salesforce

SOFTWARE

Mobile SDK Workbook

Build iOS, Android and HTML5 Applications

Quick 30-Minute Tutorials

iOS

HTML

Table of Contents

Mobile SDK Workbook

The number of people using mobile devices to access application data grows every year. In the future, it's likely that mobile apps will be the primary way people do business.

The Salesforce Mobile SDK simplifies development by providing the following:

- Native OAuth implementations that work out-of-the-box
- OAuth access token management, including persistence and refresh capabilities
- Native REST API wrappers for building native applications
- Containers for building hybrid applications

About the Mobile SDK Version 1.1

The Salesforce Mobile SDK is an open-source suite of developer technologies that simplify the development of mobile applications. Because the Mobile SDK is new technology, expect lots of updates over the coming months.

The Salesforce Mobile SDK development team uses GitHub, a social coding community, where you can always find the latest releases in our public repositories at `https://github.com/forcedotcom/SalesforceMobileSDK-iOS` and `https://github.com/forcedotcom/SalesforceMobileSDK-Android`.

Important: If you have an earlier version of the Mobile SDK, you must upgrade to the latest version. This workbook and the associated samples contain additional code that will not work with previous versions of the SDK.

Prerequisites

This workbook assumes you are somewhat familiar with developing on the various platforms and technologies, and with development on Database.com or Force.com. We try and make it easy where we can, by supplying applications you can easily download and execute. All of the prerequisites are listed in the appropriate tutorials, but it's important to know that:

- For all of the tutorials, you'll need either a Database.com account or a Force.com Developer Edition organization.
- To build the iOS applications, you'll need Mac OS X Snow Leopard or Lion, Xcode 4.2+, and the Salesforce Mobile SDK for iOS cloned from the GitHub repository.
- To build the Android applications, you'll need the Java JDK 6, Eclipse, Android ADT plugin, and the latest Android SDK.
- Some familiarity with the REST API is assumed.
- Most of our resources are on GitHub, a social coding community. You can access all of our files in our public repository, but we think it's a good idea to join. `https://github.com`.

Choosing a Mobile App Development Scenario

There are three ways to develop mobile applications:

- **Native** — Native apps are coded using a mobile platform's native capabilities. Typically, these apps are more difficult to develop, but they also offer the best performance.
- **HTML5** — HTML5 apps are built using HTML5, CSS and JavaScript. These lightweight server-side pages typically offer the most portability, but don't have access to native platform features. HTML5 apps aren't device-specific, so the same app will run on an iOS, Android, Windows Mobile, or other device.
- **Hybrid** — Hybrid apps use a JavaScript bridge in a native container to merge the portability of HTML5 with native device capabilities, such as the camera or address book.

Choosing a Back End: Force.com or Database.com

The mobile applications that you build in this workbook will work whether you store your data on a Database.com or Force.com organization. Hereafter, the workbook assumes you are using a Force.com Developer Edition that uses a Force.com login end point such as `login.salesforce.com`. However, you can simply substitute your Database.com credentials and end point to get it working.

Note: If you choose Database.com, you can only do the native tutorials.

Tell Me More....

This workbook is designed so that you can go through the steps as quickly as possible. At the end of each step, there is an optional Tell Me More section with supporting information.

- The Mobile SDK home page is at `http://wiki.developerforce.com/Mobile_SDK`.
- `developer.force.com/workbooks`.
- You can find the latest version of this workbook and other workbooks at `http://developer.force.com/workbook`.
- To learn more about Force.com and to access a rich set of resources, visit Developer Force at `http://developer.force.com`.

Tutorial #1: Setting Up the Schema

The tutorials in this workbook build mobile applications that access data in the cloud. The applications first authenticate using OAuth 2.0, and then access the data by issuing standard HTTP verbs through the REST API. This tutorial lays the groundwork for future tutorials by creating a schema for the database, loading it with data, and configuring OAuth for the mobile clients.

This tutorial is a prerequisite to all other tutorials in this workbook.

Step 1: Sign Up

This workbook is designed to be used with either a Force.com Developer Edition organization or a Database.com base organization. If you sign up for Database.com, note that you are limited to native apps only, and some of the instructions might be slightly different, but not so much that you'll have trouble following along.

Sign Up for Force.com Developer Edition

1. In your browser go to `http://developer.force.com/join`.
2. Fill in the fields about you and your company.
3. In the `Email Address` field, make sure to use a public address you can easily check from a Web browser.
4. Enter a unique `Username`. Note that this field is also in the *form* of an email address, but it does not have to be the same as your email address, and in fact it's usually better if they aren't the same. Your username is your login and your identity on `developer.force.com`, and so you're often better served by choosing a username that describes the work you're doing, such as `develop@workbook.org`, or that describes you, such as `firstname@lastname.com`.
5. Read and then select the checkbox for the `Master Subscription Agreement`.
6. Enter the Captcha words shown and click **Submit Registration**.
7. In a moment you'll receive an email with a login link. Click the link and change your password.

Sign Up for Database.com Base Edition

1. In your browser go to `www.database.com`.
2. Click **Sign up now**.
3. Fill in the fields about you and your company.
4. In the `Email Address` field, make sure to use a public address you can easily check from a Web browser.
5. The `Username` field is also in the *form* of an email address, but it does not have to be the same as your actual email address, or even an email that you use. It's helpful to change the username to something that describes the use of the organization. In this workbook we'll use admin-user@workbook.db.
6. Enter the Captcha words shown.
7. Read and then select the checkbox for the `Master Subscription Agreement` and supplemental terms.
8. Click **Sign up**.
9. After signing up, you'll be sent an email with a link that you must click to verify your account. Click the link.
10. Now supply a password, and a security question and answer.

Step 2: Import the Schema, Visualforce Metadata, and Test Data

The tutorials in this workbook are based on a simple schema for an online music business. The schema has just two objects: Album and Track. Your first task is to import the schema into your organization using the Workbench tool. At the same time, you'll also import the Visualforce pages and JavaScript files needed for the HTML5 and hybrid tutorials that come later in this workbook.

1. Download the schema archive from `https://github.com/forcedotcom/SalesforceMobileSDK-Samples/raw/master/CloudTunes-metadata/CloudTunes-force.zip`. If you're using Database.com, download the schema archive from `https://github.com/forcedotcom/SalesforceMobileSDK-Samples/raw/master/CloudTunes-metadata/CloudTunes-database.zip`. Don't extract the contents, leave them zipped up.
2. Navigate to the Workbench at `https://workbench.developerforce.com/`
3. Log in using your Force.com or Database.com credentials and confirm that Workbench may access your data.
4. Click **Migration** > **Deploy**.
5. Click **Choose File** (or **Browse**, depending on your browser), and select the downloaded ZIP file.
6. Enable **Rollback on Error** and **Single Package**.
7. Click **Next** and then **Deploy**.

We'll continue using Workbench to upload data about Albums and Tracks.

1. Download the data set from `https://raw.github.com/forcedotcom/SalesforceMobileSDK-Samples/master/CloudTunes-metadata/CloudTunes-data.txt`.

Note: If the text file opens in your browser rather than downloads, that's OK, you can copy and paste instead.

2. In Workbench, click **Utilities** > **Apex Execute**.
3. Paste in the contents of the `data.txt` file.

Note: The `data.txt` file contains Apex calls that first delete any existing records in the Album and Track objects, and then repopulate those albums with album and track data.

4. Click **Execute**.

That's it! The schema and data are both deployed to your environment. To see the objects and data you created:

- On Force.com, click your name, then **Setup** > **App Setup** > **Create** > **Objects**.
- On Database.com, you can explore it by navigating to **Create** > **Objects**.

Tell Me More....

