Version 29.0: Winter '14

Creating a Native Android Warehouse Application

Last updated: October 23, 2013

Table of Contents

Chapter 1

Tutorial: Creating a Native Android Warehouse Application

In this chapter ...

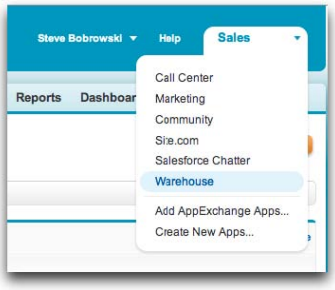
Apply your knowledge of the native Android SDK by building a mobile inventory management app. This tutorial demonstrates a simple master-detail architecture that defines two activities. It demonstrates Mobile SDK application setup, use of REST API wrapper classes, and Android SDK integration.

Prerequisites

This tutorial requires the following tools and packages.

- This tutorial uses a Warehouse app that contains a basic inventory database. You'll need to install this app in a DE org. If you install it in an existing DE org, be sure to delete any existing Warehouse components you've made before you install.

  1. Click the installation URL link: https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti0000000MMMT
  2. If you aren't logged in already, enter the username and password of your DE org.
  3. On the Package Installation Details page, click **Continue**.
  4. Click **Next**, and on the Security Level page click **Next**.
  5. Click **Install**.
  6. Click **Deploy Now** and then **Deploy**.
  7. Once the installation completes, you can select the **Warehouse** app from the app picker in the upper right corner.



  8. To create data, click the **Data** tab.
  9. Click the **Create Data** button.

- Install the latest versions of:

  ◊ Java JDK 6.
  ◊ Apache Ant 1.8 or later.
  ◊ Android SDK, version 21 or later—`http://developer.android.com/sdk/installing.html`.

Note: For best results, install all previous versions of the Android SDK as well as your target version.

  ◊ Eclipse 3.6 or later. See `http://developer.android.com/sdk/requirements.html` for other versions.
  ◊ Android ADT (Android Development Tools) plugin for Eclipse, version 21 or later—`http://developer.android.com/sdk/eclipse-adt.html#installing`.
  ◊ In order to run the application in the Emulator, you need to set up at least one Android Virtual Device (AVD) that targets Platform 2.2 or above (we recommend 4.0 or above). To learn how to set up an AVD in Eclipse, follow the instructions at `http://developer.android.com/guide/developing/devices/managing-avds.html`.
  ◊ A Developer Edition organization with a connected app.

- Install the Salesforce Mobile SDK using npm:

  1. If you've already successfully installed Node.js and npm, skip to step 4.
  2. Install Node.js on your system. The Node.js installer automatically installs npm.

     i. Download Node.js from www.nodejs.org/download.

     ii. Run the downloaded installer to install Node.js and npm. Accept all prompts asking for permission to install.

3. At the Terminal window, type `npm` and press `Return` to make sure your installation was successful. If you don't see a page of usage information, revisit Step 2 to find out what's missing.
4. At the Terminal window, type `sudo npm install forcedroid -g`

   This command uses the forcedroid package to install the Mobile SDK globally. With the `-g` option, you can run `npm install` from any directory. The npm utility installs the package under `/usr/local/lib/node_modules`, and links binary modules in `/usr/local/bin`. Most users need the `sudo` option because they lack read-write permissions in `/usr/local`.

Create a Native Android App

In this tutorial, you learn how to get started with the Salesforce Mobile SDK, including how to install the SDK and a quick tour of the native project template using your DE org. Subsequent tutorials show you how to modify the template app and make it work with the Warehouse schema.

