Version 29.0: Winter '14

# Salesforce Mobile SDK Development Guide

## Salesforce.com Mobile Development


Last updated: July 30, 2013

# Table of Contents

# Chapter 1

# Introduction to Mobile Development

**In this chapter ...**



Force.com has proven itself as an easy, straightforward, and highly productive platform for cloud computing. Developers can define application components, such as custom objects and fields, workflow rules, Visualforce pages, and Apex classes and triggers, using point-and-click tools of the Web interface, and assembling the components into killer apps. As a mobile developer, you might be wondering how you can leverage the power of the Force.com platform to create sophisticated apps.

The Mobile SDK seamlessly integrates with the Force.com cloud architecture by providing:

- SmartSync Data Framework for accessing Salesforce data through JavaScript
- Secure offline storage
- Data syncing for hybrid apps
- Implementation of Force.com Connected App policy that works out of the box
- OAuth credentials management, including persistence and refresh capabilities
- Wrappers for Salesforce REST APIs
- Libraries for building native iOS and Android applications
- Containers for building hybrid applications

**Note:**

Be sure to visit Salesforce Platform Mobile Services website regularly for tutorials, blog postings, and other updates.

# Intended Audience

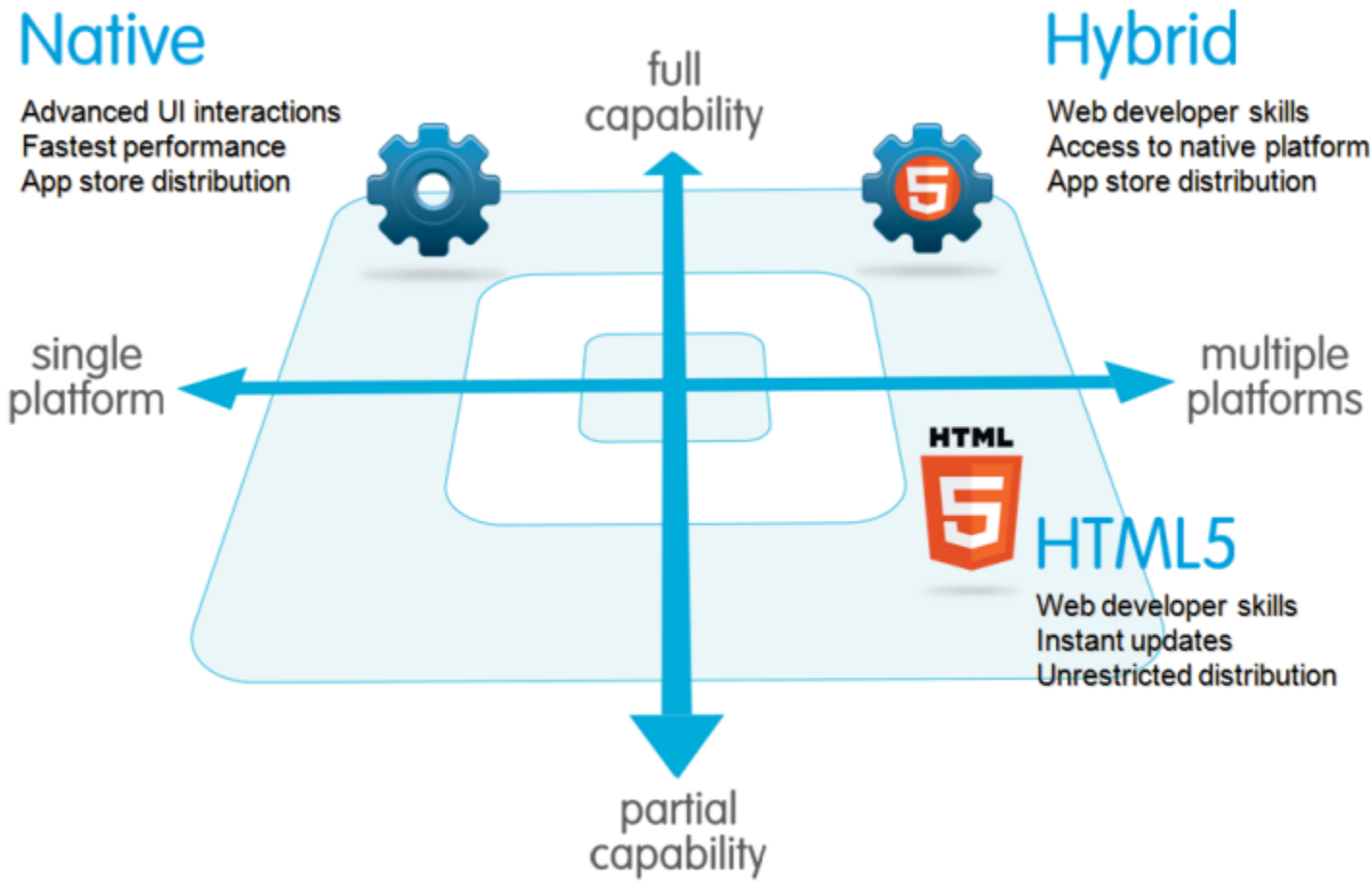
This guide is primarily for developers who are already familiar with mobile technology, OAuth2, and REST APIs, and who probably have some Force.com experience. But if that doesn't exactly describe you, don't worry. We've tried to make this guide usable by a wider audience. For example, you might be a Salesforce admin tasked with developing a new mobile app to support your organization, or you might be a mobile developer who's entirely new to Force.com. If either of those descriptions fit you, then you should be able to follow along just fine.

If you're an admin setting up users for mobile devices, you're probably looking for the Salesforce Mobile Implementation Guide.

# About Native, HTML5, and Hybrid Development

Many factors play a part in your mobile strategy, such as your team's development skills, required device functionality, the importance of security, offline capability, interoperability, and so on. In the end, it's not just a question of what your app will do, but how you'll get it there. The Mobile SDK offers three ways to create mobile apps:

- **Native** apps are specific to a given mobile platform (iOS or Android) and use the development tools and language that the respective platform supports (for example, Xcode and Objective-C with iOS, Eclipse and Java with Android). Native apps look and perform best but require the most development effort.
- **HTML5** apps use standard web technologies—typically HTML5, JavaScript and CSS—to deliver apps through a mobile Web browser. This "write once, run anywhere" approach to mobile development creates cross-platform mobile applications that work on multiple devices. While developers can create sophisticated apps with HTML5 and JavaScript alone, some challenges remain, such as session management, secure offline storage, and access to native device functionality (such as camera, calendar, notifications, and so on).
- **Hybrid** apps combine the ease of HTML5 Web app development with the power of the native platform by wrapping a Web app inside the Salesforce container. This combined approach produces an application that can leverage the device's native capabilities and be delivered through the app store. You can also create hybrid apps using Visualforce pages delivered through the Salesforce hybrid container.




## Native Apps

Native apps provide the best usability, the best features, and the best overall mobile experience. There are some things you get only with native apps:

- **Fast graphics API**—the native platform gives you the fastest graphics, which might not be a big deal if you're showing a static screen with only a few elements, or a very big deal if you're using a lot of data and require a fast refresh.
- **Fluid animation**—related to the fast graphics API is the ability to have fluid animation. This is especially important in gaming, highly interactive reporting, or intensely computational algorithms for transforming photos and sounds.
- **Built-in components**—The camera, address book, geolocation, and other features native to the device can be seamlessly integrated into mobile apps. Another important built-in component is encrypted storage, but more about that later.
- **Ease of use**—The native platform is what people are accustomed to. When you add that familiarity to the native features they expect, your app becomes that much easier to use.

Native apps are usually developed using an integrated development environment (IDE). IDEs provide tools for building, debugging, project management, version control, and other tools professional developers need. You need these tools because native apps are more difficult to develop. Likewise, the level of experience required is higher than in other development scenarios. If you're a professional developer, you don't have to be sold on proven APIs and frameworks, painless special effects through established components, or the benefits of having all your code in one place.

## HTML5 Apps

An HTML5 mobile app is basically a web page, or series of web pages, that are designed to work on a small mobile device screen. As such, HTML5 apps are device agnostic and can be opened with any modern mobile browser. Because your content is on the web, it's searchable, which can be a huge benefit for certain types of apps (shopping, for example).

If you're new to mobile development, the technological bar is lower for Web apps; it's easier to get started here than in native or hybrid development. Unfortunately, every mobile device seems to have its own idea of what constitutes usable screen size and resolution. This diversity imposes an additional burden of testing on different devices. Browser incompatibility is especially common on Android devices, for example.

An important part of the "write once, run anywhere" HTML5 methodology is that distribution and support is much easier than for native apps. Need to make a bug fix or add features? Done and deployed for all users. For a native app, there are longer development and testing cycles, after which the consumer typically must log into a store and download a new version to get the latest fix.

If HTML5 apps are easier to develop, easier to support, and can reach the widest range of devices, where do these apps lose out?

- **Secure offline storage** — HTML5 browsers support offline databases and caching, but with no out-of-the-box encryption support. You get all three features in Mobile SDK native applications.
- **Security** — In general, implementing even trivial security measures on a native platform can be complex tasks for a mobile Web developer. It can also be painful for users. For example, a web app with authentication requires users to enter their credentials every time the app restarts or returns from a background state.
- **Native features** — the camera, address book, and other native features are accessible on limited, if any, browser platforms.
- **Native look and feel** — HTML5 can only emulate the native look, while customers won't be able to use familiar compound gestures.

## Hybrid Apps

Hybrid apps are built using HTML5 and JavaScript wrapped inside a thin container that provides access to native platform features. For the most part, hybrid apps provide the best of both worlds, being almost as easy to develop as HTML5 apps with all the functionality of native. In addition, hybrid apps can use the SmartSync Data Framework in JavaScript to model Salesforce data, query and search it, edit it, securely cache it for offline use, and synchronize it with the Salesforce server.

You know that native apps are installed on the device, while HTML5 apps reside on a Web server, so you might be wondering whether hybrid apps store their files on the device or on a server? You can implement a hybrid app locally or remotely.

**Local**

You can package HTML and JavaScript code inside the mobile application binary, in a structure similar to a native application. In this scenario you use REST APIs and Ajax to move data back and forth between the device and the cloud.

**Server**

Alternatively, you can implement the full web application from the server (with optional caching for better performance). Your container app retrieves the full application from the server and displays it in a browser window.

Both types of hybrid development are covered in this guide.

## Native, HTML5, and Hybrid Summary

The following table sums up how the three mobile development scenarios stack up.

| | Native | HTML5 | Hybrid |
|---|---|---|---|
| **Graphics** | Native APIs | HTML, Canvas, SVG | HTML, Canvas, SVG |
| **Performance** | Fastest | Fast | Fast |
| **Look and feel** | Native | Emulated | Emulated |

| | Native | HTML5 | Hybrid |
|---|---|---|---|
| **Distribution** | App store | Web | App store |
| **Camera** | Yes | Browser dependent | Yes |
| **Notifications** | Yes | No | Yes |
| **Contacts, calendar** | Yes | No | Yes |
| **Offline storage** | Secure file system | Shared SQL | Secure file system, shared SQL |
| **Geolocation** | Yes | Yes | Yes |
| **Swipe** | Yes | Yes | Yes |
| **Pinch, spread** | Yes | Yes | Yes |
| **Connectivity** | Online, offline | Mostly online | Online, offline |
| **Development skills** | Objective C, Java | HTML5, CSS, JavaScript | HTML5, CSS, JavaScript |

# Enough Talk; I'm Ready

If you'd rather read about the details later, there are Quick Start topics in this guide for each native development scenario.

- iOS Native Quick Start
- Android Native Quick Start

# Development Prerequisites

It's helpful to have some experience with Database.com or Force.com. You'll need either a Database.com account or a Force.com Developer Edition organization.