The Track object is in a master-detail relationship with the Album object. It's a simple data model that allows us to list tracks within albums, the price of individual tracks, and aggregate the prices of tracks to create a price for the albums.

The Album object has a standard name field, Name, of type Text, a Description field, of type Long Text Area, a Released On field, of type Date, and a roll-up summary field, summing the Price field of Track:

Action	Field Label	API Name	Data Type
Edit | Del	Description	Description__c	Long Text Area(5000)
Edit | Del	Price	Price__c	Roll-Up Summary (SUM Track)
Edit | Del	Released On	Released_On__c	Date

The Track object has: a standard name field, Name, of type Text, a Price field, of type Currency (5, 2), and a master-detail relationship to Album:

Action	Field Label	API Name	Data Type
Edit | Del	Album	Album__c	Master-Detail(Album)
Edit | Del	Price	Price__c	Currency(5, 2)

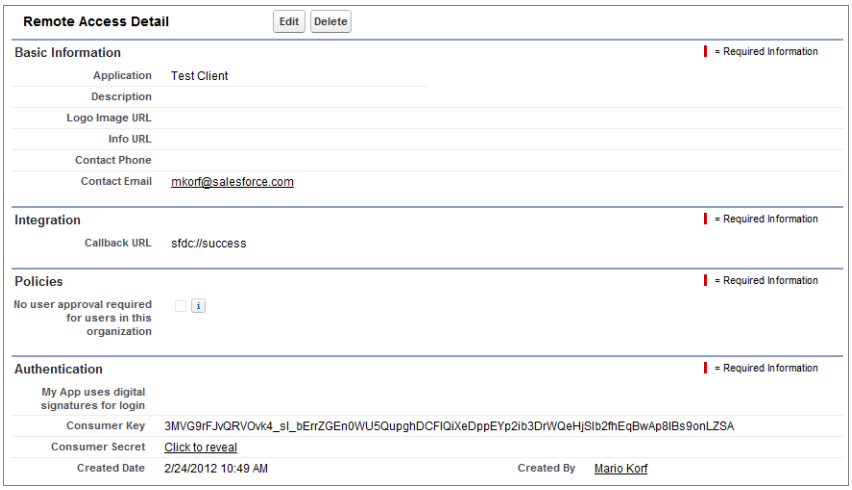
Step 3: Configure OAuth

You don't want just anyone accessing your data, so there needs to be authentication on the mobile device. The Mobile SDK provides authentication via OAuth 2.0. Using OAuth 2.0, the client application delegates the authentication to a provider (in this case Database.com or Force.com), which in turn issues an access token if the user successfully authenticates. Thereafter, as long as a valid access token accompanies all API interactions, you don't need to worry about authentication.

Before an application can use OAuth, you have to configure your environment.

1. Log into your Database.com or Force.com instance as an administrator.
2. Navigate to **App Setup** > **Develop** > **Remote Access**.
3. Click **New**.
4. For `Application`, enter a name such as `Test Client`
5. For `Email`, enter your email address.
6. For `Callback URL`, enter `sfdc://success`
7. Click **Save**.

Important: The detail page for your remote access configuration will display a consumer key. It's a good idea to copy/paste the key into a text file, as you'll need this later.



Tell Me More....

- If you're familiar with OAuth, you'll understand that the callback URL is traditionally the URL that a user's browser is redirected to after a successful authentication transaction. For mobile applications, the callback URL means something subtly different. Native applications on a mobile device can't be redirected to a URL (you're building an application, not a browser). So instead, the callback URL can be thought of as a signal. If, during the OAuth dance, the callback URL is sent to the mobile application, then the application can detect that it has successfully authenticated.
- For more information on OAuth, see “Authenticating Remote Access Application OAuth” in the online help.

Step 4: Explore the REST API

Let's quickly explore the REST API using Workbench. This will give you a feel for some of the method calls in the REST API and their return values, and help make sense of the calls made in the following tutorials.

1. Go to Workbench at https://workbench.developerforce.com/, and log in again if needed.
2. Click **Utilities** > **REST Explorer**.
3. In the text area, enter the following

```
/services/data/v24.0/query/?q=SELECT id, name, price__c FROM album__c
```

4. Ensure **GET** is selected, and then click **Execute**.
5. Now click **Show Raw Response**.

The REST API call (to the query resource, with the query set as a parameter) returns a list of the albums in your database (in this case in the JSON format):

```
{
    "totalSize": 5,
    "done": true,
        "records": [
        {
            "attributes": {
            "type": "Album__c",
            "url": "/services/data/v24.0/sobjects/Album__c/a003000000HAHSGAA5"
        },
```

```
        "Id": "a003000000HAHSGAA5",
        "Name": "Help!",
        "Price__c": 10.29
        },
...
        ]
}
```

Accessing and manipulating data is simply a matter of manipulating URLs, and using standard HTTP verbs like GET, POST, and DELETE. All of the URLs start with `/services/data/`, followed by a version number, followed by a path to the resource. The exact format of the URL is described in the REST API Developer's Guide, but these examples give you a feel for them.

Note how this list returns the albums within a records element, embedded in the response. The response also contains the ID of each record. For example, in the above output, the Help! album has an ID of `a003000000HAHSGAA5`. It also provides the REST URL needed for retrieving the contents of a particular record.

1. In the text area, enter the value of the URL attribute. In our case, it's `/services/data/v24.0/sobjects/Album__c/a003000000HAHSGAA5`, but your org will be different.
2. Click **Execute**, and then **Show Raw Response**.

The server will respond with the details of the album resource, something like this.

```
{
  "attributes": {
    "type": "Album__c",
    "url": "/services/data/v24.0/sobjects/Album__c/a003000000HAHSGAA5"
  },
  "Id": "a003000000HAHSGAA5",
  "OwnerId": "00530000004qf3jAAA",
  "IsDeleted": false,
  "Name": "Help!",
  "CreatedDate": "2011-08-12T10:44:29.000+0000",
  "CreatedById": "00530000004qf3jAAA",
  "LastModifiedDate": "2011-08-12T10:44:29.000+0000",
  "LastModifiedById": "00530000004qf3jAAA",
  "SystemModstamp": "2011-08-12T10:44:29.000+0000",
  "Description__c": "North American release",
  "Released_On__c": "1965-08-06",
  "Price__c": 10.29
}
```

This result contains a few attributes describing the record, all your custom fields, as well as a number of system fields. Note the form of the URL. In this case the resource, `Album__c`, is there in the URL. `album__c` is the API name of the Album object you created earlier. So a GET to `/services/data/v24.0/sobjects/<Object Type Name>/<ID>` returns the record of the given identifier and object. In fact, a `DELETE` to the same URL will delete the record, and a `POST` to `/services/data/v24.0/sobjects/Album__c/` (with the correct body) will create a new record.

There are two important elements of interacting with the REST API that are masked in the above interactions:

- Every HTTP request has a header element, `Authorization`, which contains an access token. The access token was returned as part of logging in to your environment.
- Every HTTP request must be made to a base URL — the URL of the Database.com or Force.com instance. This URL is also returned as part of the OAuth authentication process.

You can find these two pieces of data in Workbench:

- Navigate to **Info** > **Session Information** and expand **Connection**.

The Endpoint value starts with `https://na1.salesforce.com/services`—that's where the HTTP requests are being sent. Note that your endpoint might well be different. The Session Id contains the access token.

If you dig deep enough into the code in the following tutorials, you will always find that building applications on the mobile devices contains these elements:

- OAuth dance
- Retrieval of the instance URL and access token
- Use of these in all subsequent interactions with the REST API

Summary

In this tutorial, you signed up for a Force.com Developer Edition organization or a Database.com base organization. You then used the Workbench tool to quickly create two objects, Album and Track, and populated their fields with data. You then logged into the database as an administrator and created a remote access application through which you can make authenticated calls to the database. Finally, you used Workbench to examine some key aspects of the REST API and how we interact with the database.