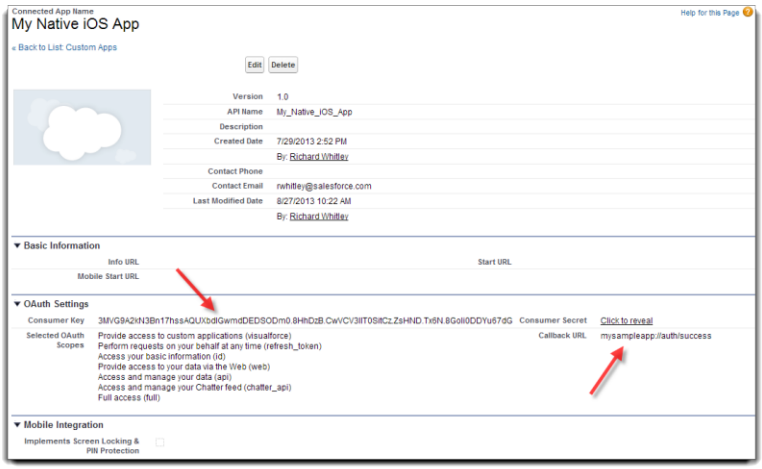
Step 1: Create a Connected App

In this step, you learn how to configure a Connected App in Force.com. Doing so authorizes the mobile app you will soon build to communicate securely with Force.com and access Force.com APIs on behalf of users via the industry-standard OAuth 2.0 protocol.

1. In your DE org, click ***`Your Name`*** > **Setup** then click **Create** > **Apps**.
2. Under **Connected Apps**, click **New** to bring up the **New Connected App** page.
3. Under **Basic Information**, fill out the form as follows:
   - **Connected App Name**: `My Native Android App`
   - **API Name**: accept the suggested value
   - **Contact Email**: enter your email address
4. Under OAuth Settings, check the **Enable OAuth Settings** checkbox.
5. Set **Callback URL** to `mysampleapp://auth/success`.
6. Under **Available OAuth Scopes**, check "Access and manage your data (api)" and "Perform requests on your behalf at any time (refresh_token)", then click **Add**.
7. Click **Save**.

After you save the configuration, notice the details of the Connected App you just created.

- Note the Callback URL and Consumer Key; you will use these when you set up your native app in the next step.
- Mobile apps do not use the Consumer Secret, so you can ignore this value.



Step 2: Create a Native Android Project

To create a new Mobile SDK project, use the forcedroid utility again in the Terminal window.

1. Change to the directory in which you want to create your project.
2. To create an Android project, type `forcedroid create`.

   The forcedroid utility prompts you for each configuration value.
3. For application type, enter `native`.
4. For application name, enter `Warehouse`.
5. For target directory, enter `tutorial/AndroidNative`.
6. For package name, enter `com.samples.warehouse`.
7. When asked if you want to use SmartStore, press **Return** to accept the default.

Step 3: Run the New Android App

Now that you've successfully created a new Android app, build and run it in Eclipse to make sure that your environment is properly configured.

Note: If you run into problems, first check the Android SDK Manager to make sure that you've got the latest Android SDK, build tools, and development tools. You can find the Android SDK Manager under **Window** > **Android SDK Manager** in Eclipse. After you've installed anything that's missing, close and restart Android SDK Manager to make sure you're up-to-date.

Importing and Building Your App in Eclipse

The forcedroid script also prints instructions for running the new app in the Eclipse editor.

1. Launch Eclipse and select `tutorial/AndroidNative` as your workspace directory.
2. Select **Eclipse** > **Preferences**, choose the **Android** section, and enter the Android SDK location.
3. Click OK.
4. Select **File** > **Import** and select **General** > **Existing Projects into Workspace**.
5. Click Next.
6. Specify the `forcedroid/native` directory as your root directory. Next to the list that displays, click **Deselect All**, then browse the list and check the SalesforceSDK project.

7. Click **Import**.
8. Repeat Steps 4–8. In Step 6, choose `tutorial/AndroidNative` as the root, then select only your new project.

When you've finished importing the projects, Eclipse automatically builds your workspace. This process can take several minutes. When the status bar reports zero errors, you're ready to run the project.

1. In your Eclipse workspace, Control-click or right-click your project.
2. From the popup menu, choose **Run As** > **Android Application**.

Eclipse launches your app in the emulator or on your connected Android device.

Step 4: Explore How the Android App Works

The native Android app uses a straightforward Model View Controller (MVC) architecture.