This guide also assumes you are familiar with the following technologies and platforms:

- OAuth, login and passcode flows, and Salesforce connected apps. See Authentication, Security, and Identity in Mobile Apps.
- To build iOS applications (hybrid or native), you'll need Mac OS X "Lion" or higher, iOS 6.0 or higher, and Xcode 4.5 or higher.
- To build Android applications (hybrid or native), you'll need the Java JDK 6, Eclipse, Android ADT plugin, and the Android SDK.
- To build remote hybrid applications, you'll need an organization that has Visualforce.
- 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/forcedotcom

## Choosing Between Database.com and Force.com

You can build mobile applications that store data on a Database.com or Force.com organization. Hereafter, this guide assumes you are using a Force.com Developer Edition that uses Force.com login end points such as `login.salesforce.com`. However, you can simply substitute your Database.com credentials in the appropriate places.

**Note:** If you choose to use Database.com, you can't develop Visualforce–driven hybrid apps.

## Sign Up for Force.com

1. In your browser go to `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

1. In your browser go to `www.database.com`.
2. Click **Signup**.
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.

# Mobile SDK Installation

Salesforce Mobile SDK provides two installation paths. The path you choose depends on your development goals.

## Mobile SDK NPM Packages

Most developers, who want to use the SDK as a "black box" and create a mobile app quickly, prefer the Node Packaged Module (NPM) installers. Salesforce provides two packages: **forceios** for the iOS Mobile SDK, and **forcedroid** for the Android version of the Mobile SDK. These packages provide a static snapshot of an SDK release. In the case of iOS, the NPM installer package provides binary modules rather than uncompiled source code. In the case of Android, the NPM installer package provides a snapshot of the SDK source code rather than binaries. You use the NPM package both to install Mobile SDK and to create new template projects.

NPM packages for the Salesforce Mobile SDK reside at https://www.npmjs.org.

**Note:** NPM packages do not support source control, so you can't update your installation dynamically for new releases. Instead, you install each release separately. To upgrade to new versions of the SDK, go to the npmjs.org website and download the new package.

## Mobile SDK GitHub Repository

More adventurous developers who want to delve into the SDK, keep up with the latest changes, and possibly contribute to SDK development can clone the open source repository from GitHub. Using GitHub allows you to monitor source code in public pre-release development branches. In this scenario, both iOS and Android apps include the SDK source code, which is built along with your app.

You don't need to sign up for GitHub to access the Mobile SDK, but we think it's a good idea to be part of this social coding community. https://github.com/forcedotcom

You can always find the latest Mobile SDK releases in our public repositories:

- `https://github.com/forcedotcom/SalesforceMobileSDK-iOS`
- `https://github.com/forcedotcom/SalesforceMobileSDK-Android`

# Keeping Up With the Mobile SDK

The Mobile SDK evolves rapidly, so you'll want to check the following regularly.

- You can always find the most current releases in the NPM registry or our Mobile SDK GitHub Repository
- Keep up to date with What's New.
- The latest articles, blog posts, tutorials, and webinars are on http://www2.developerforce.com/mobile/resources.
- Join the conversation on our message boards at http://boards.developerforce.com/t5/Mobile/bd-p/mobile.

## What's New in This Release

For a summary of what's new and changed in this release of the Salesforce Mobile SDK, visit the Mobile SDK Release Notes. This page also provides a history of previous releases.

# Chapter 2

# Native iOS Development

**In this chapter ...**



Salesforce Mobile SDK delivers libraries and sample Xcode projects for developing mobile apps on iOS.

Two main things that the iOS native SDK provides are:

- Automation of the OAuth2 login process, making it easy to integrate OAuth with your app.
- Access to the REST API with infrastructure classes (including third-party libraries such as RestKit) to make that access as easy as possible.

When you create a native app using the forceios application, your project starts as a fully functioning native sample app. This simple app allows you to connect to a Salesforce organization and run a simple query. It doesn't do much, but it lets you know things are working as designed.

# iOS Native Quick Start

Use the following procedure to get started quickly.

1. Make sure you meet all of the native iOS requirements.
2. Install the Mobile SDK for iOS. If you prefer, you can install the Mobile SDK for iOS from GitHub instead.
3. Run the template app.

# Native iOS Requirements

- Xcode—4.5 is the minimum, but we recommend the latest version.
- iOS 6.0 or higher.
- Mac OS X "Lion" or higher.
- Install the Mobile SDK.
- A Developer Edition organization with a connected app on page 118.

For important information on using various versions of XCode, see the Readme at https://github.com/forcedotcom/SalesforceMobileSDK-iOS/blob/master/readme.md.

# Installing and Uninstalling Salesforce Mobile SDK for iOS

For the fastest, easiest route to iOS development, use NPM to install Salesforce Mobile SDK for iOS.

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.
   a. Download Node.js from www.nodejs.org/download.
   b. Run the downloaded installer to install Node.js and NPM. Accept all prompts asking for permission to install.
3. At a command prompt, 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. Use the forceios package to install the Mobile SDK either globally (recommended) or locally.
   a. To install Salesforce Mobile SDK in a global location, use the `sudo` command and append the "global" option, `-g`:

      ```
      sudo npm install forceios -g
      ```

      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`.
   b. To install Salesforce Mobile SDK in a local folder, `cd` to that folder and use the NPM command without `sudo` or `-g`:

      ```
      npm install forceios
      ```

      This command installs Salesforce Mobile SDK in a `node_modules` folder under your current folder. It links binary modules in `./node_modules/.bin/`. In this scenario, you rarely use `sudo` because you typically install in a local folder where you already have read-write permissions.

### Uninstalling a Forceios Package Installation

Instructions for uninstalling the forceios package vary with whether you installed the package globally or locally. If you installed the package globally, you can run the `uninstall` command from any folder. Be sure to use `sudo` and the `-g` option.

```
$ pwd
/Users/joeuser
$ sudo npm uninstall forceios -g
$
```

To uninstall a package that you installed locally, run the `uninstall` command from the folder where you installed the package. For example:

```
$ pwd
/Users/joeuser
cd <my_projects/my_sdk_folder>
npm uninstall forceios
```

If you try to uninstall a local installation from the wrong directory, you'll get an error message similar to this:

```
npm WARN uninstall not installed in /Users/joeuser/node_modules:
"my_projects/my_sdk_folder/node_modules/forceios"
```

### (Optional) Clone the Salesforce Mobile SDK Source Code from GitHub

If you're adventurous or just curious, you can also install the Salesforce iOS SDK source code from its GitHub repository. Doing so allows you to contribute to the open source and keep up with source code changes.

1. Clone the Mobile SDK iOS repository to your local file system by issuing the following command at the OS X Terminal app: `git clone git://github.com/forcedotcom/SalesforceMobileSDK-iOS.git`

   **Note:** If you have the GitHub app for Mac OS X, click **Clone in Mac**. In your browser, navigate to the Mobile SDK iOS GitHub repository: `https://github.com/forcedotcom/SalesforceMobileSDK-iOS`.

2. In the OS X Terminal app, change to the directory where you installed the cloned repository. By default, this is the SalesforceMobileSDK-iOS directory.
3. Run the install script from the command line: `./install.sh`

# Creating a Native iOS App in Xcode

To create a new app, use forceios again on the command line. You have two options for configuring your app.

- Configure your application options interactively as prompted by the forceios app.
- Specify your application options directly at the command line.

### Specifying Application Options Interactively

To enter application options interactively, do one of the following:

- If you installed Mobile SDK globally, type `forceios create`.
- If you installed Mobile SDK locally, type **<*forceios_path*>**`/node_modules/.bin/forceios create`.

The forceios utility prompts you for each configuration value.

```
rwhitley-ltm1:Downloads rwhitley$ forceios create
Enter your application type (native, hybrid_remote, or hybrid_local): native
Enter your application name: MyNativeiOSApp
Enter your company identifier (com.mycompany): com.acme.goodapps
Enter your organization name (Acme, Inc.): GoodApps, Inc.
Enter the output directory for your app (defaults to the current directory):
Enter your Connected App ID (defaults to the sample app's ID):
Enter your Connected App Callback URI (defaults to the sample app's URI):
Creating app in /Users/rwhitley/Downloads/MyNativeiOSApp
Successfully created native app 'MyNativeiOSApp'.
```

## Specifying Application Options Directly

You can also specify your configuration directly by typing the full forceios command string. To see usage information, type `forceios` without arguments. The list of available options displays:

```
$ forceios
Usage:
forceios create
    --apptype=<Application Type> (native, hybrid_remote, hybrid_local)
    --appname=<Application Name>
    --companyid=<Company Identifier> (com.myCompany.myApp)
    --organization=<Organization Name> (Your company's/organization's name)
    --startpage=<App Start Page> (The start page of your remote app. Only required for
hybrid_remote)
    [--outputdir=<Output directory> (Defaults to the current working directory)]
    [--appid=<Salesforce App Identifier> (The Consumer Key for your app. Defaults to the
sample app.)]
    [--callbackuri=<Salesforce App Callback URL (The Callback URL for your app. Defaults
to the sample app.)]
```

Using this information, type `forceios create`, followed by your options and values. For example:

```
$ forceios create --apptype="native" --appname="package-test"
--companyid="com.acme.mobile_apps" --organization="Acme Widgets, Inc."
--outputdir="PackageTest" --packagename="com.test.my_new_app"
```

## Running the New Project in XCode

Apps created with the forceios template are ready to run “right out of the box”. After the app creation script finishes, you can open and run the project in Xcode.

1. In Xcode, select **File** > **Open**.
2. Navigate to the output folder you specified.
3. Open your app’s `xcodeproj` file.
4. Click the **Run** button in the upper left corner to see your new app in action.

.

## forceios Command Parameters

These are the descriptions of the forceios command parameters:

| Parameter Name | Description |
|---|---|
| `--apptype` | One of the following:<br>• “native”<br>• “hybrid_remote” (server-side hybrid app using VisualForce)<br>• “hybrid_local” (client-side hybrid app that doesn’t use VisualForce) |

| Parameter Name | Description |
|---|---|
| `--appname` | Name of your application |
| `--companyid` | A unique identifier for your company. This value is concatenated with the app name to create a unique app identifier for publishing your app to the App Store. For example, “com.myCompany.apps”. |
| `--organization` | The formal name of your company. For example, “Acme Widgets, Inc.”. |
| `--packagename` | Package identifier for your application. For example, “com.acme.app”. |
| `--startpage` | (hybrid remote apps only) Server path to the remote start page. For example: `/apex/MyAppStartPage`. |
| `--outputdir` | (Optional) Folder in which you want your project to be created. If the folder doesn’t exist, the script creates it. Defaults to the current working directory. |
| `--appid` | (Optional) Your connected app’s Consumer Key. Defaults to the consumer key of the sample app.<br>  **Note:** If you don’t specify the value here, you’re required to change it in the app before you publish to the App Store. |
| `--callbackuri` | (Optional) Your connected app’s Callback URL. Defaults to the callback URL of the sample app.<br>  **Note:** If you don’t specify the value here, you’re required to change it in the app before you publish to the App Store. |
| `--usesmartstore=true` | (Optional) Include only if you want to use SmartStore for offline data. Defaults to false if not specified. |

## Running the Xcode Project Template App

The Xcode project template includes a sample application you can run right away.

1. Press **Command-R** and the default template app runs in the iOS simulator.
2. On startup, the application starts the OAuth authentication flow, which results in an authentication page. Enter your credentials, and click **Login**.
3. Tap **Allow** when asked for permission

You should now be able to compile and run the sample project. It's a simple app that logs you into an org via OAuth2, issues a `select Name from Account` SOQL query, and displays the result in a `UITableView` instance.