For more information on the REST API, see the Force.com REST API Developers Guide.

Tutorial #2: Creating a Native iOS App

In this tutorial, you create and configure an iOS application that accesses the album and track data created in Tutorial 1. When completed, this application runs in the Xcode simulator.

This tutorial installs an Xcode project template, which creates most of the basic elements of the application, and includes a developer preview of the Mobile SDK.

Prerequisites

- You need to have Xcode 4.2 (or later) installed
- You need to complete Tutorial 1

Step 1: Download and Install the Salesforce Mobile SDK for iOS

The Salesforce Mobile SDK Native App Template makes it easy to create new iOS projects. Let's use it.

1. In your browser, navigate to the Mobile SDK iOS GitHub repository: `https://github.com/forcedotcom/SalesforceMobileSDK-iOS`.
2. Clone the repository to your local file system by issuing the following command: `git clone git://github.com/forcedotcom/SalesforceMobileSDK-iOS.git`

Note: If you have the GitHub app for Mac OS X, click **Clone in Mac**.

3. Open the OS X Terminal app in the directory where you installed the cloned repository, and run the install script from the command line: `./install.sh`
4. You also need to download the sample app from GitHub: `https://github.com/forcedotcom/SalesforceMobileSDK-Samples/tree/master/iOS/CloudTunesNative`.

Tell Me More....

The install script prepares the entire SDK, including all of the sample apps, for building, including retrieving and building third-party libraries such as `callback-ios` (from PhoneGap). The install script also installs the Xcode project template used in this tutorial.

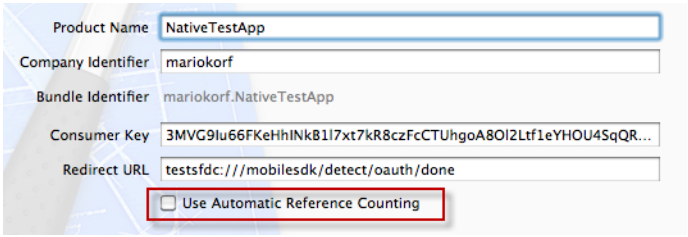
Step 2: Create a New Native App

In this step you create a new project for native iOS. The project template includes a simple application that runs a query on your Database.com account or Force.com organization.

1. Open Xcode and create a new project (Shift-Command-N).
2. Select **Native Force.com REST App** and click **Next**.
3. In the `Choose options for your new project` dialog, enter `NativeTestApp`.

Note: You may also need to enter a Company Identifier prefix if you haven't used Xcode before.

4. Make sure the checkbox for **Use Automatic Reference Counting** is cleared.



5. Click **Next**.
6. Specify a location for your new project and click **Create**.
7. Press **Command-R** and the default template app runs in the iOS simulator.

Note: If you get build errors, make sure **Automatic Reference Counting (ARC)** is turned off.

   a. Select your project in the Navigator.
   b. In the Build Settings tab, toggle the `Objective-C Automatic Reference Counting` value to `No`.

8. On startup, the application starts the OAuth authentication flow, which results in an authentication page. Enter your credentials, and click **Login**.
9. Tap **Allow** when asked for permission

This default template application queries the User object and returns a list of names, probably just your name if you recently signed up.

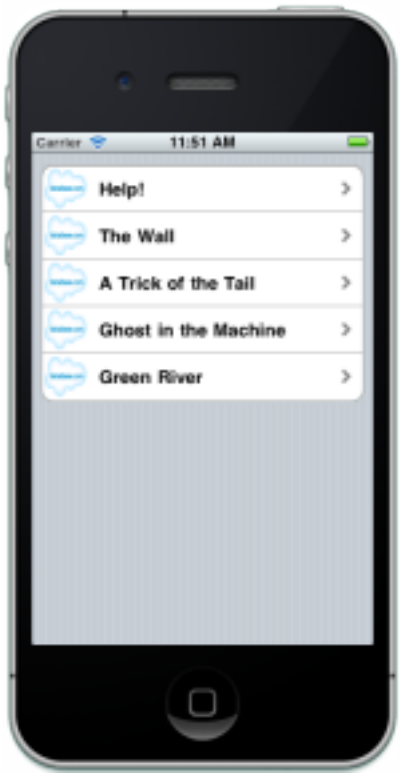
Tell Me More....

The project template fills in the `Consumer Key` and `Redirect URL` for you. The values provided will work for testing purposes on any production or sandbox org; however, if you create an app using this template and wish to publish the app to the iTunes App Store, you will need to enter your own Remote Access object settings from a Production org.

Step 3: Run the CloudTunes App

For convenience, we have created a more complex sample app based on the native template. You can download this sample app from GitHub here:

1. Download the sample app:
   https://github.com/forcedotcom/SalesforceMobileSDK-Samples/tree/master/iOS/CloudTunesNative
2. In Xcode, open `CloudTunesNative.xcodeproj`.
3. Press Command-R and your application runs in the iOS simulator.
4. On startup, the application starts the OAuth authentication flow, which results in an authentication page. Enter your credentials, and click **Login**.
5. You now have to explicitly grant the application permission to access your database. Click **Allow**.
6. The application then performs a query of album records from the database and displays them in a table.



Step 4: Examine the Source Files

By comparing the project's default template app with the CloudTunesNative app you can see that we've modified AppDelegate and RootViewController and added a few more files to the Classes group.

- `AppDelegate` sets up `RootViewController` in a way slightly different from the simple app template. In this case we rely on the `UINavigationController` to provide navigation stack functionality, so we add `RootViewController` as the root view controller to a `UINavigationViewController`.
- `RootViewController` has been modified to be a subclass of `UITableViewController`, and it retrieves a list of albums. `RootViewController` acts as the data source and delegate for the `UITableView`.
- `TrackDetailsViewController` displays details on specific songs stored in an album.

Dependencies

The Dependencies folder lists a number of libraries used in the application. We're using:

- `RESTKit` as a way of simplifying the interactions with the REST API.
- `SalesforceSDK` as the main Objective-C interface to the API. Look at the `SFRestAPI.h` header file to see the calls and examples of how to use them.
- `SalesforceOAuth` as an implementation of OAuth 2.0 in Objective-C.

Summary

In this tutorial you learned how an iOS application retrieves data from Force.com. The tutorial made use of a project template that enables you to easily create new iOS projects that already have the necessary dependencies installed. The Mobile SDK provides wrappers around the underlying REST API, making OAuth painless, and seamlessly ensuring that OAuth access tokens are sent on all API calls.

Tutorial #3: Creating a Native Android App

In this tutorial, you create and configure an Android application that accesses the album and track data created in Tutorial 1. When completed, this application runs in the Android Emulator.

Prerequisites

- Java JDK 6.
- Android SDK, version 14 or later—http://developer.android.com/sdk/installing.html.
- Eclipse 3.6. See http://developer.android.com/sdk/requirements.html for other versions.
- Android ADT (Android Development Tools) plugin for Eclipse, version 14 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 2.2). To learn how to set up an AVD in Eclipse, follow the instructions at http://developer.android.com/guide/developing/devices/managing-avds.html.

Step 1: Create an Android Project in Eclipse

The code for this tutorial is available as an Android project that you can download and install in Eclipse. The project has most of the Android-specific plumbing (such as setting up the various Activities, permissions etc.) included so that you can just focus on the pieces of the application that interact with Force.com.

The first step is to download the Cloud Tunes project and Mobile SDK.

