imported files check your environment settings. You must be using the Oracle JDK, your selected build target should be Google APIs for the 2.2 platform and the Java compiler level should be set to 1.6.

Running applications

- Create an Android virtual device (AVD)

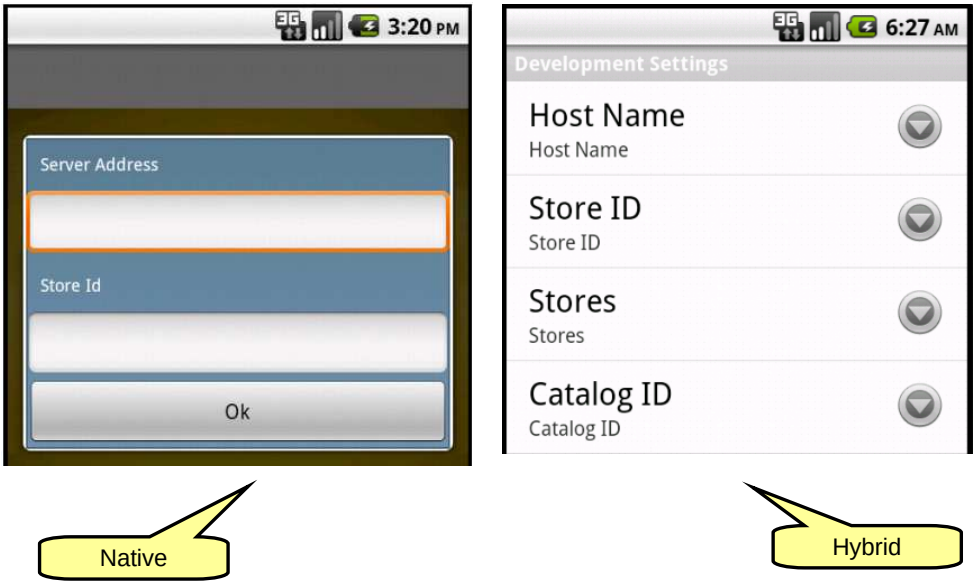


- Launch application



The Android SDK comes with an emulator you can use to test your application. In order to run the emulator, you need to create an Android virtual device (AVD). Use the Android Device Manager to create and manage AVDs. Once you have an AVD created, you can launch your application from Eclipse using the pop-up menu. You can also run your application in debug mode and set breakpoints in your Java code.

Configuring application settings in development mode



28 Mobile application architecture

When you first launch the sample Madisons applications during development, you need to provide details about the environment the applications are connecting to. In the native application you are prompted for this information the first time you launch the application. In the hybrid application you need to open the options menu to get to the Development settings menu. For a production server, you need to hard code these values into the application. The exception is the catalog ID which should be managed on the WebSphere Commerce server side. The configuration files for this data are shown on the upcoming slides.

Native application configuration

```
<resources>
    <string name="StoreName">Madisons</string>
    <string name="StoreNameDir">mobile20</string>
    <string name="StoreId"></string>

    <!-- map api key -->
    <string name="map_api_key"></string>

    <!-- context path for rest services -->
    <string name="contextPath">/wcs/resources/store/</string>
    <string name="contextPathForLBS">/wcs/resources/lbs/store/</string>
    <string name="contextPathImage">/wcsstore/</string>

    <string name="SizeThreshold">100</string>
    <string name="TimeThreshold">1</string>
    <string name="topCategoriesThumbnailImageExtension">.png</string>

    <string name="SSL_FACTORY">com.ibm.commerce.android.nativeapp.mobile.utils
    <string name="KEY_STORE_PASSWORD"></string>

    <string name="SCAN_INTENT">com.google.zxing.client.android.SCAN</string>
    <string name="SCAN_PACKAGE">com.google.zxing.client.android</string>
    <item type="integer" name="AccessThreshold">5</item
</resources>
```

- Note: no predefined property for host name
- Location: Madisons-AndroidNativeMobile\res\values\connpref.xml

Both the native and hybrid applications have configuration files where details about the application are stored. This slide shows the configuration file for the native application. Data such as the store ID and context path for REST services is stored in this file. If you are adding mapping capability to your store, the Google map API key is also stored here.