# Developing a Native iOS App

The Salesforce Mobile SDK for native iOS provides the tools you need to build apps for Apple mobile devices. Features of the SDK include:

- Classes and interfaces that make it easy to call the Salesforce REST API
- Fully implemented OAuth login and passcode protocols
- SmartStore library for securely managing user data offline

The native iOS SDK requires you to be proficient in Objective-C coding. You also need to be familiar with iOS application development principles and frameworks. If you're a newbie, Start Developing iOS Apps Today is a good place to begin learning. See Native iOS Requirements on page 9 for additional prerequisites.

In a few Mobile SDK interfaces, you're required to override some methods and properties. SDK header (.h) files include comments that indicate mandatory and optional overrides.

## About Login and Passcodes

To access Salesforce objects from a Mobile SDK app, the user logs into an organization on a Salesforce server. When the login flow begins, your app sends its connected app configuration to Salesforce. Salesforce responds by posting a login screen to the mobile device.

Optionally, a Salesforce administrator can set the connected app to require a passcode after login. The Mobile SDK handles presentation of the login and passcode screens, as well as authentication handshakes. Your app doesn't have to do anything to display these screens. However, you do need to understand the login flow and how OAuth tokens are handled. See About PIN Security on page 118 and OAuth2 Authentication Flow on page 119.

## About Memory Management

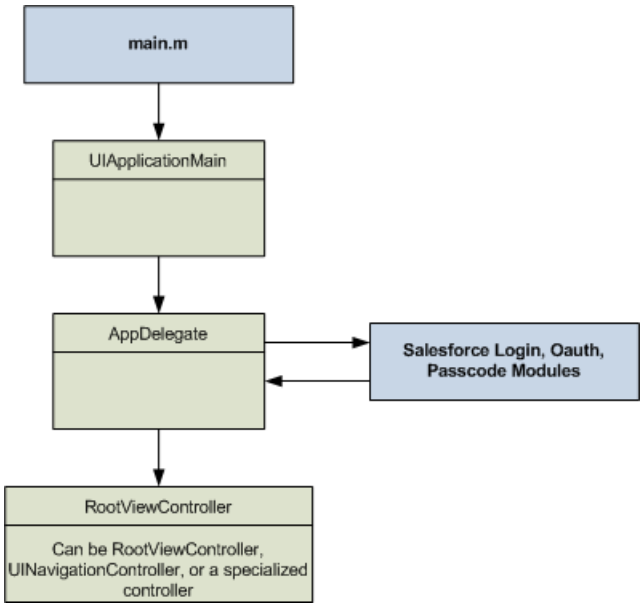
Beginning in Mobile SDK 2.0, native iOS apps use Automatic Reference Counting (ARC) to manage object memory. You don't have to allocate and then remember to deallocate your objects. See the Mac Developer Library at https://developer.apple.com for ARC syntax, guidelines, and best practices.

## Overview of Application Flow

When you create a project with the forceios application, your new app defines three classes: `AppDelegate`, `InitialViewController`, and `RootViewController`. The `AppDelegate` object loads `InitialViewController` as the first view to show. After the authentication process completes, the `AppDelegate` object displays the view associated with `RootViewController` as the entry point to your app.

The workflow demonstrated by the template app is merely an example. You can tailor your `AppDelegate` and supporting classes to achieve your desired workflow. You can retrieve data through REST API calls and display it, launch other views, perform services, and so on. Your app remains alive in memory until the user quits it, or until the device is rebooted.

Native iOS apps built with the Mobile SDK follow the same design as other iOS apps. The `main.m` source file creates a `UIApplicationMain` object that is the root object for the rest of the application. The `UIApplicationMain` constructor creates an `AppDelegate` object that manages the application lifecycle.




# AppDelegate Class

The `AppDelegate` class is the true entry point for an iOS app. In Mobile SDK apps, `AppDelegate` implements the standard iOS `UIApplicationDelegate` interface. The Mobile SDK template application for creating native iOS apps implements most of the Salesforce-specific startup functionality for you.

To customize the `AppDelegate` template, populate the following static variables with information from your Force.com Connected Application:

- `RemoteAccessConsumerKey`

```
static NSString * const RemoteAccessConsumerKey =
@"3MVG9Iu66FKeHhINkB1l7xt7kR8czFcCTUhgoA8Ol2Ltf1eYHOU4SqQRSEitYFDUpqRWcoQ2.dBv_a1Dyu5xa";
```

- `OAuthRedirectURI`

```
static NSString * const OAuthRedirectURI = @"testsfdc:///mobilesdk/detect/oauth/done";
```

OAuth functionality resides in an independent module. This separation makes it possible for you to use Salesforce authentication on demand. You can start the login process from within your `AppDelegate` implementation, or you can postpone login until it's actually required—for example, you can call OAuth from a sub-view.

## Initialization

The following high-level overview shows how the `AppDelegate` initializes the template app. Keep in mind that you can change any of these details to suit your needs.

1. When the `[AppDelegate init]` message runs, it:
   - Initializes configuration items, such as Connected App identifiers, OAuth scopes, and so on.
   - Adds notification observers that listen to `SFAuthenticationManager`, `logoutInitiated`, and `loginHostChanged` notifications.

     The `logoutInitiated` notification lets the app respond when a user logs out voluntarily or is logged out involuntarily due to invalid credentials. The `loginHostChanged` notification lets the app respond when the user changes the login

host (for example, from Production to Sandbox). See the `logoutInitiated:` and `loginHostChanged:` handler methods in the sample app.

- Initializes authentication "success" and "failure" blocks for the `[SFAuthenticationManager loginWithCompletion:failure:]` message. These blocks determine what happens when the authentication process completes.

2. `application:didFinishLaunchingWithOptions:`, a `UIApplicationDelegate` method, is called at app startup. The template app uses this method to initialize the `UIWindow` property, display the initial view (see `initializeAppViewState`), and initiate authentication. If authentication succeeds, the `SFAuthenticationManager` executes `initialLoginSuccessBlock` (the "success" block).
3. `initialLoginSuccessBlock` calls `setupRootViewController`, which creates and displays the app's `RootViewController`.

You can customize any part of this process. At a minimum, change `setupRootViewController` to display your own controller after authentication. You can also customize `initializeAppViewState` to display your own launch page, or the `InitialViewController` to suit your needs. You can also move the authentication details to where they make the most sense for your app. The Mobile SDK does not stipulate when—or if—actions must occur, but standard iOS conventions apply. For example, `self.window` must have a `rootViewController` by the time `application:didFinishLaunchingWithOptions:` completes.

### UIApplication Event Handlers

You can also use the application delegate class to implement `UIApplication` event handlers. Important event handlers that you might consider implementing or customizing include:

**`application:didFinishLaunchingWithOptions:`**

First entry point when your app launches. Called only when the process first starts (not after a backgrounding/foregrounding cycle).

**`applicationDidBecomeActive`**

Called every time the application is foregrounded. The iOS SDK provides no default parent behavior; if you use it, you must implement it from the ground up.

For a list of all `UIApplication` event handlers, see "UIApplicationDelegate Protocol Reference" in the iOS Developer Library.

## About View Controllers

In addition to the views and view controllers discussed with the `AppDelegate` class, Mobile SDK exposes the `SFAuthorizingViewController` class. This controller displays the login screen when necessary.

To customize the login screen display:

1. Override the `SFAuthorizingViewController` class to implement your custom display logic.
2. Set the `[SFAuthenticationManager sharedManager].authViewController` property to an instance of your customized class.

The most important view controller in your app is the one that manages the first view that displays, after login or—if login is postponed—after launch. This controller is called your root view controller because it controls everything else that happens in your app. The Mobile SDK for iOS project template provides a skeletal class named `RootViewController` that demonstrates the minimal required implementation.

If your app needs additional view controllers, you're free to create them as you wish. The view controllers used in Mobile SDK projects reveal some possible options. For example, the Mobile SDK iOS template project bases its root view class on the

`UITableViewController` interface, while the `RestAPIExplorer` sample project uses the `UIViewController` interface. Your only technical limits are those imposed by iOS itself and the Objective-C language.

# RootViewController Class

The `RootViewController` class exists only as part of the template project and projects generated from it. It implements the `SFRestDelegate` protocol to set up a framework for your app's interactions with the Salesforce REST API. Regardless of how you define your root view controller, it must implement `SFRestDelegate` if you intend to use it to access Salesforce data through the REST APIs.

## RootViewController Design

As an element of a very basic app built with the Mobile SDK, the `RootViewController` class covers only the bare essentials. Its two primary tasks are:

- Use Salesforce REST APIs to query Salesforce data
- Display the Salesforce data in a table

To do these things, the class inherits `UITableViewController` and implements the `SFRestDelegate` protocol. The action begins with an override of the `UIViewController:viewDidLoad` method:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.title = @"Mobile SDK Sample App";

    //Here we use a query that should work on either Force.com or Database.com
    SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name FROM
 User LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
}
```

The iOS runtime calls `viewDidLoad` only once in the view's life cycle, when the view is first loaded into memory. The intention in this skeletal app is to load only one set of data into the app's only defined view. If you plan to create other views, you might need to perform the query somewhere else. For example, if you add a detail view that lets the user edit data shown in the root view, you'll want to refresh the values shown in the root view when it reappears. In this case, you can perform the query in a more appropriate method, such as `viewWillAppear`.

After calling the superclass method, this code sets the title of the view, then issues a REST request in the form of an asynchronous SOQL query. The query in this case is a simple SELECT statement that gets the Name property from each User object and limits the number of rows returned to ten. Notice that the `requestForQuery` and `send:delegate:` messages are sent to a singleton shared instance of the `SFRestAPI` class. Use this singleton object for all REST requests. This object uses authenticated credentials from the singleton `SFAccountManager` object to form and send authenticated requests.

The Salesforce REST API responds by passing status messages and, hopefully, data to the delegate listed in the send message. In this case, the delegate is the `RootViewController` object itself:

```
[[SFRestAPI sharedInstance] send:request delegate:self];
```

The `RootViewController` object can act as an `SFRestAPI` delegate because it implements the `SFRestDelegate` protocol. This protocol declares four possible response callbacks:

- `request:didLoadResponse:` — Your request was processed. The delegate receives the response in JSON format. This is the only callback that indicates success.
- `request:didFailLoadWithError:` — Your request couldn't be processed. The delegate receives an error message.
- `requestDidCancelLoad` — Your request was canceled by some external factor, such as administrator intervention, a network glitch, or another unexpected event. The delegate receives no return value.
- `requestDidTimeout` — The Salesforce server failed to respond in time. The delegate receives no return value.

The response arrives in one of the callbacks you've implemented in `RootViewController`. Place your code for handling Salesforce data in the `request:didLoadResponse:` callback. For example:

```
- (void)request:(SFRestRequest *)request
      didLoadResponse:(id)jsonResponse {
    NSArray *records = [jsonResponse objectForKey:@"records"];
    NSLog(@"request:didLoadResponse: #records: %d", records.count);
    self.dataRows = records;
    [self.tableView reloadData];
}
```

As the use of the `id` data type suggests, this code handles JSON responses in generic Objective-C terms. It addresses the `jsonResponse` object as an instance of `NSDictionary` and treats its records as an `NSArray` object. Because `RootViewController` implements `UITableViewController`, it's simple to populate the table in the view with extracted records.

A call to `request:didFailLoadWithError:` results from one of the following conditions:

- If you use invalid request parameters, you get a `kSFRestErrorDomain` error code. For example, if you pass nil to `requestForQuery:`, or you try to update a non-existent object.
- If an OAuth access token expires, the framework tries to obtain a new access token and, if successful, retries the query. If a request for a new access token or session ID fails, you get a `kSFOAuthErrorDomain` error code. For example, if the access token expires, and the OAuth refresh token is invalid. This scenario rarely occurs.
- If the low-level HTTP request fails, you get an `RKRestKitErrorDomain` error code. For example, if a Salesforce server becomes temporarily inaccessible.

The other callbacks are self-describing, and don't return an error code. You can choose to handle the result however you want: display an error message, write to the log, retry the request, and so on.

# About Salesforce REST APIs

Native app development with the Salesforce Mobile SDK centers around the use of Salesforce REST APIs. Salesforce makes a wide range of object-based tasks available through URIs with REST parameters. Mobile SDK wraps these HTTP calls in interfaces that handle most of the low-level work in formatting a request.

In Mobile SDK for iOS, all REST requests are performed asynchronously. You can choose between delegate and block versions of the REST wrapper classes to adapt your requests to various scenarios. REST responses are formatted as `NSArray` or `NSDictionary` objects for a successful request, or `NSError` if the request fails.

See the Force.com REST API Developer's Guide for information on Salesforce REST response formats.

## Supported Operations

The iOS REST APIs support the standard object operations offered by Salesforce REST and SOAP APIs. Salesforce Mobile SDK offers delegate and block versions of its REST request APIs. Delegate request methods are defined in the `SFRestAPI` class, while block request methods are defined in the `SFRestAPI (Blocks)` category. Supported operations are:

| Operation | Delegate method | Block method |
| --- | --- | --- |
| **Manual REST request**<br>Executes a request that you've built | `send:delegate:` | `sendRESTRequest:failBlock:completeBlock:` |
| **SOQL query**<br>Executes the given SOQL string and returns the resulting data set | `requestForQuery:` | `performSOQLQuery:failBlock:completeBlock:` |
| **SOSL search**<br>Executes the given SOSL string and returns the resulting data set | `requestForSearch:` | `performSOSLSearch:failBlock:completeBlock:` |
| **Metadata** | `requestForMetadataWithObjectType:` | `performMetadataWithObjectType:failBlock:completeBlock:` |

| Operation | Delegate method | Block method |
|---|---|---|
| Returns the object's metadata | | |
| **Describe global**<br>Returns a list of all available objects in your org and their metadata | `requestForDescribeGlobal` | `performDescribeGlobalWithFailBlock:completeBlock:` |
| **Describe with object type**<br>Returns a description of a single object type | `requestForDescribeWithObjectType:` | `performDescribeWithObjectType:failBlock:completeBlock:` |
| **Retrieve**<br>Retrieves a single record by object ID | `requestForRetrieveWithObjectType:objectId:fieldList:` | `performRetrieveWithObjectType:objectId:fieldList:failBlock:completeBlock:` |
| **Update**<br>Updates an object with the given map | `requestForUpdateWithObjectType:objectId:fields:` | `performUpdateWithObjectType:objectId:fields:failBlock:completeBlock:` |
| **Upsert**<br>Updates or inserts an object from external data, based on whether the external ID currently exists in the external ID field | `requestForUpsertWithObjectType:externalIdField:externalId::fields:` | `performUpsertWithObjectType:externalIdField:externalId:fields:failBlock:completeBlock:` |
| **Create**<br>Creates a new record in the specified object | `requestForCreateWithObjectType:fields:` | `performCreateWithObjectType:fields:failBlock:completeBlock:` |
| **Delete**<br>Deletes the object of the given type with the given ID | `requestForDeleteWithObjectType:objectId:` | `performDeleteWithObjectType:objectId:failBlock:completeBlock:` |
| **Versions**<br>Returns Salesforce version metadata | `requestForVersions` | `performRequestForVersionsWithFailBlock:completeBlock:` |
| **Resources** | `requestForResources` | `performRequestForResourcesWithFailBlock:` |

| Operation | Delegate method | Block method |
|---|---|---|
| Returns available resources for the specified API version, including resource name and URI | | `completeBlock:` |

## SFRestAPI Interface

`SFRestAPI` defines the native interface for creating and formatting Salesforce REST requests. It works by formatting and sending your requests to the Salesforce service, then relaying asynchronous responses to your implementation of the `SFRestDelegate` protocol.

`SFRestAPI` serves as a factory for `SFRestRequest` instances. It defines a group of methods that represent the request types supported by the Salesforce REST API. Each `SFRestAPI` method corresponds to a single request type. Each of these methods returns your request in the form of an `SFRestRequest` instance. You then use that return value to send your request to the Salesforce server. The HTTP coding layer is encapsulated, so you don't have to worry about REST API syntax.

For a list of supported query factory methods, see Supported Operations on page 17

## SFRestDelegate Protocol

When a class adopts the `SFRestDelegate` protocol, it intends to be a target for REST responses sent from the Salesforce server. When you send a REST request to the server, you tell the shared `SFRestAPI` instance which object receives the response. When the server sends the response, Mobile SDK routes the response to the appropriate protocol method on the given object.

The `SFRestDelegate` protocol declares four possible responses:

- `request:didLoadResponse:` — Your request was processed. The delegate receives the response in JSON format. This is the only callback that indicates success.
- `request:didFailLoadWithError:` — Your request couldn't be processed. The delegate receives an error message.
- `requestDidCancelLoad` — Your request was canceled by some external factor, such as administrator intervention, a network glitch, or another unexpected event. The delegate receives no return value.
- `requestDidTimeout` — The Salesforce server failed to respond in time. The delegate receives no return value.

The response arrives in your implementation of one of these delegate methods. Because you don't know which type of response to expect, you must implement all of the methods.

### request:didLoadResponse: Method

The `request:didLoadResponse:` method is the only protocol method that handles a success condition, so place your code for handling Salesforce data in that method. For example:

```
- (void)request:(SFRestRequest *)request
      didLoadResponse:(id)jsonResponse {
    NSArray *records = [jsonResponse objectForKey:@"records"];
    NSLog(@"request:didLoadResponse: #records: %d", records.count);
    self.dataRows = records;
    [self.tableView reloadData];
}
```

At the server, all responses originate as JSON strings. Mobile SDK receives these raw responses and reformats them as iOS SDK objects before passing them to the `request:didLoadResponse:` method. Thus, the `jsonResponse` payload arrives as either an `NSDictionary` object or an `NSArray` object. The object type depends on the type of JSON data returned. If the top level of the server response represents a JSON object, `jsonResponse` is an `NSDictionary` object. If the top level represents a JSON array of other data, `jsonResponse` is an `NSArray` object.

If your method cannot infer the data type from the request, use `[NSObject isKindOfClass:]` to determine the data type. For example:

```
if ([jsonResponse isKindOfClass:[NSArray class]]) {
     // Handle an NSArray here.
} else {
     // Handle an NSDictionary here.
}
```

You can address the response as an `NSDictionary` object and extract its records into an `NSArray` object. To do so, send the `NSDictionary:objectForKey:` message using the key "records".

#### request:didFailLoadWithError: Method

A call to the `request:didFailLoadWithError:` callback results from one of the following conditions:

- If you use invalid request parameters, you get a `kSFRestErrorDomain` error code. For example, you pass nil to `requestForQuery:`, or you try to update a non-existent object.
- If an OAuth access token expires, the framework tries to obtain a new access token and, if successful, retries the query. If a request for a new access token or session ID fails, you get a `kSFOAuthErrorDomain` error code. For example, the access token expires, and the OAuth refresh token is invalid. This scenario rarely occurs.
- If the low-level HTTP request fails, you get an `RKRestKitErrorDomain` error code. For example, a Salesforce server becomes temporarily inaccessible.

#### requestDidCancelLoad and requestDidTimeout Methods

The `requestDidCancelLoad` and `requestDidTimeout` delegate methods are self-describing and don't return an error code. You can choose to handle the result however you want: display an error message, write to the log, retry the request, and so on.

## Creating REST Requests

Salesforce Mobile SDK for iOS natively supports many types of SOQL and SOSL REST requests. The `SFRestAPI` class provides factory methods that handle most of the syntactical details for you. Mobile SDK also offers considerable flexibility for how you create REST requests.

- For standard SOQL queries and SOSL searches, `SFRestAPI` methods create query strings based on minimal data input and package them in an `SFRestRequest` object that can be sent to the Salesforce server.
- If you are using a Salesforce REST API that isn't based on SOQL or SOSL, `SFRestRequest` methods let you configure the request itself to match the API format.
- The `SFRestAPI (QueryBuilder)` category provides methods that create free-form SOQL queries and SOSL search strings so you don't have to manually format the query or search string.
- Request methods in the `SFRestAPI (Blocks)` category let you pass callback code as block methods, instead of using a delegate object.

## Sending a REST Request

Salesforce Mobile SDK for iOS natively supports many types of SOQL and SOSL REST requests. Luckily, the `SFRestAPI` provides factory methods that handle most of the syntactical details for you.

At runtime, Mobile SDK creates a singleton instance of `SFRestAPI`. You use this instance to obtain an `SFRestRequest` object and to send that object to the Salesforce server.

To send a REST request to the Salesforce server from an `SFRestAPI` delegate:

1. Build a SOQL, SOSL, or other REST request string.

   For standard SOQL and SOSL queries, it's most convenient and reliable to use the factory methods in the `SFRestAPI` class. See Supported Operations.

2. Create an `SFRestRequest` object with your request string.

   Message the `SFRestAPI` singleton with the request factory method that suits your needs. For example, this code uses the`SFRestAPI:requestForQuery:` method, which prepares a SOQL query.

   ```
   // Send a request factory message to the singleton SFRestAPI instance
   SFRestRequest *request = [[SFRestAPI sharedInstance]
       requestForQuery:@"SELECT Name FROM User LIMIT 10"];
   ```

3. Send the `send:delegate:` message to the shared `SFRestAPI` instance. Use your new `SFRestRequest` object as the `send:` parameter. The second parameter designates an `SFRestDelegate` object to receive the server's response. In the following example, the class itself implements the `SFRestDelegate` protocol, so it sets `delegate:` to `self`.

   ```
   // Use the singleton SFRestAPI instance to send the
   // request, specifying this class as the delegate.
   [[SFRestAPI sharedInstance] send:request delegate:self];
   ```

## SFRestRequest Class

Salesforce Mobile SDK provides the `SFRestRequest` interface as a convenience class for apps. `SFRestAPI` provides request methods that use your input to form a request. This request is packaged as an `SFRestRequest` instance and returned to your app. In most cases you don't manipulate the `SFRestRequest` object. Typically, you simply pass it unchanged to the `SFRestAPI:send:delegate:` method.

If you're sending a REST request that isn't directly supported by the Mobile SDK—for example, if you want to use the Chatter REST API—you can manually create and configure an `SFRestRequest` object.

## Using SFRestRequest Methods

`SFRestAPI` tools support SOQL and SOSL statements natively: they understand the grammar and can format valid requests based on minimal input from your app. However, Salesforce provides some product-specific REST APIs that have no relationship to SOQL queries or SOSL searches. You can still use Mobile SDK resources to configure and send these requests. This process is similar to sending a SOQL query request. The main difference is that you create and populate your `SFRestRequest` object directly, instead of relying on `SFRestAPI` methods.

To send a non-SOQL and non-SOSL REST request using the Mobile SDK:

1. Create an instance of `SFRestRequest`.
2. Set the properties you need on the `SFRestRequest` object.
3. Call `send:delegate:` on the singleton `SFRestAPI` instance, passing in the `SFRestRequest` object you created as the first parameter.

   The following example performs a GET operation to obtain all items in a specific Chatter feed.

   ```
   SFRestRequest *request = [[SFRestRequest alloc] init];
   [request setDelegate:self];
   ```

```
[request setEndpoint:kSFDefaultRestEndpoint];
[request setMethod:SFRestMethodGET];
[request setPath:[NSString stringWithFormat:@"/v26.0/chatter/feeds/record/%@/feed-items",

    recordId]];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

4. Alternatively, you can create the same request using the `requestWithMethod:path:queryParams` class method.

```
SFRestRequest *request =
    [SFRestRequest requestWithMethod:SFRestMethodGET
                                path:[NSString stringWithFormat:
                                         @"/v26.0/chatter/feeds/record/%@/feed-items",
                                         recordId]
                         queryParams:nil];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

5. To perform a request with parameters, create a parameter string, and then use the `SFJsonUtils:objectFromJSONString` static method to wrap it in an `NSDictionary` object. (If you prefer, you can create your `NSDictionary` object directly, before the method call, instead of creating it inline.)

   The following example performs a POST operation that adds a comment to a Chatter feed.