1. Clone the SDK for Android from GitHub: https://github.com/forcedotcom/SalesforceMobileSDK-Android
2. In the directory where you installed the cloned repository, run the install script from the command line: `./install.sh`

Note: Windows users, run the following install script from the command line: `cscript install.vbs`

3. Launch Eclipse and use the Java perspective.
4. Import the Cloud Tunes project by selecting **File** > **Import** > **General** > **Existing Projects into Workspace**.
5. Click **Next** and browse to the `/native/SampleApps/CloudTunes` directory where you installed the SDK. Open the `CloudTunes` folder.
6. Click **Finish**.

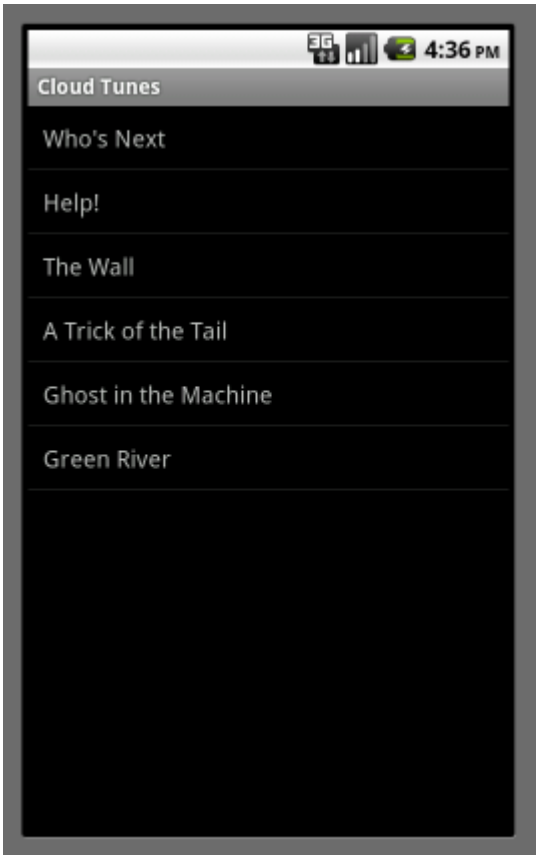
Step 2: Run the App

Let's run the application in a local Android Emulator.

1. In the Package Explorer, highlight the CloudTunes project.
2. Select **Run** > **Run** and select **Android Application**.

Note: The Android Emulator might be slow to boot up and can take several minutes to start.

3. Upon startup, you will be presented with the Android home screen. Swipe the lock icon to the right, if necessary, to display the application.
4. Log into your Force.com organization.
5. On the next screen, you'll be asked to approve the Cloud Tunes application. This is part of the OAuth dance. Click **Allow**.
6. After approving the application, you'll be presented with the list of Album records that are stored in your database—this is the `AlbumListActivity.java` class.



7. Click any Album record, and you'll be presented with the list of Tracks—this is the `TrackListActivity.java` class.

Step 3: Examine the Source Files

Let's walk through the code in the classes and see what happened.

1. In the navigation pane, expand the `src/` folder and open `AlbumListActivity.java`.
2. Similarly, open `TrackListActivity.java`.

AlbumListActivity.java

The `getAlbums()` method is invoked from the `onResume()` method after the user has successfully logged in using OAuth. This method encapsulates the simple code necessary to perform SOQL queries using the Mobile SDK.

TrackListActivity.java

The code for `TrackListActivity.java` is similar to `AlbumListActivity.java`. This is to be expected since this Activity performs a very similar action to the one before—that is, querying data using the Mobile SDK and displaying the result in a simple list. The only real difference is the SOQL query that is executed and the related parsing of the response data.

You can see the SOQL and the use of the Mobile SDK in the `getTracks()` method. As before, the `sendAsync()` method of the `RestClient` class is passed the `RestRequest` object, and an `AsyncRequestCallback` object, which gets invoked by the SDK once the query request completes. `The onSuccess()` method of the callback object then processes the JSON response and populates the list of tracks to be displayed by the `ListView`. In addition, the `AsyncRequestCallback` object specifies an `onError()` method, which will be invoked in the event that errors occur when making the request.

Summary

In this tutorial you learned how an Android application retrieves data from Force.com or Database.com using the Salesforce Mobile SDK for Android. The Mobile SDK provides wrappers around the underlying REST API, making OAuth painless, and seamlessly ensuring that OAuth access tokens are sent on all API calls.

Tutorial #4: Creating an HTML5 App

HTML5 lets you create lightweight interfaces without installing software on the mobile device; any mobile, touch or desktop device can access the same interface. In this tutorial, you create an HTML5 application that uses Visualforce to deliver the HTML content and fetches record data from Force.com using JavaScript remoting for Apex controllers. The tutorial also utilizes the jQuery Mobile library for the user interface.

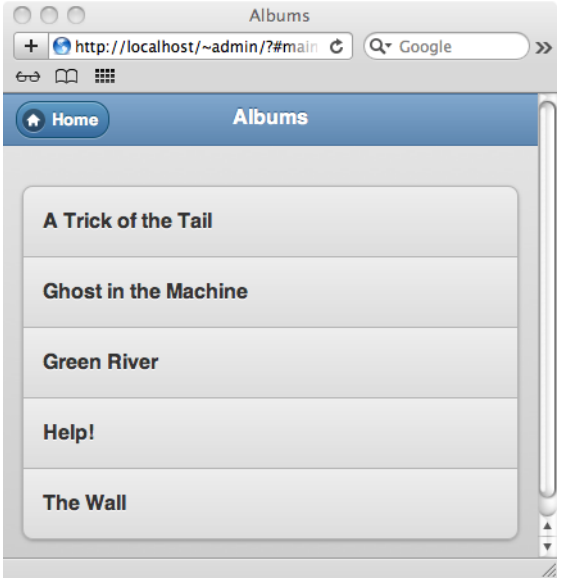
Prerequisites

- You'll need a Force.com organization as configured in Tutorial 1. Since this tutorial uses Visualforce, you can't use Database.com.
- Basic knowledge of Apex and Visualforce.

Step 1: Run the App

You already downloaded and deployed the metadata required for the HTML5 app using the Workbench tool in Tutorial #1: Setting Up the Schema, Step 2: Import the Schema, Visualforce Metadata, and Test Data on page 6. So all there is left to do is run the app and explore how it works.

1. Log into your organization.
2. In the address bar in your browser, replace the string after `.com` with `/apex/CloudTunes`. For example, if you're running on the na1 instance your URL would be `https://c.na1.visual.force.com/apex/CloudTunes`.
3. Press **Enter** and your app will run.



4. To see how the flow works on an HTML5 app, click on an album and then a track, and then click **Home**.

Summary

There are three major components of this application which we'll examine in more detail.

- Visualforce page
- Apex controller
- JavaScript libraries, deployed as static resources

Visualforce Page

The Visualforce page `cloudtunes.page` is the start page of this application and provides the required HTML interface to your application. Open it up and take a look. At the top, it references the Apex class `CloudtunesController` as the controller for this page. This controller provides the required data and operation binding to this Visualforce page.

```
<apex:page showHeader="false" docType="html-5.0" standardStylesheets="false" cache="true"
controller="CloudtunesController" >
```

Following the controller declaration, this page references various CSS stylesheets and JavaScript files from the static resource. The application uses jQuery and jQuery Mobile to control the interface, which accounts for the majority of these includes.

```
<link rel="stylesheet" href="{!URLFOR($Resource.cloudtunes_jQuery,
'jquery.mobile-1.0.1.min.css')}" />
<apex:includeScript value="{!URLFOR($Resource.cloudtunes_jQuery, 'jquery.min.js')}"/>
<apex:includeScript value="{!URLFOR($Resource.cloudtunes_jQuery,
'jquery.mobile-1.0.1.min.js')}"/>
<apex:includeScript value="{!URLFOR($Resource.cloudtunes_offline)}"/>
```

Notice the "`viewport`" meta tag. This improves the presentation of the web content based on the different screen resolutions on different devices.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0,
user-scalable=no;" />
```

The HTML is divided up into `<div>` sections that describe specific pages for the interface. Note that these have identifiers that will be used later. For example, "`detailpage`" is used to identify the HTML segment that will eventually be used to display the album data.