Sales catalogs with native applications

- New method getDefaultCatalogId(String storied) in StoreConfigurationHelper
- Create XML extension folder com.ibm.commerce.catalog-ext
- Set sales catalog ID in wc-component.xml
- Defaults to master catalog ID if not set

```
<_config:DevelopmentComponentConfiguration
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/xmlns/prod/commerce/foundation/conf
    xmlns:_config="http://www.ibm.com/xmlns/prod/commerce/foundation/config">

    <_config:extendedconfiguration>
        <_config:configgrouping name="StoreIdDefaultCatalogIdMapForREST">
            <_config:property name="10101" value="10151" />
        </_config:configgrouping>
    </_config:extendedconfiguration>
</_config:DevelopmentComponentConfiguration>
```


The native application does not hard code the catalog ID. Instead, the catalog ID is determined on the server side by the getDefaultCatalogId method in the StoreConfigurationHelper class. The default catalog ID is the master catalog. If you want your application to connect to a sales catalog, you can set the value in the wc-component.xml file in the catalog component extension folder. An example of setting a sales catalog ID is shown on the slide.

Hybrid application configuration

```
<resources>
    <string name="http_user_agent">IBM (WebSphere Commerce) Android Smartphone Hybrid Clie

    <string name="storeName">Madisons</string>
    <string name="hostName"></string>
    <string name="storeId"></string>
    <string name="storeIdentifier">madisonsesite</string>
    <string name="catalogId"></string>
    <string name="map_api_key"></string>
    <string name="languageId">-1</string>
    <!--
        By default, the SEO is used to support catalog dynamic binding in server side,
        it gives the flexibility to direct the hybrid app to the pointing master catalog
        or sales catalog without app update.
        The following two URL paths are SEO encoded.
     -->
    <string name="home_page_url_path_seo">webapp/wcs/stores/servlet/m20/{language}/{store]
    <string name="top_category_url_path_seo">webapp/wcs/stores/servlet/m20/{language}/{sto
```

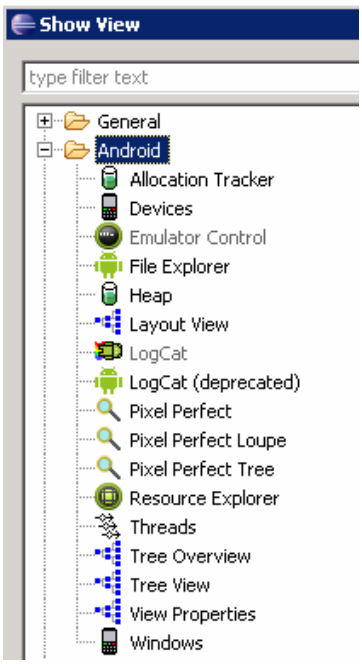
- Location: Madisons-AndroidHybridMobile\res\values\system_settings.xml

31 Mobile application architecture

Here is an example of the configuration file for the hybrid application. While the hybrid application does support setting the catalog ID within the application this is not a required step. If your store supports SEO than the catalog ID is set in the WebSphere Commerce database as part of the store and catalog token. To change to a different catalog you just need to update the SEO configuration. Similar to the native application configuration file, your Google map API key is set in this file if you choose to include mapping integration in your application.

Android development and test tools

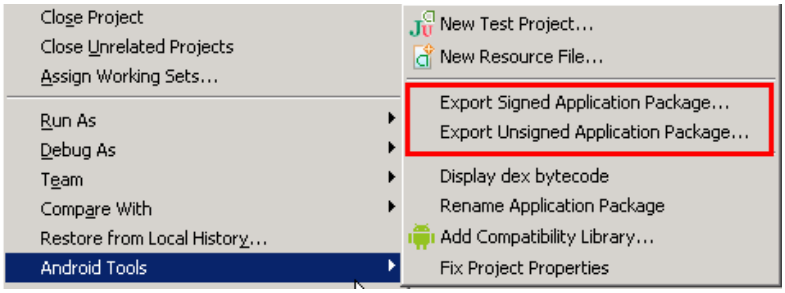
- LogCat
  - View log output from a running application
- Emulator Control
  - Set GPS data for testing



The ADT plug-in for Eclipse provides many development tools for use in creating and testing your Android application. This presentation does not review the Android development tools in detail. You can learn more on the Android developer website. Two tools that you will find useful for testing are LogCat and the Emulator Control view. LogCat displays the log output from the running application. The sample applications contain log statements and you can add your own. The Emulator Control view is useful for setting sample GPS data to test mapping features and location-based services.

Exporting Android applications

- Compiled Android application is exported as a .apk file
- Install compiled file on emulator or physical device
- Applications must be signed in order to be installed
  - Debug key provided for development and test
  - Private key required for applications ready for release
  - Only export unsigned application when you will sign it as a separate step



When you are ready to install your application on an Android phone, you need to export your application as an Android package. The package will have a .apk extension. You can install an apk file directly onto an emulator or onto a physical device. In order for your application to be installed it must be signed. A debug key is provided with the Android plug-in. The location of this key varies by operating system. You can find the default locations on the Android developer website. When your application is ready to deploy to the Android market, you need to sign it with your own private key instead of the debug key.