- The model is the Warehouse database schema
- The views come from the activities defined in your project
- The controller functionality represents a joint effort between the Android SDK classes, the Salesforce Mobile SDK, and your app

Within the view, the finished tutorial app defines two Android activities in a master-detail relationship. MainActivity lists records from the Merchandise custom objects. DetailActivity, which you access by clicking on an item in MainActivity, lets you view and edit the fields in the selected record.

MainActivity Class

When the app is launched, the `WarehouseApp` class initially controls the execution flow. After the login process completes, the `WarehouseApp` instance passes control to the main activity class, via the `SalesforceSDKManager` singleton.

In the template app that serves as the basis for your new app, and also in the finished tutorial, the main activity class is named `MainActivity`. This class subclasses `SalesforceActivity`, which is the Mobile SDK base class for all activities.

Before it's customized, though, the app doesn't include other activities or touch event handlers. It simply logs into Salesforce, issues a request using Salesforce Mobile SDK REST APIs, and displays the response in the main activity. In this tutorial you replace the template app controls and repurpose the SOQL REST request to work with the Merchandise custom object from the Warehouse schema.

DetailActivity Class

The `DetailActivity` class also subclasses `SalesforceActivity`, but it demonstrates more interesting customizations. `DetailActivity` implements text editing using standard Android SDK classes and XML templates. It also demonstrates how to update a database object in Salesforce using the `RestClient` and `RestRequest` classes from the Mobile SDK.

RestClient and RestRequest Classes

Mobile SDK apps interact with Salesforce data through REST APIs. However, you don't have to construct your own REST requests or work directly at the HTTP level. You can process SOQL queries, do SOSL searches, and perform CRUD operations with minimal coding by using static convenience methods on the `RestRequest` class. Each `RestRequest` convenience method returns a `RestRequest` object that wraps the formatted REST request.

To send the request to the server, you simply pass the `RestRequest` object to the `sendAsync()` or `sendSync()` method on your `RestClient` instance. You don't create `RestClient` objects. If your activity inherits a Mobile SDK activity class such as `SaleforceActivity`, Mobile SDK sends it an instance of `RestClient` to the `onResume()` method. Otherwise, you can call `ClientManager.getRestClient()`. Your app uses the connected app information from your `bootconfig.xml` file so that the `RestClient` object can send REST requests on your behalf.

Customize the List Screen

In this tutorial, you modify the main activity and its layout to make the app specific to the Warehouse schema. You also adapt the existing SOQL query to obtain all the information we need from the Merchandise custom object.

Step 1: Remove Existing Controls

The template code provides a main activity screen that doesn't suit our purposes. Let's gut it to make room for our code.

1. From the Package Explorer in Eclipse, open the `res/layout/main.xml` file. Make sure to set the view to text mode. This XML file contains a `<LinearLayout>` root node, which contains three child nodes: an `<include>` node, a nested `<LinearLayout>` node, and a `<ListView>` node.
2. Delete the nested `<LinearLayout>` node that contains the three `<Button>` nodes. The edited file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="vertical" android:layout_width="fill_parent"
 android:layout_height="fill_parent" android:background="#454545"
 android:id="@+id/root">

 <include layout="@layout/header" />

 <ListView android:id="@+id/contacts_list" android:layout_width="fill_parent"
           android:layout_height="fill_parent" />
</LinearLayout>
```

3. Save the file, then open the `src/com.samples.warehouse/MainActivity.java` file.
4. Delete the `onClearClick()`, `onFetchAccountsClick()`, and `onFetchContactsClick()` methods. If the compiler warns you that the `sendRequest()` method is never used locally, that's OK. You just deleted all calls to that method, but you'll fix that in the next step.

Step 2: Update the SOQL Query

The `sendRequest()` method provides code for sending a SOQL query as a REST request. We can reuse some of this code while customizing the rest to suite our new app.

1. Rename `sendRequest()` to `fetchDataForList()`. Replace

```
private void sendRequest(String soql) throws UnsupportedEncodingException
```

with