```
<div data-role="page" data-theme="b" id="detailpage">
   <div data-role="header">
      <a href='#mainpage' id="backAlbums" class='ui-btn-left' data-icon='arrow-l'>
         Albums</a>
      <h1>Album</h1>
   </div>
   <div data-role="content">
      <h1 id="AlbumName"></h1>
      <table>
        <tr><td>Price:</td><td id="AlbumPrice"></td></tr>
      </table>
      <input type="hidden" id="AlbumId" />
      <ol id="tracklist" data-inset="true" data-role="listview" data-theme="c"
          data-dividertheme="c">
      </ol>
   </div>
</div>
```

jQuery Mobile is capable of splitting the sections up and treating them as separate portions of an interface. Consider the case when our JavaScript later calls a function like this:

```
$j.mobile.changePage('#detailpage', {changeHash: true});
```

Here, jQuery Mobile will change the page to the with an id of detailpage using a slide animation.

Apex Controller

The Apex class CloudtunesController provides the important methods to fetch records from the database and deliver them to the caller. The first method queryAlbums queries the list of records from the Album__c custom object and returns that list. We put a limit of 20 records on our SELECT clause, but it a real-world scenario you'd want to implement pagination.

```
@RemoteAction
global static List<Album__c> queryAlbums() {
   return [SELECT Id, Name, Price__c FROM Album__c ORDER BY Name LIMIT 20];}
```

The second method queryTracks accepts an album record Id and then queries the related list of records from the Track__c custom object.

```
@RemoteAction
global static List<Track__c> queryTracks(String albumId) {
     return [SELECT Id, Name, Price__c, Album__c, Album__r.Name
             FROM Track__c WHERE Album__c = :albumId
             ORDER BY CREATEDDATE LIMIT 200];}
```

Both of these methods are declared as global static methods and are annotated with @RemoteAction. This is required to allow access to these methods from the Visualforce page using the JavaScript remoting feature.

JavaScript Libraries

The application contains two static resource files: jQuery and cloudtunes.

- The jQuery static resource contains all the JavaScript and stylesheet files for the jQuery and jQuery Mobile libraries.
- The cloudtunes static resource is a JavaScript file that contains the methods to pull data from the Apex controller using JavaScript remoting. This data is then wrapped into appropriate HTML elements and rendered on the Visualforce page.

Let's take a closer look at the methods in this JavaScript file.

```
function getAlbums(callback) {
    $j('#albumlist').empty();
    CloudtunesController.queryAlbums(function(records, e)
        { showAlbums(records, callback) }, {escape:true});
}
```

- In getAlbums(), calls such as $j('#albumlist').empty() are an indication of jQuery at work. In this case, jQuery retrieves the HTML element identified by albumlist, and clears out the HTML, readying it for results.
- The method then makes a call to the Apex controller's queryAlbums() method. This is where the JavaScript remoting magic happens. Visualforce provides all the required plumbing to allow the call to the controller method directly from the JavaScript.
- Finally, a callback function is passed as an argument to queryAlbums() that is automatically invoked once the records are returned from the Apex controller. The showAlbums() function takes these records and displays them.

Now let's take a look at `showAlbums()`.

```
function showAlbums(records, callback) {
   currentAlbums.length = 0;
   for(var i = 0; i < records.length; i++) { currentAlbums[records[i].Id] = records[i]; }

   $j.each(records, function() {
      $j('<li></li>')
      .attr('id',this.Id)
      .hide()
      .append('<h2>' + this.Name + '</h2>')
      .click(function(e) {
         e.preventDefault();
         $j.mobile.showPageLoadingMsg();
         $j('#AlbumName').html(currentAlbums[this.id].Name);
         $j('#AlbumPrice').html('$'+currentAlbums[this.id].Price__c);
         if($j('#AlbumPrice').html().indexOf(".") > 0
               && $j('#AlbumPrice').html().split(".")[1].length == 1) {
            $j('#AlbumPrice').html($j('#AlbumPrice').html()+"0"); //add trailing zero
    }
         $j('#AlbumId').val(currentAlbums[this.id].Id);
         var onTracksLoaded = function() {
         $j.mobile.hidePageLoadingMsg();
         $j.mobile.changePage('#detailpage', {changeHash: true});
    }
         getTracks(currentAlbums[this.id].Id, onTracksLoaded);
    })
     .appendTo('#albumlist')
     .show();
     });

   $j('#albumlist').listview('refresh');
   if(callback != null && typeof callback == 'function') { callback(); }
}
```

- This function gets the records from the callback, loops through them and creates a new list of `<li>` HTML elements to display within the `albumlist` div.
- Notice this function also dynamically attaches a new event to each list item so that when the user clicks the list item, they can browse down to a list of tracks associated with the album. The list of those tracks is fetched using `getTracks()` .

Now let's take a look at `getTracks()`. Functionally, this code is very similar to the `getAlbums()` and `showAlbums()` code. The only significant difference to the code that handled albums is that a different Apex controller method is used, and of course, a different callback function is provided to update the page with the results.