Section

Problem determination

This section provides some problem determination tips.

Diagnosing problems – mobile web

- Does the problem exist in the desktop store as well?
  - Yes: Start with the component where the problem is
  - No: Start with the store JSP
- UI elements not working correctly
  - Device does not have WebKit browser support
  - Dojo is not being loaded
- Debugging mobile web stores from a desktop
  - Use Safari or Chrome for WebKit support
  - User agent switching simulates different mobile devices

When debugging problems in the mobile web stores it helps to verify whether the function is working in the desktop store. If not, the component that has the problem is a good place to start. If the desktop store is working correctly and it's only the mobile store that has a problem, start your debugging from the affected store page. If you have problems with the UI not displaying or working correctly there are two likely causes. You should check that you are viewing the store in a browser with WebKit support and that Dojo is being loaded correctly. To debug a mobile web store from a desktop machine, use either Safari or Chrome. Both browsers have WebKit support. You can make use of user agent switching in your desktop browser to simulate different mobile devices.

Diagnosing problems – hybrid applications

- Does the problem exist in the mobile web application as well?
  - Yes: Debug web application
  - No:
    - Check the application configuration
    - Check if the function is using the JavaScript bridge or native code
- To debug problems in Android code
  - Run the application in the emulator
    - LogCat
    - Java debugger

Since hybrid applications get the majority of their screen content from the same JSP files as the mobile web application, you should start by checking whether the problem also exists in the mobile web store. If the problem is unique to the hybrid application, the next step is to verify the application is configured correctly. The host name, store ID or catalog ID for the application might be incorrect. If the application is configured correctly, the problem might be in one of the JavaScript bridge functions or in the native application code. You can debug problems in the Android code by running the application in the emulator and viewing the log output in the LogCat view. You can also set breakpoints in the Java code and use the Eclipse Java debugger.

Diagnosing problems – native application

- Does the problem exist in any other storefront?
  - Yes: Start with the component where the problem is
  - No: Start with the Android activity
- To debug problems in Android code
  - Run the application in the emulator
    - LogCat
    - Java debugger
- To debug problems in REST services calls
  - Call the service using Poster and verify response
  - Enable server-side trace

The native application has entirely different storefront code from any of the other mobile stores. If you find a problem in the native application that you can reproduce in any other storefront then the problem lies in the component-specific code. For problems that are unique to the native application you can start by debugging the Android activity using the emulator and LogCat or the Java debugger. To test the REST services separate from the Android client code, you can use the Poster add-on for Firefox to send a REST service request directly to the WebSphere Commerce server and view the response. You can also enable the REST component trace as described on the next slide.

Trace

- com.ibm.commerce.location.*
  - Location-based services
- com.ibm.commerce.rest.*
  - Resource handlers
- com.ibm.commerce.foundation.rest.*
  - Entity providers
  - Caching
  - Low level BOD to JSON mapping
  - JSON response string
    - com.ibm.commerce.foundation.rest.bodmapping.BODMappingUtility createMapFromBOD RETURN
- LogCat
  - Android application trace

There are several trace options that are useful in debugging problems with the mobile web stores and native applications. For any problems with location-based services such as check-in and e-flyer you should start with the location component trace. You might want to add the marketing component trace if the problem seems to be with a location-based marketing activity. For the native Android application, the first level to look at on the server side is the REST component trace or, if you need to go deeper, the foundation REST component. If the problem is not at the REST service layer than the next step is the component trace for the store function that has the problem. Individual components are the starting point for the hybrid application and mobile web store which are all JSP based and call WebSphere Commerce services directly.

References

- Smart phone and tablet starter stores
  - http://publib.boulder.ibm.com/infocenter/wchelp/v7r0m0/topic/com.ibm.commerce.starterstores.doc/concepts/csmmobile.htm
- Mobile web applications
  - http://publib.boulder.ibm.com/infocenter/wchelp/v7r0m0/topic/com.ibm.commerce.starterstores.doc/concepts/csmmobileweb.htm
- Mobile applications
  - http://publib.boulder.ibm.com/infocenter/wchelp/v7r0m0/topic/com.ibm.commerce.starterstores.doc/concepts/csmmobileapp.htm
- Android developer site
  - http://developer.android.com/index.html

This slide contains some references for further reading on this topic.

Summary

- Overview
- Architecture
- Building Android applications
- Problem determination

This presentation began with a brief overview of the mobile commerce solution for Feature Pack 4. It then focused on the solution architecture, particularly for the device-specific applications. Following the architecture was a short discussion on how to build Android applications. The presentation concluded with some problem determination tips.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send email feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_MobileAppArchitecture.ppt

This module is also available in PDF format at: ../MobileAppArchitecture.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, disclaimer, and copyright information