```
private void fetchDataForList()
```

Note that we've removed the `throw` declaration. We'll reinstate it within the method body to keep the exception handling local. We'll add a `try...catch` block around the call to `RestRequest.getRequestForQuery()`, rather than throwing exceptions to the `fetchDataForList()` caller.

2. Add a hard-coded SOQL query that returns up to 10 records from the `Merchandise__c` custom object:

```
private void fetchDataForList()  {
    String soql = "SELECT Name, Id, Price__c, Quantity__c
        FROM Merchandise__c LIMIT 10";
```

3. Add the `try...catch` block. Replace this:

```
RestRequest restRequest = RestRequest.getRequestForQuery(getString(R.string.api_version),
 soql);
```

with this:

```
RestRequest restRequest = null;
try {
    restRequest =
        RestRequest.getRequestForQuery(getString(R.string.api_version), soql);
} catch (UnsupportedEncodingException e) {
    showError(MainActivity.this, e);
    return;
}
```

Here's the completed version of what was formerly the `sendRequest()` method:

```
private void fetchDataForList() {
    String soql = "SELECT Name, Id, Price__c, Quantity__c FROM
        Merchandise__c LIMIT 10";
    RestRequest restRequest = null;
    try {
        restRequest =
            RestRequest.getRequestForQuery(
                getString(R.string.api_version), soql);
    } catch (UnsupportedEncodingException e){
        showError(MainActivity.this, e);
        return;
    }

    client.sendAsync(restRequest, new AsyncRequestCallback() {
        @Override
        public void onSuccess(RestRequest request,
            RestResponse result) {
            try {
                listAdapter.clear();
                JSONArray records =
                    result.asJSONObject().getJSONArray("records");
                for (int i = 0; i < records.length(); i++) {
                    listAdapter.add(records.
                        getJSONObject(i).getString("Name"));
                }
            } catch (Exception e) {
                onError(e);
            }
        }

        @Override
        public void onError(Exception exception) {
            Toast.makeText(MainActivity.this,
                MainActivity.this.getString(
                    SalesforceSDKManager.getInstance().
                    getSalesforceR().stringGenericError(),
                    exception.toString()),
                Toast.LENGTH_LONG).show();
        }
    });
}
```

We'll call `fetchDataForList()` when the screen loads, after authentication completes.

4. In the `onResume(RestClient client)` method, add the following line at the end of the method body:

```
@Override
public void onResume(RestClient client) {
    // Keeping reference to rest client
    this.client = client;

    // Show everything
    findViewById(R.id.root).setVisibility(View.VISIBLE);
    // Fetch data for list
    fetchDataForList();
}
```

5. Finally, implement the `showError()` method to report errors through a given activity context. At the top of the file, add the following line to the end of the list of imports:

```
import android.content.Context;
```

6. At the end of the `MainActivity` class definition add the following code:

```
public static void showError(Context context, Exception e) {
    Toast toast = Toast.makeText(context,
        context.getString(
            SalesforceSDKManager.getInstance().getSalesforceR().stringGenericError(),
            e.toString()),
        Toast.LENGTH_LONG);
    toast.show();
}
```

7. Save the `MainActivity.java` file.

Step 3:Try Out the App

To test the app, Control-Click the app in Package Explorer and select **Run As** > **Android Application**. When the Android emulator displays, wait a few minutes as it loads. Unlock the screen and wait a while longer for the Salesforce login screen to appear. After you log into Salesforce successfully, click **Allow** to give the app the permissions it requires.

At this point, if you click a Merchandise record, nothing happens. You'll fix that in the next tutorial.

Create the Detail Screen

In the previous step, you modified the template app so that the main activity presents a list of up to ten Merchandise records. In this step, you finish the job by creating a detail activity and layout. You then link the main activity and the detail activity.

Step 1: Create the Detail Screen

To start, design the layout of the detail activity by creating an XML file named `res/layout/detail.xml`.

1. In Package Explorer, expand `res/layout`.
2. Control-click the layout folder and select **New** > **Android XML File**.
3. In the **File** field, type `detail.xml`.
4. Under **Root Element**, select **LinearLayout**.
5. Click **Finish**.

In the new file, define layouts and resources to be used in the detail screen. Start by adding fields and labels for name, price, and quantity.

6. Replace the contents of the new file with the following XML code.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/root"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#454545"
    android:orientation="vertical" >

    <include layout="@layout/header" />

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/name_label"
            android:width="100dp" />

        <EditText
            android:id="@+id/name_field"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:inputType="text" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/price_label"
            android:width="100dp" />

        <EditText
            android:id="@+id/price_field"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:inputType="numberDecimal" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/quantity_label"
            android:width="100dp" />

        <EditText
            android:id="@+id/quantity_field"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:inputType="number" />
    </LinearLayout>
```

```
</LinearLayout>
```