```
function getTracks(albumid, callback) {
  $j('#tracklist').empty();
  CloudtunesController.queryTracks(albumid, function(records, e) {
    showTracks(records,callback) }, {escape:true} );
  return true;
}
```

Now anytime the album name is clicked, a new set of track data will be retrieved and the `itemlist` will be rewritten. Clicking on the track name will rewrite the HTML of the elements displaying the track information and use jQuery Mobile to move to that page. A real application may of course cache this information as well.

Tutorial #5: Creating an iOS Hybrid App

In this tutorial, you create a hybrid iOS application that uses Visualforce within your org. Hybrid applications can run natively on a device, while the interface and business logic can be built with Visualforce, HTML, CSS, and JavaScript, reducing the amount of native code needed to create an application. This tutorial uses the Force.com Hybrid App template of the Salesforce Mobile SDK.

Prerequisites

- Complete Tutorial #1: Setting Up the Schema on page 5 and Tutorial #4: Creating an HTML5 App on page 19.
- Xcode 4.2 or later.

Step 1: Install the Salesforce Mobile SDK for iOS

In this step you download and install the Mobile SDK for iOS. If you've already completed Tutorial #2: Creating a Native iOS App, you can skip this step.

1. In your browser, navigate to the Mobile SDK iOS GitHub repository: `https://github.com/forcedotcom/SalesforceMobileSDK-iOS`.
2. Clone the repository to your local file system by issuing the following command: `git clone git://github.com/forcedotcom/SalesforceMobileSDK-iOS.git`

Note: If you have the GitHub app for Mac OS X, click **Clone in Mac**.

3. Open the OS X Terminal app in the directory where you installed the cloned repository, and run the install script from the command line: `./install.sh`
4. You also need to download the sample app from GitHub: `https://github.com/forcedotcom/SalesforceMobileSDK-Samples/tree/master/iOS/CloudTunesNative`.

Step 2: Create a New Hybrid Force.com App

In this step you add the OAuth Consumer key from Tutorial #1: Setting Up the Schema, Step 3: Configure OAuth on page 7.

1. Open Xcode and create a new project (Shift-Command-N).
2. Select **Hybrid Force.com App** and click **Next**.
3. In the `Choose options for your new project` dialog, enter `CloudTunes`.

Note: You may also need to enter a Company Identifier prefix if you haven't used Xcode before.

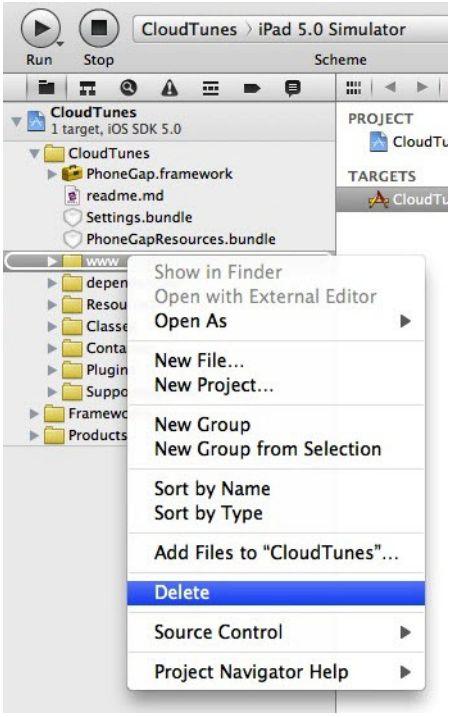
4. Make sure the checkbox for **Use Automatic Reference Counting** is cleared.
5. Click **Next**.

6. Specify a location for you new project and click **Create**.

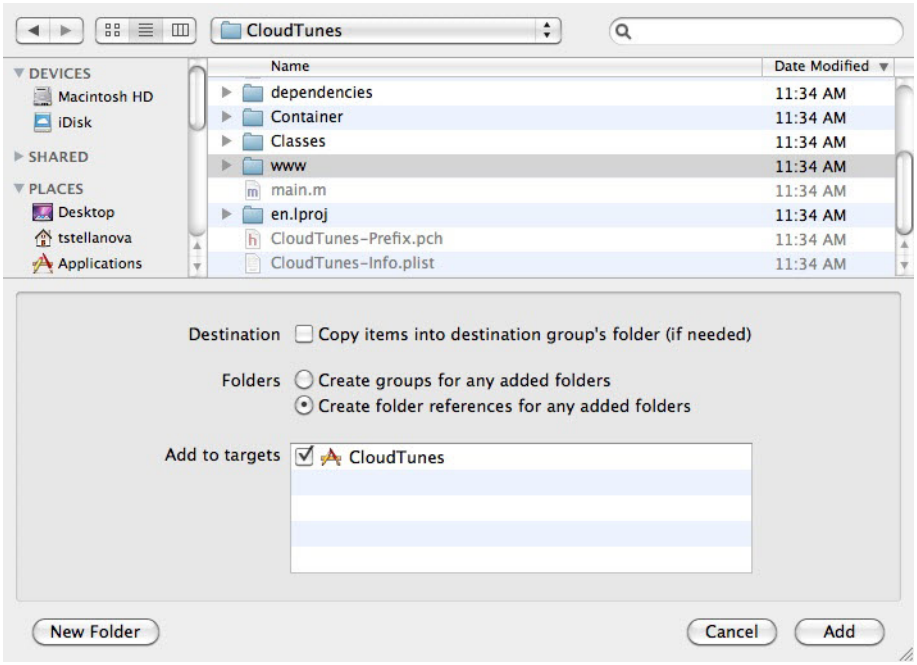
Step 3: Configure the Hybrid Project

Next, we need to do a bit of cleanup of the newly created project to overcome some limitations in the Xcode project creation process. Specifically, the `www` folder is added as a group instead of a folder, so you need to delete it as a group and add it back as a folder. We'll work with the Project Navigator, which is typically shown as a hierarchy of files in the left pane of Xcode.

1. Right-click on the **www** group in the Project Navigator and select **Delete**.



2. When prompted, select **Remove References Only** to remove references to this folder.
3. Right-click the **CloudTunes** group in the Project Navigator and select **Add Files to CloudTunes**.
4. In the file selection dialog, ensure that **Create folder references for any added folders** is selected.
5. Navigate to the `CloudTunes` folder next to the `CloudTunes.xcodeproj`.
6. Select the **www** folder within it and click **Add**.



Step 4: Set Up Your Hybrid App for Visualforce

In this step you edit your project configuration to access the CloudTunes Visualforce app.

1. Open the `www` folder in the Project Navigator, and click on the `bootconfig.js` file.
2. Comment out the line that defines the `startData` variable (`var startData = new SFHybridApp.LocalAppStartData();`), and uncomment the line below it (`var startData = new SFHybridApp.RemoteAppStartData("/apex/BasicVFPage");`).
3. The `startData` variable defines information about where your initial Visualforce page resides. Change the `RemoteAppStartData` argument to point to the CloudTunes start page, `"/apex/cloudtunes?context=container"`. This tells the app container what Visualforce URL to load and passes a parameter to the app indicating the app is running inside the container.

Tell Me More....

- Note the `remoteAccessConsumerKey` and `oauthRedirectURI` settings. These are populated with the values you provided when we created the project from the project template, and are likely the default test values provided by the template. If and when you are ready to publish your app, you must change these values to match a Remote Access object you've created on Production.
- We use the `"context"` variable to let our app know about its runtime environment, so that it can take advantage of mobile platform capabilities, and disable those features when running inside a web browser.
- Note that `autoRefreshOnForeground` is set to `true` by default. This tells the app container to automatically refresh your login session each time the hybrid app is brought to the foreground. This can help prevent your users' login sessions from timing out by guaranteeing that they have a fresh session each time the app becomes the foreground app.

Step 5: Build and Run the App

You can now compile and run the application.

1. Press **Command-R** to build and run the app.

   Note: If you get build errors, make sure **Automatic Reference Counting (ARC)** is turned off.

   a. Select your project in the Navigator.
   b. In the Build Settings tab, toggle the `Objective-C Automatic Reference Counting` value to `No`.

2. You will be presented with a standard Salesforce mobile OAuth login screen. Log into the org where you installed the CloudTunes Visualforce metadata.
3. Click **Allow**, and after a few moments the CloudTunes app will become visible.

Tell Me More....

On the OAuth Approval screen, notice that the name of your Remote Access object (SalesforceMobileSDK Sample App) is visible on this screen. Before you release this app to customers you will need to create your own Remote Access object and update the `remoteAccessConsumerKey` and `oauthRedirectURI` settings in `bootconfig.js`. Currently these are populated with the values you provided when we created the project from the Xcode template, and are likely the default test values provided by the template. If and when you are ready to publish your app to the iTunes App Store, you must change these values to match a Remote Access object you've created on Production.

Summary

In this tutorial, you ran a hybrid iOS application that uses Visualforce, CSS, and JavaScript within your org. Compared to a native application, you used a lot less code in the hybrid application, while retaining access to the native features of the device.

Tutorial #6: Creating an Android Hybrid App

In this tutorial you create a hybrid Android application using the HTML5 code from Tutorial 4 as a starting point. Hybrid applications can run natively on a device however the interface and business logic can be built with HTML, CSS and JavaScript reducing the amount of native code needed to build the application.

Prerequisites

- Complete Tutorial #1: Setting Up the Schema on page 5 and Tutorial #4: Creating an HTML5 App on page 19.

Step 1: Install the Mobile SDK for Android

In this step, you download and install the Mobile SDK for Android. If you've already completed Tutorial #3: Creating a Native Android App, you can skip this step.

1. Clone the SDK for Android from GitHub: `https://github.com/forcedotcom/SalesforceMobileSDK-Android`
2. In the directory where you installed the cloned repository, run the install script from the command line: `./install.sh`

Note: Windows users, run the following install script from the command line: `cscript install.vbs`

Step 2: Load the VFConnector App in Eclipse

The easiest way to understand how to create a container app for a Visualforce application is to walk through a working example. We will be creating a container-based app that points at the CloudTunes Visualforce sample that you've worked with in the previous tutorials. To do this, we'll start with the existing VFConnector hybrid sample app from the Salesforce Mobile SDK for Android and modify it to point at our CloudTunes app.

1. In Eclipse, click **File** > **Import**.
2. Expand the General section at the top of the dialog.
3. Select **Existing Projects into Workspace** and click **Next**.
4. Make sure **Select root directory** is selected and click **Browse**.
5. Navigate to where you downloaded the Mobile SDK for Android and to the `/hybrid/SampleApps` folder. Click **Open**.
6. You should see two projects in the Projects list, and both of them are selected by default. We will be working with the `VFConnector` app, but you may keep the `ContactExplorer` project selected as well, it will not affect this tutorial.
7. Click **Finish**.

Step 3: Transforming the VFConnector App Into CloudTunes

Now that you have VFConnector configured in Eclipse, let's walk through the steps you can take to transform this app into the CloudTunes app. You can follow a similar workflow to create your own hybrid apps from the VFConnector sample, or simply use your experience from this tutorial to create your hybrid apps from scratch.

The first thing we need to change is the app configuration that determines how to access the CloudTunes Visualforce application. This configuration information is contained in the `bootconfig.js` file.

1. In the Package Explorer in Eclipse, in the VFConnector project, expand the `assets/www` folder, and open `bootconfig.js`.
2. Change the `remoteAccessConsumerKey` and `oauthRedirectURI` variables to the respective values you generated for your remote access object in the first tutorial.
3. The `startData` variable defines information about where your initial Visualforce page resides. Change the `RemoteAppStartData` argument to point to the CloudTunes start page, `"/apex/cloudtunes?context=container"`. This tells the app container what Visualforce URL to load and passes a parameter to the app indicating the app is running inside the container.
4. Notice the `oauthScopes` variable sets the scope of permissions your app needs. These should currently be set to `"visualforce"` and `"api"`, and shouldn't need to be changed for this tutorial.

After you configure these values to point to the CloudTunes Visualforce app, you should be able to run the app from Eclipse, log in, and see your Visualforce application loaded into your new native container!

Tell Me More....

- Note the `remoteAccessConsumerKey` and `oauthRedirectURI` settings. These are populated with the values you provided when we created the project from the project template, and are likely the default test values provided by the template. If and when you are ready to publish your app, you must change these values to match a Remote Access object you've created on Production.
- We use the `"context"` variable to let our app know about its runtime environment, so that it can take advantage of mobile platform capabilities, and disable those features when running inside a web browser.
- Note that `autoRefreshOnForeground` is set to `true` by default. This tells the app container to automatically refresh your login session each time the hybrid app is brought to the foreground. This can help prevent your users' login sessions from timing out by guaranteeing that they have a fresh session each time the app becomes the foreground app.

Step 4: Changing the Mobile App Identity Configuration

There are also a number of places in the app that we'll want to configure to establish the identity of the CloudTunes mobile application to the Android ecosystem.

1. Open `AndroidManifest.xml` file by right-clicking on it and selecting **Open With** > **Android Manifest Editor**. You can find this file at the root level of your `VFConnector` project in Eclipse.
2. At the top of the first tab, you'll see the Package field. This contains a unique identifier to use out on the Android Market to differentiate your app's code and resources. Even though we won't be deploying this app to the Android Market, let's follow the given format and set this to something unique for CloudTunes.
3. Change the value to `com.salesforce.samples.cloudtunes`, then save the file. In the following dialog asking if you want to change launch configurations, click **Yes**.

4. You'll notice that your project has generated a bunch of build errors! This is because your package name in the manifest file no longer matches the package name of your source files. Let's fix the source files package.
5. Expand the `src/` folder in the project.
6. Right click the package name (currently `com.salesforce.samples.vfconnector`) and select **Refactor** > **Rename**.
7. You will be prompted to give a new name. Change this value to `com.salesforce.samples.cloudtunes`, and click **OK**.
8. You will be presented with a warning that this package already exists in the `gen/` folder. The change to the manifest file created this package, so it's okay. Click **Continue**. Your build errors should disappear.
9. Finally, expand the `res/values/` folder, and open `rest.xml`.
10. Change `com.salesforce.samples.vfconnector` to `com.salesforce.samples.cloudtunes`. We've now successfully updated the package name, and your unique application could now be submitted to the Android Market!
11. Next, if you click over to the Application tab of the `AndroidManifest.xml` document, you should see the Label field, set to `@string/app_name`. This is how your app will be labelled on the device. `@string` is a shorthand convention for pointing to a well-known string resource/configuration location. Let's find that resource, and update our app label to reflect our newly-minted CloudTunes app.
12. Expand the `res/values` folder of the project.
13. Right-click on `strings.xml` and select **Open With** > **Android Resource Editor**.
14. Click on the **app_name** row of the left pane.
15. In the righthand pane, you'll see the value associated with the **app_name** configuration option. Change it, from `VFConnector` to `CloudTunes`, and save the file. Now the app is properly labelled as `CloudTunes`.

Step 5: Change the Code Artifact Names and Locations

There are a couple of remaining code artifacts that we should change, for completeness sake. First, let's change the name of the project to CloudTunes and then we'll change our app name in the code from `VFConnectorApp.java`, to `CloudTunesApp`.

1. Right-click on the project folder and select **Refactor** > **Rename**.
2. Change the name to `CloudTunes`, and click **OK**.
3. Expand the `src/` folder and our renamed package.
4. Right-click on `VFConnectorApp.java` and select **Refactor** > **Rename**
5. Change the name to `CloudTunesApp` and click **Finish**.

Summary

Congratulations! You have rebranded the default VFConnector app to be CloudTunes. You can follow this same approach with your own apps, to create Visualforce-based apps that can run natively on an Android device.

Tutorial #7: Securely Caching Data Offline in Hybrid Apps

Mobile devices can lose connection at any time, and some environments like hospitals and airplanes prohibit connectivity, so it's important to make sure your app still functions when offline. This tutorial shows how to add offline capability for caching pages and securely storing sensitive business data. This section requires the completion of the iOS or Android hybrid tutorials.

In this tutorial, you'll modify the Visualforce–based CloudTunes application to enable offline functionality by using the HTML5 manifest and the Smartstore module of the Salesforce Mobile SDK. This makes it possible for you to securely store business data on a device, allowing users to browse the albums and related tracks, even when the device is not connected to the Internet.

This tutorial requires a few edits to Visualforce pages and related JavaScript files in order to use the Smartstore JavaScript library. Smartstore can also be used to cache direct REST API results in non-Visualforce and native applications.

Prerequisites

- You'll need a Force.com organization as configured in Tutorial 1. Since this tutorial uses Visualforce, you can't use Database.com.
- You'll need to complete Tutorial #4: Creating an HTML5 App and Tutorial #5: Creating an iOS Hybrid App.
- If you previously deployed the package for the 1.0 version of the Mobile SDK workbook, please re-deploy the updated package and follow the deployment steps in the hybrid tutorials.

Step 1: Update the `cloudtunes_offline` JavaScript File

You deployed `cloudtunes_offline.js` and `phonegap-1.2.0.js` as static resources in Tutorial #4: Creating an HTML5 App, but you didn't use it. This file is similar to cloudtunes.js, but has some additional code for data storage and retrieval. This additional code is currently commented out, so you'll start by uncommenting this code so we can walk you through it.

1. Log in to your Force.com organization and click <***Your Name***> > **Setup** > **App Setup** > **Develop** > **Static Resources**.
2. Click **cloudtunes_offline**, and on the following page, right click on the **View File** link and click **Save Link As** to save `cloudtunes_offline.js` on your local machine.
3. Open `cloudtunes_offline.js` in a text editor.
4. Uncomment the code `checkCacheAndPrepareSession()` inside the `init()` method. This method checks if the HTML5 cache manifest has triggered an application update, and then waits to reload the page until the latest codebase is fetched from the server.
5. Next, uncomment the code `resetOfflineStore()`. This method will create and cleanup the SmartStore tables that are used to store the Albums and Tracks data for offline use. The SmartStore table clean-up is only triggered when the app is online, as the fresh data is being pulled from the server.
6. Next, un-comment the code `addOfflineAlbums(records)` inside the `getAlbums()` method. `addOfflineAlbums()` stores the albums data, which is fetched using JS remoting call to Apex controller, in the SmartStore.
7. To enable offline albums data fetch from SmartStore, uncomment the `else` section inside `getAlbums()`. In this code, we are calling `fetchOfflineAlbums()`, which queries the stored album data from the SmartStore and then renders it on the page using the `showAlbums()` method.
8. Similar to the previous steps, uncomment the code inside the `getTracks()` method. The method `addOfflineTracks()` allows us to store the Tracks data offline whenever we fetch the new data from Salesforce.
9. Save `cloudtunes_offline.js` and then upload it again into Salesforce.

10. On the `cloudtunes_offline` static resource detail page click **Edit**. Then click **Choose File** and select the updated `cloudtunes_offline.js` file. Click **Save**.

Tell Me More....

There's a lot going on here; let's take a look behind the scenes.

- The `phonegap-1.2.0.js` and `SFSmartStorePlugin.js` JavaScript files provide an interface to various native OS features which you'll use to access the offline storage features of the mobile container.
- In the code you uncommented, the functions `addOfflineAlbums()`, `fetchOfflineAlbums()`, `addOfflineTracks()`, and `fetchOfflineTracks()` are defined in the `cloudtunes_smartstore.js` static resource file. This file was created during the metadata deploy process of Tutorial #4: Creating an HTML5 App. You should look at this JavaScript file to see how SmartStore soups are created and how the data is stored and fetched from them.
- Also in this CloudTunes example, we are doing a lazy storing of records into SmartStore. By lazy storing, we mean that we are not actively pulling and storing all the tracks data from Salesforce, but instead we are pulling and then storing the list of only those albums and tracks that the user has fetched from server while browsing the CloudTunes application.
- We use the `"context"` variable to let our app know about its runtime environment, so that it can take advantage of mobile platform capabilities and disable those features when running inside a web browser. When running inside the container, the app and the Smartstore must be initialized in the `onDeviceReady()` function provided by PhoneGap.

Step 2: Update the Visualforce Page

In this step, you'll update the cloudtunes Visualforce page to include the new JavaScript files to enable offline storage. We will also update the page to include a HTML5 cache manifest setting that will enable the Hybrid app to cache the UI static elements (HTML, CSS, JS and images), and hence enable the offline view of data.

1. Click <***Your Name***> > **Setup** > **App Setup** > **Develop** > **Pages**.
2. Click the **Edit** link next to the cloudtunes Visualforce page.
3. Modify the `<apex:includeScript>` tag that specifies the `$Resource.cloudtunes` static resource to now use `$Resource.cloudtunes_offline` instead. The updated tag should look like this:

```
<apex:includeScript value="{!URLFOR($Resource.cloudtunes_offline)}"/>
```

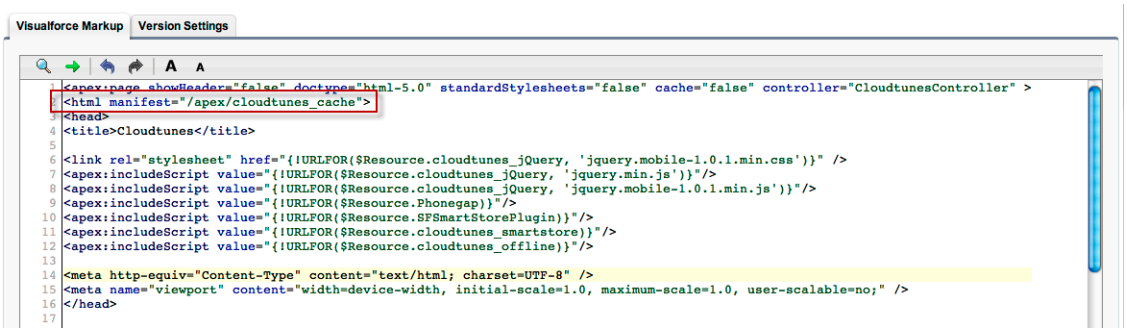
4. Now add the following code to the Visualforce page after all the script tags.