```
NSString *body = [NSString stringWithFormat:@"{\"body\" : {\"messageSegments\" :
                                             [{ \"type\" : \"Text\", \"text\" : \"%@\"}
 ] } }",
                                              comment];

SFRestRequest *request =
    [SFRestRequest requestWithMethod:SFRestMethodPOST
                path:[NSString stringWithFormat:
                       @"/v26.0/chatter/feeds/record/%@/feed-items",
                       recordId]
        queryParams:(NSDictionary *)[SFJsonUtils objectFromJSONString:body]];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

## SFRestAPI (Blocks) Category

If you prefer, you can use blocks instead of a delegate to execute callback code. Salesforce Mobile SDK for native iOS provides a block corollary for each `SFRestAPI` request method. These methods are defined in the `SFRestAPI (Blocks)` category.

Block request methods look a lot like delegate request methods. They all return a pointer to `SFRestRequest`, and they require the same parameters. Block request methods differ from their delegate siblings in these ways:

1. In addition to copying the REST API parameters, each method requires two blocks: a fail block of type `SFRestFailBlock`, and a complete block of type `SFRestDictionaryResponseBlock` or type `SFRestArrayResponseBlock`, depending on the expected response data.
2. Block-based methods send your request for you, so you don't need to call a separate send method. If your request fails, you can use the `SFRestRequest *` return value to retry the request. To do this, use the `SFRestAPI:sendRESTRequest:failBlock:completeBlock:` method.

Judicious use of blocks and delegates can help fine-tune your app's readability and ease of maintenance. Prime conditions for using blocks often correspond to those that mandate inline functions in C++ or anonymous functions in Java. However, this observation is just a general suggestion. Ultimately, you need to make a judgement call based on research into your app's real-world behavior.

## SFRestAPI (QueryBuilder) Category

If you're unsure of the correct syntax for a SOQL query or a SOSL search, you can get help from the `SFRestAPI (QueryBuilder)` category methods. These methods build query strings from basic conditions that you specify, and return the formatted string. You can pass the returned value to one of the following `SFRestAPI` methods.

- `- (SFRestRequest *)requestForQuery:(NSString *)soql;`
- `- (SFRestRequest *)requestForSearch:(NSString *)sosl;`

`SFRestAPI (QueryBuilder)` provides two static methods each for SOQL queries and SOSL searches: one takes minimal parameters, while the other accepts a full list of options.

### SOSL Methods

SOSL query builder methods are:

```
+ (NSString *) SOSLSearchWithSearchTerm:(NSString *)term
                            objectScope:(NSDictionary *)objectScope;

+ (NSString *) SOSLSearchWithSearchTerm:(NSString *)term
                             fieldScope:(NSString *)fieldScope
                            objectScope:(NSDictionary *)objectScope
                                  limit:(NSInteger)limit;
```

Parameters for the SOSL search methods are:

- `term` is the search string. This string can be any arbitrary value. The method escapes any SOSL reserved characters before processing the search.
- `fieldScope` indicates which fields to search. It's either `nil` or one of the IN search group expressions: "IN ALL FIELDS", "IN EMAIL FIELDS", "IN NAME FIELDS", "IN PHONE FIELDS", or "IN SIDEBAR FIELDS". A `nil` value defaults to "IN NAME FIELDS". See Salesforce Object Search Language (SOSL).
- `objectScope` specifies the objects to search. Acceptable values are:
  - `nil`—No scope restrictions. Searches all searchable objects.
  - An `NSDictionary` object pointer—Corresponds to the SOSL RETURNING fieldspec. Each key is an `sObject` name; each value is a string that contains a field list as well as optional WHERE, ORDER BY, and LIMIT clauses for the key object.

    If you use an `NSDictionary` object, each value must contain at least a field list. For example, to represent the following SOSL statement in a dictionary entry:

    ```
    FIND {Widget Smith}
    IN Name Fields
    RETURNING Widget__c (name Where createddate = THIS_FISCAL_QUARTER)
    ```

    set the key to "Widget__c" and its value to "name WHERE `createddate` = "THIS_FISCAL_QUARTER". For example:

    ```
    [SFRestAPI
        SOSLSearchWithSearchTerm:@"all of these will be escaped:~{]"
                     objectScope:[NSDictionary
                                  dictionaryWithObject:@"name WHERE
                                                         createddate="THIS_FISCAL_QUARTER"

                                               forKey:@"Widget__c"]];
    ```

  - `NSNull`—No scope specified.

- `limit`—If you want to limit the number of results returned, set this parameter to the maximum number of results you want to receive.

### SOQL Methods

SOQL QueryBuilder methods that construct SOQL strings are:

```
+ (NSString *) SOQLQueryWithFields:(NSArray *)fields
                           sObject:(NSString *)sObject
                             where:(NSString *)where
                             limit:(NSInteger)limit;

+ (NSString *) SOQLQueryWithFields:(NSArray *)fields
                           sObject:(NSString *)sObject
                             where:(NSString *)where
                           groupBy:(NSArray *)groupBy
                            having:(NSString *)having
                           orderBy:(NSArray *)orderBy
                             limit:(NSInteger)limit;
```

Parameters for the SOQL methods correspond to SOQL query syntax. All parameters except `fields` and `sObject` can be set to `nil`.

| Parameter name | Description |
|---|---|
| `fields` | An array of field names to be queried. |
| `sObject` | Name of the object to query. |
| `where` | An expression specifying one or more query conditions. |
| `groupBy` | An array of field names to use for grouping the resulting records. |
| `having` | An expression, usually using an aggregate function, for filtering the grouped results. Used only with `groupBy`. |
| `orderBy` | An array of fields name to use for ordering the resulting records. |
| `limit` | Maximum number of records you want returned. |

See SOQL SELECT Syntax.

### SOSL Sanitizing

The `QueryBuilder` category also provides a class method for cleaning SOSL search terms:

```
+ (NSString *) sanitizeSOSLSearchTerm:(NSString *)searchTerm;
```

This method escapes every SOSL reserved character in the input string, and returns the escaped version. For example:

```
NSString *soslClean = [SFRestAPI sanitizeSOSLSearchTerm:@"FIND {MyProspect}"];
```

This call returns “FIND \{MyProspect\}”.

The `sanitizeSOSLSearchTerm:` method is called in the implementation of the SOSL and SOQL QueryBuilder methods, so you don’t need to call it on strings that you’re passing to those methods. However, you can use it if, for instance, you’re building your own queries manually. SOSL reserved characters include:

```
\ ? & | ! { } [ ] ( ) ^ ~ * : " ' + -
```

# iOS Sample Applications

The app you created in Running the Xcode Project Template App is itself a sample application, but it only does one thing: issue a SOQL query and return a result. The native iOS sample apps have a lot more functionality you can examine and work into your own apps.

- The RestAPIExplorer sample app exercises all of the native REST API wrappers. It is in the Mobile SDK for iOS under `native/SampleApps/RestAPIExplorer`.
- The NativeSqlAggregator sample app shows SQL aggregation examples as well as a native SmartStore implementation. It resides in the Mobile SDK for iOS under `native/SampleApps/NativeSqlAggregator`.

# Chapter 3

# Native Android Development

**In this chapter ...**



Salesforce Mobile SDK delivers libraries and sample projects for developing native mobile apps on Android.

The Android native SDK provides two main features:

- Automation of the OAuth2 login process, making it easy to integrate the process with your app.
- Access to the Salesforce REST API, with utility classes that simplify that access.

The Android Salesforce Mobile SDK includes several sample native applications. It also provides an `ant` target for quickly creating a new application.

# Android Native Quick Start

Use the following procedure to get started quickly.

1. Make sure you meet all of the native Android requirements.
2. Install the Mobile SDK for Android.
3. At the command line, run an `ant` script to create a new Android project , and then run that app in Eclipse or from the command line.
4. Set up sample projects in Eclipse.

# Native Android Requirements

- 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 remote access application.

The `SalesforceSDK` project is built with the Android 3.0 (Honeycomb) library. The primary reason for this is that we want to be able to make a conditional check at runtime for file system encryption capabilities. This check is bypassed on earlier Android platforms; thus, you can still use the `salesforcesdk.jar` in earlier Android application versions, down to the mininum-supported Android 2.2.

# Installing and Uninstalling Salesforce Mobile SDK for Android

For the fastest, easiest route to Android development, use NPM to install Salesforce Mobile SDK for Android.

1. If you've already successfully installed Node.js and npm, skip to Step 4.
2. Install Node.js and npm on your system.
   a. a. Download Node.js from www.nodejs.org/download.
   b. b. Run the downloaded installer to install Node.js and npm. Accept all prompts asking for permission to install.
3. At a command prompt, 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. Use the forcedroid package to install the Mobile SDK either globally (recommended) or locally.

**a.** To install Salesforce Mobile SDK in a global location, append the "global" option, `-g`, to the end of the command. For non-Windows environments, use the `sudo` command:

```
sudo npm install forcedroid -g
```

On Windows:

```
npm install forcedroid -g
```

With the `-g` option, you run `npm install` from any directory. In non-Windows environments, 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`. In Windows environments, global packages are installed in `%APPDATA%\npm\node_modules`, and binaries are linked in `%APPDATA%\npm`.

**b.** To install Salesforce Mobile SDK in a local directory, `cd` to that directory and use the NPM command without `sudo` or the `-g` option:

```
npm install forcedroid
```

This command installs Salesforce Mobile SDK in a `node_modules` directory under your current directory. It links binary modules in `./node_modules/.bin/`. In this scenario, you rarely use `sudo` because you typically install in a local folder where you already have read-write permissions.

## Uninstalling the Forcedroid Package

The instructions for uninstalling the forcedroid package vary with whether you installed the package globally or locally.

If you installed the package globally, you can run the `uninstall` command from any folder. Be sure to use the `-g` option. On a Unix-based platform such as Mac OS X, use `sudo` as well.

```
$ pwd
/Users/joeuser
$ sudo npm uninstall forcedroid -g
$
```

If you installed the package locally, run the `uninstall` command from the folder where you installed the package. For example:

```
cd <my_projects/my_sdk_folder>
npm uninstall forcedroid
```

If you try to uninstall a local installation from the wrong directory, you'll get an error message similar to this:

```
npm WARN uninstall not installed in /Users/joeuser/node_modules:
"my_projects/my_sdk_folder/node_modules/forcedroid"
```

## (Optional) Clone the Salesforce Mobile SDK Source Code from GitHub

If you're adventurous or just curious, you can choose to install the Salesforce Mobile SDK source code from its GitHub repository. Doing so allows you to contribute to the open source and keep up with source code changes.

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

3. Open a command prompt in the directory where you installed the cloned repository, and run the install script from the command line: `./install.sh`

**Note: Windows users:** Run `cscript install.vbs`.

Create shell variables:

1. `ANDROID_SDK_DIR` pointing to the Android SDK directory
2. `SALESFORCE_SDK_DIR` pointing to your clone of the Salesforce Mobile SDK repository, for example: `/home/jon/SalesforceMobileSDK-Android`
3. `NATIVE_DIR` pointing to `$SALESFORCE_SDK_DIR/native`
4. `TARGET_DIR` pointing to a location you've defined to contain your Android project

**Note:** These variables are for your own convenience. If you don't set up these variables, make sure to replace `$ANDROID_SDK_DIR, $SALESFORCE_SDK_DIR, $NATIVE_DIR` and `$TARGET_DIR` in the various code snippets in this guide with the actual paths.

# Creating a New Android Project

To create a new app, use forcedroid again on the command line. You have two options for configuring your app.