7. Save the file.
8. To finish the layout, define the display names for the three labels (`name_label`, `price_label`, and `quantity_label`) referenced in the `TextView` elements.

   Add the following to `res/values/strings.xml` just before the close of the `<resources>` node:

```
<!-- Detail screen -->
<string name="name_label">Name</string>
<string name="price_label">Price</string>
<string name="quantity_label">Quantity</string>
```

9. Save the file, then open the `AndroidManifest.xml` file in text view. If you don't get the text view, click the **AndroidManifest.xml** tab at the bottom of the editor screen.
10. Declare the new activity in `AndroidManifest.xml` by adding the following in the `<application>` section:

```
<!-- Merchandise detail screen -->
<activity android:name="com.samples.warehouse.DetailActivity"
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
</activity>
```

Except for a button that we'll add later, you've finished designing the layout and the string resources for the detail screen. To implement the screen's behavior, you define a new activity.

Step 2: Create the DetailActivity Class

In this module we'll create a new class file named `DetailActivity.java` in the `com.samples.warehouse` package.

1. In Package Explorer, expand the **WarehouseApp** > **src** > **com.samples.warehouse** node.
2. Control-click the `com.samples.warehouse` folder and select **New** > **Class**.
3. In the **Name** field, enter **DetailActivity**.
4. In the **Superclass** field, enter or browse for `com.salesforce.androidsdk.ui.sfnative.SalesforceActivity`.
5. Click **Finish**.

   The compiler provides a stub implementation of the required `onResume()` method. Mobile SDK passes an instance of `RestClient` to this method. Since you need this instance to create REST API requests, it's a good idea to cache a reference to it.
6. Add the following declaration to the list of member variables at the top of the new class:

```
private RestClient client;
```

7. In the `onResume()` method body, add the following code:

```
@Override
public void onResume(RestClient client) {
        // Keeping reference to rest client
        this.client = client;
}
```

Step 3: Customize the DetailActivity Class

To complete the activity setup, customize the `DetailActivity` class to handle editing of Merchandise field values.

1. Add the following imports to the list of imports at the top of `DetailActivity.java`:

```
import android.widget.EditText;
import android.os.Bundle;
```

2. At the top of the class body, add private `EditText` members for the three input fields.

```
private EditText nameField;
private EditText priceField;
private EditText quantityField;
```

3. Add a variable to contain a record ID from the Merchandise custom object. You'll add code to populate it later when you link the main activity and the detail activity.

```
private String merchandiseId;
```

4. Add an `onCreate()` method that configures the view to use the `detail.xml` layout you just created. Place this method just before the end of the class definition.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Setup view
    setContentView(R.layout.detail);
    nameField = (EditText) findViewById(R.id.name_field);
    priceField = (EditText) findViewById(R.id.price_field);
    quantityField = (EditText)
              findViewById(R.id.quantity_field);
}
```

Step 4: Link the Two Activities, Part 1: Create a Data Class

Next, you need to hook up `MainActivity` and `DetailActivity` classes so they can share the fields of a selected Merchandise record. When the user clicks an item in the inventory list, `MainActivity` needs to launch `DetailActivity` with the data it needs to display the record's fields.

Right now, the list adapter in `MainActivity.java` is given only the names of the Merchandise fields. Let's store the values of the standard fields (`id` and `name`) and the custom fields (`quantity`, and `price`) locally so you can send them to the detail screen.

To start, define a static data class to represent a Merchandise record.

1. In the Package Explorer, open **src** > **com.samples.warehouse** > **MainActivity.java**.
2. Add the following class definition at the end of the `MainActivity` definition:

```
/**
 * Simple class to represent a Merchandise
 */
static class Merchandise {
 public final String name;
 public final String id;
```

```
 public final int quantity;
 public final double price;

 public Merchandise(String name, String id, int quantity, double price) {
  this.name = name;
  this.id = id;
  this.quantity = quantity;
  this.price = price;
 }

 public String toString() {
  return name;
 }
}
```

3. To put this class to work, modify the main activity's list adapter to take a list of Merchandise objects instead of strings. In the `listAdapter` variable declaration, change the template type from `String` to `Merchandise`:

```
private ArrayAdapter<Merchandise> listAdapter;
```

4. To match the new type, change the `listAdapter` instantiation in the `onResume()` method:

```
listAdapter = new ArrayAdapter<Merchandise>(this, android.R.layout.simple_list_item_1,
    new ArrayList<Merchandise>());
```

Next, modify the code that populates the `listAdapter` object when the response for the SOQL call is received.

5. Add the following import to the existing list at the top of the file:

```
import org.json.JSONObject;
```

6. Change the `onSuccess()` method in `fetchDataForList()` as follows:

```
public void onSuccess(RestRequest request, RestResponse result) {
 try {
  listAdapter.clear();
  JSONArray records = result.asJSONObject().getJSONArray("records");
  for (int i = 0; i < records.length(); i++) {
   JSONObject record = records.getJSONObject(i);
   Merchandise merchandise = new Merchandise(record.getString("Name"),
record.getString("Id"), record.getInt("Quantity__c"), record.getDouble("Price__c"));
   listAdapter.add(merchandise);
  }
 } catch (Exception e) {
  onError(e);
 }
}
```

Step 5: Link the Two Activities, Part 2: Implement a List Item Click Handler

Next, we need to catch click events and launch the detail screen when these events occur. Let's make `MainActivity` the listener for clicks on list view items.

1. Open the `MainActivity.java` file in the editor.
2. Add the following import:

```
import android.widget.AdapterView.OnItemClickListener;
```

3. Change the class declaration to implement the `OnItemClickListener` interface:

```
public class MainActivity extends SalesforceActivity implements OnItemClickListener {
```

4. Add a private member for the list view:

```
private ListView listView;
```

5. Add the following code in bold to the `onResume()` method just before the `super.onResume()` call:

```
public void onResume() {
    // Hide everything until we are logged in
    findViewById(R.id.root).setVisibility(View.INVISIBLE);

    // Create list adapter
    listAdapter = new ArrayAdapter<Merchandise>(
        this, android.R.layout.simple_list_item_1, new ArrayList<Merchandise>());
    ((ListView) findViewById(R.id.contacts_list)).setAdapter(listAdapter);

// Get handle for list view
    listView = (ListView) findViewById(R.id.contacts_list);
    listView.setOnItemClickListener(this);

    super.onResume();
}
```

Now we're ready to add the implementation for the list item click handler.

6. Add the following imports:

```
import android.widget.AdapterView;
import android.content.Intent;
```

7. Just before the Merchandise class definition, add an `onItemClick()` method.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {

}
```

8. Get the selected item from the list adapter in the form of a Merchandise object.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
  Merchandise merchandise = listAdapter.getItem(position);

}
```

9. Create an Android intent to start the detail activity, passing the merchandise details into it.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    Merchandise merchandise = listAdapter.getItem(position);
    Intent intent = new Intent(this, DetailActivity.class);
        intent.putExtra("id", merchandise.id);
        intent.putExtra("name", merchandise.name);
        intent.putExtra("quantity", merchandise.quantity);
        intent.putExtra("price", merchandise.price);
        startActivity(intent);
}
```

Let's finish by updating the `DetailActivity` class to extract the merchandise details from the intent.