```
<link rel="stylesheet" href="{!URLFOR($Resource.SFHybridAppCss)}" />
<apex:variable var="pgUrl" value="{!phoneGapUrl}"/>
<apex:includeScript value="{!URLFOR($Resource.Phonegap, pgUrl)}"/>
<apex:includeScript value="{!URLFOR($Resource.SFHybridAppJs)}"/>
<apex:includeScript value="{!URLFOR($Resource.SFSmartStorePlugin)}"/>
<apex:includeScript value="{!URLFOR($Resource.cloudtunes_smartstore)}"/>
```

5. Finally update the top `<html>` tag to include the reference to the Cache Manifest Visualforce page, `cloudtunes_cache`. This Visualforce page, along with metadata, was already deployed in your org in Tutorial #1: Setting Up the Schema, Step 2: Import the Schema, Visualforce Metadata, and Test Data on page 6. The updated tag should look like this

```
<html manifest="/apex/cloudtunes_cache">
```

6. Click **Save**.



Tell Me More....

The reason we include the PhoneGap and SFSmartStorePlugin static resources on the Visualforce page is because, when used within the Salesforce container, these JavaScript files provide the interface to native OS features, such as the camera, smartstore etc. The static resource `cloudtunes_smartstore` contains the utility methods used by `cloudtunes_offline.js`.

Cache Manifest is a new HTML5 property which tells the browser to cache certain content, such as HTML pages, JavaScript, CSS, image files etc., on the client's browser. This not just enables quick page loads but also helps in running the app even when the device is not connected to the Internet. Look at the `cloudtunes_cache` Visualforce page to see how to use the caching capability of new HTML5 enabled browsers.

For more information about HTML5 manifest, please follow the tutorial found at http://www.html5rocks.com/en/tutorials/appcache/beginner/.

Step 3: Run the Hybrid App Offline

You can now run the hybrid application that you created earlier. Before doing this, ensure that your machine and the iOS device/Simulator are connected to the Internet.

1. Run the app.
2. Now browse through the Albums and Tracks list within the application. While doing this, the application will store the browsed Albums and Tracks data for offline use.
3. Now stop the app. Disconnect the device from Internet or put your emulator in offline mode.
4. Relaunch the CloudTunes hybrid in offline mode and see that you can still browse the Albums and Tracks even while you're not connected.

Summary

Congratulations, you've created a hybrid application that securely caches data offline! Now when a mobile device suddenly loses connectivity, people can still use your app.

Now that you're finished with the workbook, you might be wondering where to go for additional resources.

- For the latest news and articles, see the Mobile SDK landing page : http://wiki.developerforce.com/page/MobileSDK

- Make sure to keep up date with our rapidly evolving GitHub repository for iOS https://github.com/forcedotcom/SalesforceMobileSDK-iOS and Android https://github.com/forcedotcom/SalesforceMobileSDK-Android.