10. In the Package Explorer, open **srccom.samples.warehouseDetailActivity.java**.
11. In the `onCreate()` method, add code in bold font to assign values from the list screen selection to their corresponding data members in the detail activity:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Setup view
    setContentView(R.layout.detail);
    nameField = (EditText) findViewById(R.id.name_field);
    priceField = (EditText) findViewById(R.id.price_field);
    quantityField = (EditText)
        findViewById(R.id.quantity_field);
    // Populate fields with data from intent
    Bundle extras = getIntent().getExtras();
    merchandiseId = extras.getString("id");
    nameField.setText(extras.getString("name"));
    priceField.setText(extras.getDouble("price") + "");
    quantityField.setText(extras.getInt("quantity") + "");
}
```

Step 6: Implement the Update Button

You're almost there! The only part of the UI that's missing is a button that writes the user's edits to the server. You need to:

- Add the button to the layout
- Define the button's label
- Implement a click handler
- Implement functionality that saves the edits to the server

1. Reopen `detail.xml` and add the following `<Button>` node to the layout.

```
<Button
        android:id="@+id/update_button"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:onClick="onUpdateClick"
        android:text="@string/update_button" />
```

2. Save the `detail.xml` file, then open `strings.xml`.
3. Add the following button label string to the end of the list of strings:

```
<string name="update_button">Update</string>
```

4. Save the `strings.xml` file, then open `DetailActivity.java`.

   In the `DetailActivity` class, add a handler for the Update button's `onClick` event. The handler's name must match the `android:onClick` value in the `<Button>` node that you just added to `detail.xml`. In this case, the name is `onUpdateClick`. This method simply creates a map that matches `Merchandise__c` field names to corresponding values in the detail screen. Once the values are set, it calls the `saveData()` method to write the changes to the server.

5. To support the handler, add the following imports to the existing list at the top of the file:

```
import java.util.HashMap;
import java.util.Map;
import android.view.View;
```

6. Add the following method to the `DetailActivity` class definition:

```
public void onUpdateClick(View v) {
 Map<String, Object> fields = new HashMap<String, Object>();
 fields.put("Name", nameField.getText().toString());
 fields.put("Quantity__c", quantityField.getText().toString());
 fields.put("Price__c", priceField.getText().toString());
 saveData(merchandiseId, fields);
}
```

   The compiler reminds you that `saveData()` isn't defined. Let's fix that. The `saveData()` method creates a REST API update request to update the `Merchandise__c` object with the user's values. It then sends the request asynchronously to the server using the `RestClient.sendAsync()` method. The callback methods that will receive the server response (or server error) are defined inline in the `sendAsync()` call.

7. Add the following imports to the existing list at the top of the file:

```
import com.salesforce.androidsdk.rest.RestRequest;
import com.salesforce.androidsdk.rest.RestResponse;
```

8. Implement the `saveData()` method to the `DetailActivity` class definition:

```
private void saveData(String id, Map<String, Object> fields)  {
 RestRequest restRequest;
 try {
  restRequest = RestRequest.getRequestForUpdate(getString(R.string.api_version),
"Merchandise__c", id, fields);
 } catch (Exception e) {
        // You might want to log the error or show it to the user
  return;
 }

 client.sendAsync(restRequest, new RestClient.AsyncRequestCallback() {
  @Override
  public void onSuccess(RestRequest request, RestResponse result) {
   try {
    DetailActivity.this.finish();
   } catch (Exception e) {
                // You might want to log the error or show it to the user
   }
  }

  @Override
  public void onError(Exception e) {
            // You might want to log the error or show it to the user
  }
 });
}
```

That's it! Your app is ready to run and test.

Step 7: Try Out the App

1. Build your app and run it in the Android emulator. If you did everything correctly, a detail page appears when you click a Merchandise record in the Warehouse screen.
2. Update a record's quantity and price. Be sure to click the **Update** button in the detail view after you edit the values. When you navigate back to the detail view, the updated values display.
3. Log into your DE org and view the record using the browser UI to see the updated values.

Index

A

T