- Configure your application options interactively as prompted by the forcedroid app.
- Specify your application options directly at the command line.

## Specifying Application Options Interactively

To enter application options interactively, do one of the following:

- If you installed Mobile SDK globally, type `forcedroid create`.
- If you installed Mobile SDK locally, type <***forcedroid_path***>`/node_modules/.bin/forcedroid create`.

The forcedroid utility prompts you for each configuration option.

```
rwhitley-ltm1:Downloads rwhitley$ forcedroid create
Enter your application type (native, hybrid_remote, or hybrid_local): native
Enter your application name: MyNativeAndroidApp
Enter the target directory of your app: /Users/rwhitley/Development/AndroidApps
Enter the package name for your app (com.mycompany.my_app): com.acme.goodapps
Do you want to use SmartStore in your app? [yes/NO] ('No' by default)
Adjusting SalesforceSDK library project reference in project.properties.
Renaming application class to MyNativeAndroidAppApp in source.
Renaming application to MyNativeAndroidApp in source.
Renaming package name to com.acme.goodapps in source.
Moving source files to proper package path.
Renaming the app class filename to MyNativeAndroidAppApp.java.

Your application project is ready in /Users/rwhitley/Development/AndroidApps.
```

## Specifying Application Options Directly

You can also specify your configuration directly by typing the full forcedroid command string. To see usage information, type `forcedroid` without arguments. The list of available options displays:

```
$ node_modules/.bin/forcedroid
Usage:
forcedroid create
```

```
    --apptype=<Application Type> (native, hybrid_remote, hybrid_local)
    --appname=<Application Name>
    --targetdir=<Target App Folder>
    --packagename=<App Package Identifier> (com.my_company.my_app)
    --apexpage=<Path to Apex start page> (/apex/MyPage — Only required/used for
'hybrid_remote')
    [--usesmartstore=<Whether or not to use SmartStore> (--usesmartstore=true — false by
default)]
```

Using this information, type `forcedroid create`, followed by your options and values. For example:

```
$ node_modules/.bin/forcedroid create --apptype="native" --appname="package-test"
--targetdir="PackageTest" --packagename="com.test.my_new_app"
```

## Building and Running Your App From the Command Line

After the command line returns to the command prompt, the forcedroid script prints instructions for running Android utilities to configure and clean your project. Follow these instructions only if you want to build and run your app from the command line.

1. To build the new application, type the following commands at the command prompt:

   ```
   cd <your_project_directory>
   $ANDROID_SDK_DIR/tools/android update project -p . -t <id>
   ant clean debug
   ```

   where `ANDROID_SDK_DIR` points to your Android SDK directory.

   **Note:** The `-t <id>` parameter specifies API level of the target Android version. Use `android.bat list targets` to see the IDs for API versions installed on your system. See Native Android Requirements on page 28 for supported API levels.

2. If your emulator is not running, use the Android AVD Manager to start it. If you're using a device, connect it.
3. Type the following command at the command prompt:

   ```
   ant installd
   ```

**Note:** You can safely ignore the following warning:

```
It seems that there are sub-projects. If you want to update them please use the
--subprojects parameter.
```

The Android project you created contains a simple application you can build and run.

## 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 the `-targetdir` directory as your workspace directory.
2. Select **Window** > **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. If you set `-use_smartstore=true`, check the SmartStore project as well.
8. Click **Import**.
9. Repeat Steps 4–8. In Step 6, choose your target directory 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.

### forcedroid Command Parameters

These are descriptions of the forcedroid command parameters:

| Parameter Name | Description |
|---|---|
| `--apptype` | One of the following:<br>• "native"<br>• "hybrid_remote" (server-side hybrid app using VisualForce)<br>• "hybrid_local" (client-side hybrid app that doesn't use VisualForce) |
| `--appname` | Name of your application |
| `--targetdir` | Folder in which you want your project to be created. If the folder doesn't exist, the script creates it. |
| `--packagename` | Package identifier for your application (for example, "com.acme.app"). |
| `--apexpage` | (hybrid remote apps only) Server path to the Apex start page. For example: `/apex/MyAppStartPage`. |
| `--usesmartstore=true` | (Optional) Include only if you want to use SmartStore for offline data. Defaults to false if not specified. |

## Setting Up Sample Projects in Eclipse

The repository you cloned has other sample apps you can run. To import those into Eclipse:

1. Launch Eclipse and select `–target_dir` as your workspace directory.
2. If you haven't done so already, select **Window** > **Preferences**, choose the **Android** section, and enter the Android SDK location. Click OK.
3. Select **File** > **Import** and select **General** > **Existing Projects into Workspace**.
4. Click Next.
5. Select `forcedroid/native` as your root directory and import the projects listed in Android Project Files.

## Android Project Files

Inside the $NATIVE_DIR, you will find several projects:

1. `SalesforceSDK`—The SalesforceSDK, which provides support for OAuth2 and REST API calls
2. `test/SalesforceSDKTest`—Tests for the SalesforceSDK project
3. `TemplateApp`—Template used when creating new native applications using SalesforceSDK
4. `test/TemplateAppTest`—Tests for the TemplateApp project
5. `SampleApps/RestExplorer`—App using SalesforceSDK to explore the REST API calls
6. `SampleApps/NativeSqlAggregator` —A native app that uses SmartStore

# Developing a Native Android App

The native Android version of the Salesforce Mobile SDK empowers you to create rich mobile apps that directly use the Android operating system on the host device. To create these apps, you need to understand Java and Android development well enough to write code that uses Mobile SDK native classes.

## The create_native Script

The `create_native` script creates the app folder you specify, then populates it with a project file, build file, manifest file and resource files. Next, it copies the entire TemplateApp project to the new folder. It then updates the project properties, file names, class names, and directory paths to match the new app's configuration. As a result, your new project replicates all the settings and components used by the TemplateApp project.

If your new app supports SmartStore, the script also:

- Adds the SmartStore support library to the app directory.
- References the SmartStore library in the new project's properties.
- Changes the application class to extend `SalesforceSDKManagerWithSmartStore` rather than `SalesforceSDKManager`.

Finally, the script posts an important message:

```
"Before you ship, make sure to plug in your oauth client id and callback url in:
      ${target.dir}/res/values/bootconfig.xml"
```

If you're wondering where to get the OAuth client ID and callback URL, look in your connected app definition in your Salesforce organization. The OAuth client ID is the connected app's Consumer Key. The callback URL is the one you specified when you created your connected app. You enter these keys in the `res/values/bootconfig.xml` file of your project, which contains a few clearly named `<string>` nodes. Here's an example `bootconfig.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>

<resources>
    <string name="remoteAccessConsumerKey">3MVG92.uWdyphVj4bnolD7yuIpCQsNgddW
        tqRND3faxrv9uKnbj47H4RkwheHA2lKY4cBusvDVp0M6gdGE8hp</string>
    <string name="oauthRedirectURI">sfdc:///axm/detect/oauth/done</string>
    <string-array name="oauthScopes">
        <item>api</item>
    </string-array>
</resources>
```

The `create_native` script pre-populates `oauthRedirectURI` and `remoteAccessConsumerKey` strings with dummy values. Replace those values with the strings from your connected app definition.

## Android Application Structure

Typically, native Android apps that use the Mobile SDK require:

- An application entry point class that extends `android.app.Application`.
- At least one activity that extends `android.app.Activity`.

With the Mobile SDK, you:

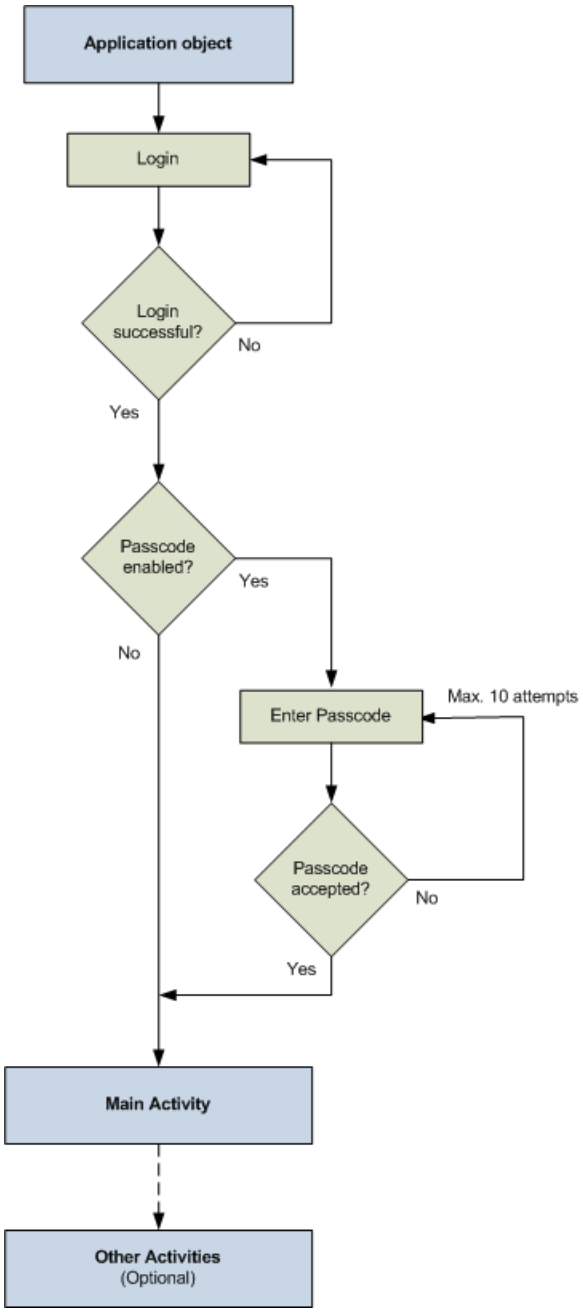
- Create a stub class that extends `android.app.Application`.
- Implement `onCreate()` in your `Application` stub class to call `SalesforceSDKManager.initNative()`.
- Extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`. This extension is optional but recommended.

The top-level `SalesforceSDKManager` class implements passcode functionality for apps that use passcodes, and fills in the blanks for those that don't. It also sets the stage for login, cleans up after logout, and provides a special event watcher that informs your app when a system-level account is deleted. OAuth protocols are handled automatically with internal classes.

The `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity` classes offer free handling of application pause and resume events and related passcode management. We recommend that you extend one of these classes for all activities in your app—not just the main activity. If you use a different base class for an activity, you're responsible for replicating the pause and resume protocols found in `SalesforceActivity`.

Within your activities, you interact with Salesforce objects by calling Salesforce REST APIs. The Mobile SDK provides the `com.salesforce.androidsdk.rest` package to simplify the REST request and response flow.

You define and customize user interface layouts, image sizes, strings, and other resources in XML files. Internally, the SDK uses an `R` class instance to retrieve and manipulate your resources. However, the Mobile SDK makes its resources directly accessible to client apps, so you don't need to write code to manage these features.




## Native API Packages

Salesforce Mobile SDK groups native APIs into seven packages. Here's a quick overview of these packages and points of interest within them.

| Package name | Description |
|---|---|
| `app` | Contains `SalesforceSDKManager`, the entry point class for all Mobile SDK applications. This package also contains app utility classes for internal use. |
| `auth` | Internal use only. Handles login, OAuth authentication, and HTTP access. |

| Package name | Description |
|---|---|
| `phonegap` | Internal classes used by hybrid applications to create a bridge between native code and Javascript code. Includes plugins that implement Mobile SDK Javascript libraries. If you want to implement your own Javascript plugin within an SDK app, extend `ForcePlugin` and implement the abstract `execute()` function. See ForcePlugin Class on page 41. |
| `rest` | Provides classes for handling REST API activities. These classes manage the communication with the Salesforce instance and handle the HTTP protocol for your REST requests. See `ClientManager` and `RestClient` for information on available synchronous and asynchronous methods for sending requests. |
| `security` | Internal classes that handle passcodes and encryption. If you provide your own key, you can use the `Encryptor` class to generate hashes. See `Encryptor`. |
| `ui`, `ui.sfhybrid`, `ui.sfnative` | Mostly internal classes that define the UI activities common to all Mobile SDK apps. These packages include `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity`, which are intended to serve individually as potential base classes for all app activities. |
| `util` | Contains utility and test classes. These classes are mostly for internal use, with some notable exceptions.<br>• You can register an instance of the `TokenRevocationReceiver` class to detect when an OAuth access token has been revoked.<br>• You can implement the `EventObserver` interface to eavesdrop on any event type.<br>• The `EventsListenerQueue` class is useful for implementing your own tests.<br>• Browse the `EventsObservable` source code to see a list of all supported event types. |

# Overview of Native Classes

This overview of the Mobile SDK native classes give you a look at pertinent details of each class and a sense of where to find what you need.

## SalesforceSDKManager Class

The `SalesforceSDKManager` class is the entry point for all native Android applications that use the Salesforce Mobile SDK. It provides mechanisms for:

- Login and logout

- Passcodes
- Encryption and decryption of user data
- String conversions
- User agent access
- Application termination
- Application cleanup

### initNative() Method

During startup, you initialize the singleton `SalesforceSDKManager` object by calling its static `initNative()` method. This method takes four arguments:

| Parameter Name | Description |
|---|---|
| `applicationContext` | An instance of `Context` that describes your application's context. In an `Application` extension class, you can satisfy this parameter by passing a call to `getApplicationContext()`. |
| `keyImplementation` | An instance of your implementation of the`KeyInterface` Mobile SDK interface. You are required to implement this interface. |
| `mainActivity` | The descriptor of the class that displays your main activity. The main activity is the first activity that displays after login. |
| `loginActivity` | (Optional) The class descriptor of your custom `LoginActivity` class. |

Here's an example from the TemplateApp:

```
SalesforceSDKManager.initNative(getApplicationContext(), new KeyImpl(), MainActivity.class);
```

In this example, `KeyImpl` is the app's implementation of `KeyInterface`. `MainActivity` subclasses `SalesforceActivity` and is designated here as the first activity to be called after login.

### logout() Method

The `SalesforceSDKManager.logout()` method clears user data. For example, if you've introduced your own resources that are user-specific, you don't want them to persist into the next user session. SmartStore destroys user data and account information automatically at logout.

Always call the superclass method somewhere in your method override, preferably after doing your own cleanup. Here's a pseudo-code example.

```
@Override
public void logout(Activity frontActivity) {
   // Clean up all persistent and non-persistent app artifacts
   // Call superclass after doing your own cleanup
   super.logout(frontActivity);
}
```

### getLoginActivityClass() Method

This method returns the descriptor for the login activity. The login activity defines the `WebView` through which the Salesforce server delivers the login dialog.

#### getUserAgent() Methods

The Mobile SDK builds a user agent string to publish the app's versioning information at runtime. This user agent takes the following form.

```
SalesforceMobileSDK/<salesforceSDK version> android/<android OS version> appName/appVersion
 <Native|Hybrid>
```

Here's a real-world example.

```
SalesforceMobileSDK/2.0 android mobile/4.2 RestExplorer/1.0 Native
```

To retrieve the user agent at runtime, call the `SalesforceSDKManager.getUserAgent()` method.

#### isHybrid() Method

Imagine that your Mobile SDK app creates libraries that are designed to serve both native and hybrid clients. Internally, the library code switches on the type of app that calls it, but you need some way to determine the app type at runtime. To determine the type of the calling app in code, call the boolean `SalesforceSDKManager.isHybrid()` method. True means hybrid, and false means native.

## KeyInterface Interface

KeyInterface is a required interface that you implement and pass into the `SalesforceSDKManager.initNative()` method.

#### getKey() Method

You are required to return a Base64-encoded encryption key from the `getKey()` abstract method. Use the `Encryptor.hash()` and `Encryptor.isBase64Encoded()` helper methods to generate suitable keys. The Mobile SDK uses your key to encrypt app data and account information.

## AccountWatcher Class

`AccountWatcher` informs your app when the user's account is removed through Settings. Without `AccountWatcher`, the application gets no notification of these changes. It's important to know when an account is removed so that its passcode and data can be disposed of properly, and logout can begin.

`AccountWatcher` defines an internal interface, `AccountRemoved`, that each app must implement. `SalesforceSDKManager` implements this interface to terminate the app's current (front) activity and reset the passcode, if used, and encryption key.

## PasscodeManager Class

The `PasscodeManager` class manages passcode encryption and displays the passcode page as required. It also reads mobile policies and caches them locally. This class is used internally to handle all passcode-related activities with minimal coding on your part. As a rule, apps call only these three `PasscodeManager` methods:

- `public void onPause(Activity ctx)`
- `public boolean onResume(Activity ctx)`
- `public void recordUserInteraction()`

These methods must be called in any native activity class that

- Is in an app that requires a passcode, and
- Does not extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`.

You get this implementation for free in any activity that extends `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`.

#### onPause() and onResume()

These methods handle the passcode dialog box when a user pauses and resumes the app. Call each of these methods in the matching methods of your activity class. For example, `SalesforceActivity.onPause()` calls `PasscodeManager.onPause()`, passing in its own class descriptor as the argument, before calling the superclass.

```
@Override
public void onPause() {
    passcodeManager.onPause(this);
    super.onPause();
}
```

Use the boolean return value of `PasscodeManager.onResume()` method as a condition for resuming other actions. In your app's `onResume()` implementation, be sure to call the superclass method before calling the `PasscodeManager` version. For example:

```
@Override
public void onResume() {
    super.onResume();
    // Bring up passcode screen if needed
    passcodeManager.onResume(this);
}
```

#### recordUserInteraction()

This method saves the time stamp of the most recent user interaction. Call `PasscodeManager.recordUserInteraction()` in the activity's `onUserInteraction()` method. For example:

```
@Override
public void onUserInteraction() {
    passcodeManager.recordUserInteraction();
}
```

## Encryptor class

The `Encryptor` helper class provides static helper methods for encrypting and decrypting strings using the hashes required by the SDK. It's important for native apps to remember that all keys used by the Mobile SDK must be Base64-encoded. No other encryption patterns are accepted. Use the `Encryptor` class when creating hashes to ensure that you use the correct encoding.

Most `Encryptor` methods are for internal use, but apps are free to use this utility as needed. For example, if an app implements its own database, it can use `Encryptor` as a free encryption and decryption tool.

## SalesforceActivity, SalesforceListActivity, and SalesforceExpandableListActivity Classes

`SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity` are the skeletal base classes for native SDK activities. They extend `android.app.Activity`, `android.app.ListActivity`, and `android.app.ExpandableListActivity`, respectively.

Each of these classes provides a free implementation of `PasscodeManager` calls. When possible, it's a good idea to extend one of these classes for all of your app's activities, even if your app doesn't currently use passcodes.

**For passcode-protected apps:** If any of your activities don't extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`, you'll need to add a bit of passcode protocol to each of those activities. See Using Passcodes on page 42

Each of these activity classes contain a single abstract method:

```
public abstract void onResume(RestClient client);
```

This method overloads the `Activity.onResume()` method, which is implemented by the class. The class method calls your overload after it instantiates a `RestClient` instance. Use this method to cache the client that's passed in, and then use that client to perform your REST requests.

## UI Classes

Activities in the `com.salesforce.androidsdk.ui` package represent the UI resources that are common to all Mobile SDK apps. You can style, skin, theme, or otherwise customize these resources through XML. With the exceptions of `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity`, do not override these activity classes with intentions of replacing the resources at runtime.

## ClientManager and RestClient Classes

`ClientManager` works with the Android `AccountManager` class to manage user accounts. More importantly for apps, it provides access to `RestClient` instances through two methods:

- `getRestClient()`
- `peekRestClient()`

The `getRestClient()` method asynchronously creates a `RestClient` instance for querying Salesforce data. Asynchronous in this case means that this method is intended for use on UI threads. The `peekRestClient()` method creates a `RestClient` instance synchronously, for use in non-UI contexts.

Once you get the `RestClient` instance, you can use it to send REST API calls to Salesforce. Again, the method you call depends on whether you're calling from a UI context. The `RestClient` methods for sending HTTP requests are:

- `sendAsync()`—Call this method if you called `ClientManager.getRestClient()`
- `sendSync()`—Call this method if you called `ClientManager.peekRestClient()`

You can choose from three overloads of `RestClient.sendSync()`, depending on the degree of information you can provide for the request.

## LoginActivity Class

`LoginActivity` defines the login screen. The login workflow is worth describing because it explains two other classes in the activity package. In the login activity, if you press the Menu button, you get three options: **Clear Cookies**, **Reload**, and **Pick Server**. **Pick Server** launches an instance of the `ServerPickerActivity` class, which displays **Production**, **Sandbox**, and **Custom Server** options. When a user chooses **Custom Server**, `ServerPickerActivity` launches an instance of the `CustomServerURLEditor` class. This class displays a popover dialog that lets you type in the name of the custom server.

## Other UI Classes

Several other classes in the `ui` package are worth mentioning, although they don't affect your native API development efforts.

The `PasscodeActivity` class provides the UI for the passcode screen. It runs in one of three modes: Create, CreateConfirm, and Check. Create mode is presented the first time a user attempts to log in. It prompts the user to create a passcode. After the user submits the passcode, the screen returns in CreateConfirm mode, asking the user to confirm the new passcode. Thereafter, that user sees the screen in Check mode, which simply requires the user to enter the passcode.

`SalesforceR` is a deprecated class. This class was required when the Mobile SDK was delivered in JAR format, to allow developers to edit resources in the binary file. Now that the Mobile SDK is available as a library project, `SalesforceR` is not needed. Instead, you can override resources in the SDK with your own.

`SalesforceDroidGapActivity` and `SalesforceGapViewClient` are used only in hybrid apps.

## UpgradeManager Class

`UpgradeManager` provides a mechanism for silently upgrading the SDK version installed on a device. This class stores the SDK version information in a shared preferences file on the device. To perform an upgrade, `UpgradeManager` queries the current `SalesforceSDKManager` instance for its SDK version and compares its version to the device's version information. If an upgrade is necessary—for example, if there are changes to a database schema or to encryption patterns—`UpgradeManager` can take the necessary steps to upgrade SDK components on the device. This class is intended for future use. Its implementation in Mobile SDK 2.0 simply stores and compares the version string.

## Utility Classes

Though most of the classes in the `util` package are for internal use, several of them can also benefit third-party developers.

| Class | Description |
|---|---|
| `EventsObservable` | See the source code for a list of all events that the Mobile SDK for Android propagates. |
| `EventsObserver` | Implement this interface to eavesdrop on any event. This functionality is useful if you're doing something special when certain types of events occur. |
| `TokenRevocationReceiver` | This class handles what happens when an administrator revokes a user's refresh token. See Handling Refresh Token Revocation in Android Native Apps on page 126. |
| `UriFragmentParser` | You can directly call this static helper class. It parses a given URI, breaks its parameters into a series of key/value pairs, and returns them in a map. |

## ForcePlugin Class

All classes in the`com.salesforce.androidsdk.phonegap` package are intended for hybrid app support. Most of these classes implement Javascript plugins that access native code. The base class for these Mobile SDK plugins is `ForcePlugin`. If you want to implement your own Javascript plugin in a Mobile SDK app, extend `ForcePlugin`, and implement the abstract `execute()` function.

`ForcePlugin` extends `CordovaPlugin`, which works with the Javascript framework to let you create a Javascript module that can call into native functions. PhoneGap provides the bridge on both sides: you create a native plugin with `CordovaPlugin`, then you create a Javascript file that mirrors it. Cordova calls the plugin's `execute()` function when a script calls one of the plugin's Javascript functions.

# Using Passcodes

User data in Mobile SDK apps is secured by encryption. The administrator of your Salesforce org has the option of requiring the user to enter a passcode for connected apps. In this case, your app uses that passcode as an encryption hash key. If the Salesforce administrator doesn't require a passcode, you're responsible for providing your own key.

Salesforce Mobile SDK does all the work of implementing the passcode workflow. It calls the passcode manager to obtain the user input, and then combines the passcode with prefix and suffix strings into a hash for encrypting the user's data. It also handles decrypting and re-encrypting data when the passcode changes. If an organization changes its passcode requirement, the Mobile SDK detects the change at the next login and reacts accordingly. If you choose to use a passcode, your only responsibility is to implement the `SalesforceSDKManager.getKey()` method. All your implementation has to do in this case is return a Base64-encoded string that can be used as an encryption key.

Internally, passcodes are stored as Base64-encoded strings. The SDK uses the `Encryptor` class for creating hashes from passcodes. You should also use this class to generate a hash when you provide a key instead of a passcode. Passcodes and keys are used to encrypt and decrypt SmartStore data as well as oAuth tokens, user identification strings, and related security information. To see exactly what security data is encrypted with passcodes, browse the `ClientManager.changePasscode()` method.

Mobile policy defines certain passcode attributes, such as the length of the passcode and the timing of the passcode dialog. Mobile policy files for connected apps live on the Salesforce server. If a user enters an incorrect passcode more than ten consecutive times, the user is logged out. The Mobile SDK provides feedback when the user enters an incorrect passcode, apprising the user of how many more attempts are allowed. Before the screen is locked, the `PasscodeManager` class stores a reference to the front activity so that the same activity can be resumed if the screen is unlocked.

If you define activities that don't extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity` in a passcode-protected app, be sure to call these three `PasscodeManager` methods from each of those activity classes:

- `PasscodeManager.onPause()`
- `PasscodeManager.onResume(Activity)`
- `PasscodeManager.recordUserInteraction()`

Call `onPause()` and `onResume()` from your activity's methods of the same name. Call `recordUserInteraction()` from your activity's `onUserInteraction()` method. Pass your activity class descriptor to `onResume()`. These calls ensure that your app enforces passcode security during these events. See PasscodeManager Class on page 38.

**Note:** The `SalesforceActivity`, `SalesforceListActivity`, and `SalesforceExpandableListActivity` classes implement these mandatory methods for you for free. Whenever possible, base your activity classes on one of these classes.

# Resource Handling

Salesforce Mobile SDK resources are configured in XML files that reside in the `native/SalesforceSDK/res` folder. You can customize many of these resources by making changes in this folder.

Resources in the `/res` folder are grouped into categories, including:

- Drawables—Backgrounds, drop shadows, image resources such as PNG files
- Layouts—Screen configuration for any visible component, such as the passcode screen
- Values—Strings, colors, and dimensions that are used by the SDK

Two additional resource types are mostly for internal use:

- Menus
- XML

Drawable, layout, and value resources are subcategorized into folders that correspond to a variety of form factors. These categories handle different device types and screen resolutions. Each category is defined in its folder name, which allows the resource file name to remain the same for all versions. For example, if the developer provides various sizes of an icon named `icon1.png`, for example, the smart phone version goes in one folder, the low-end phone version goes in another folder, while the tablet icon goes into a third folder. In each folder, the file name is `icon1.png`. The folder names use the same root but with different suffixes.

The following table describes the folder names and suffixes.

| Folder name | Usage |
|---|---|
| `drawable` | Generic versions of drawable resources |
| `drawable-hdpi` | High resolution; for most smart phones |
| `drawable-ldpi` | Low resolution; for low-end feature phones |
| `drawable-mdpi` | Medium resolution; for low-end smart phones |
| `drawable-xlarge` | For tablet screens in landscape orientation |
| `drawable-xlarge-port` | For tablet screens in portrait orientation |
| `layout` | Generic versions of layouts |
| `layout-land` | For landscape orientation |
| `layout-xlarge` | For tablet screens |
| `values` | Generic styles and values |
| `values-xlarge` | For tablet screens |

The compiler looks for a resource in the folder whose name matches the target device configuration. If the requested resource isn't in the expected folder (for example, if the target device is a tablet, but the compiler can't find the requested icon in the `drawables-xlarge` or `drawables-xlarge-port` folder) the compiler looks for the icon file in the generic `drawable` folder.

## Layouts

Layouts in the Mobile SDK describe the screen resources that all apps use. For example, layouts configure dialog boxes that handle logins and passcodes.

The name of an XML node in a layout indicates the type of control it describes. For example, the following `EditText` node from `res/layout/sf__passcode.xml` describes a text edit control:

```
      <EditText android:id="@+id/sf__passcode_text"
          style="@style/SalesforceSDK.Passcode.Text.Entry"
          android:inputType="textPassword" />
```

In this case, the `EditText` control uses an `android:inputType` attribute. Its value, "textPassword", tells the operating system to obfuscate the typed input.

The style attribute references a global style defined elsewhere in the resources. Instead of specifying style attributes in place, you define styles defined in a central file, and then reference the attribute anywhere it's needed. The value

`@style/SalesforceSDK.Passcode.Text.Entry` refers to an SDK-owned style defined in `res/values/sf__styles.xml`. Here's the style definition.

```
<style name="SalesforceSDK.Passcode.Text.Entry">
    <item name="android:layout_width">wrap_content</item>
    <item name="android:lines">1</item>
    <item name="android:maxLength">10</item>
    <item name="android:minWidth">@dimen/sf__passcode_text_min_width</item>
    <item name="android:imeOptions">actionGo</item>
</style>
```

You can override any style attribute with a reference to one of your own styles. Rather than changing `sf__styles.xml`, define your styles in a different file, such as `xyzcorp__styles.xml`. Place your file in the `res/values` for generic device styles, or the `res/values-xlarge` folder for tablet devices.

### Values

The res/values and res/values-xlarge folders contain definitions of style components, such as dimens and colors, string resources, and custom styles. File names in this folder indicate the type of resource or style component. To provide your own values, create new files in the same folders using a file name prefix that reflects your own company or project. For example, if your developer prefix is XYZ, you can override `sf__styles.xml` in a new file named `XYZ__styles.xml`.

| File name | Contains |
|---|---|
| `sf__colors.xml` | Colors referenced by Mobile SDK styles |
| `sf__dimens.xml` | Dimensions referenced by Mobile SDK styles |
| `sf__strings.xml` | Strings referenced by Mobile SDK styles; error messages can be overridden |
| `sf__styles.xml` | Visual styles used by the Mobile SDK |
| `strings.xml` | App-defined strings |

You can override the values in `strings.xml`. However, if you used the `create_native` script to create your app, strings in `strings.xml` already reflect appropriate values.

### Other Resources

Two other folders contain Mobile SDK resources.

- `res/menu` defines menus used internally. If your app defines new menus, add them as resources here in new files.
- `res/xml` includes one file that you must edit: `servers.xml`. In this file, change the default Production and Sandbox servers to the login servers for your org. The other files in this folder are for internal use. The `authenticator.xml` file configures the account authentication resource, and the `config.xml` file defines PhoneGap plugins for hybrid apps.

## Using REST APIs

To query, describe, create, or update data from a Salesforce org, native apps call Salesforce REST APIs. Salesforce REST APIs honor SOQL strings and can accept and return data in either JSON or XML format. REST APIs are fully documented at REST API Developer's Guide You can find links to related Salesforce development documentation at the Force.com developer documentation website..

With Android native apps, you do only minimal coding to access Salesforce data through REST calls. The classes in the `com.salesforce.androidsdk.rest` package initialize the communication channels and encapsulate low-level HTTP plumbing. These classes include:

- `ClientManager`—Serves as a factory for RestClient instances. It also handles account logins and handshakes with the Salesforce server. Implemented by the Mobile SDK.
- `RestClient`—Handles protocol for sending REST API requests to the Salesforce server. Don't directly create instances of `RestClient`. Instead, call the `ClientManager.getRestClient()` method. Implemented by the Mobile SDK.
- `RestRequest`—Formats REST API requests from the data your app provides. Also serves as a factory for instances of itself. Don't directly create instances of `RestRequest`. Instead, call an appropriate `RestRequest` static getter function such as `RestRequest.getRequestForCreate()`. Implemented by the SDK.
- `RestResponse`—Formats the response content in the requested format, returns the formatted response to your app, and closes the content stream. The `RestRequest` class creates instances of `RestResponse` and returns them to your app through your implementation of the `RestClient.AsyncRequestCallback` interface. Implemented by the SDK.

The `RestRequest` class natively handles the standard Salesforce data operations offered by the Salesforce REST and SOAP APIs. Supported operations are:

| Operation | Parameters | Description |
|---|---|---|
| Versions | None | Returns Salesforce version metadata |
| Resources | API version | Returns available resources for the specified API version, including resource name and URI |
| Metadata | API version, object type | |
| DescribeGlobal | API version | Returns a list of all available objects in your org and their metadata |
| Describe | API version, object type | Returns a description of a single object type |
| Create | API version, object type, map of field names to value objects | Creates a new record in the specified object |
| Retrieve | API version, object type, object ID, list of fields | Retrieves a record by object ID |
| Update | API version, object type, object ID, map of field names to value objects | Updates an object with the given map |
| Upsert | API version, object type, external ID field, external ID, map of field names to value objects | Updates or inserts an object from external data, based on whether the external ID currently exists in the external ID field |
| Delete | API version, object type, object ID | Deletes the object of the given type with the given ID |

To obtain an appropriate `RestRequest` instance, call the `RestRequest` static method that matches the operation you want to perform. Here are the RestRequest static methods.

- `getRequestForCreate()`
- `getRequestForDelete()`
- `getRequestForDescribe()`

- `getRequestForDescribeGlobal()`
- `getRequestForMetadata()`
- `getRequestForQuery()`
- `getRequestForResources()`
- `getRequestForRetrieve()`
- `getRequestForSearch()`
- `getRequestForUpdate()`
- `getRequestForUpsert()`
- `getRequestForVersions()`

These methods return a `RestRequest` object which you pass to an instance of `RestClient`. The `RestClient` class provides synchronous and asynchronous methods for sending requests: `sendSync()` and `sendAsync()`. Use`sendAsync()` when you're sending a request from a UI thread. Use `sendSync()` only on non-UI threads, such as a service or a worker thread spawned by an activity.

Here's the basic procedure for using the REST classes on a UI thread:

1. Create an instance of `ClientManager`.
   a. Use the `SalesforceSDKManager.getInstance().getAccountType()` method to obtain the value to pass as the second argument of the `ClientManager` constructor.
   b. For the `LoginOptions` parameter of the `ClientManager` constructor, call `SalesforceSDKManager.GetInstance().getLoginOptions()`.
2. Implement the `ClientManager.RestClientCallback` interface.
3. Call `ClientManager.getRestClient()` to obtain a `RestClient` instance, passing it an instance of your `RestClientCallback` implementation. This code from the `native/SampleApps/RestExplorer` sample app implements and instantiates `RestClientCallback` inline:

```
String accountType = SalesforceSDKManager.getInstance().getAccountType();

LoginOptions loginOptions = SalesforceSDKManager.getInstance().getLoginOptions();
// Get a rest client
new ClientManager(this, accountType, loginOptions,
SalesforceSDKManager.getInstance().shouldLogoutWhenTokenRevoked()).getRestClient(this,
 new RestClientCallback() {
  @Override
  public void authenticatedRestClient(RestClient client) {
   if (client == null) {
    SalesforceSDKManager.getInstance().logout(ExplorerActivity.this);
    return;
   }
   // Cache the returned client
   ExplorerActivity.this.client = client;
  }
 });
```

4. Call a static `RestRequest()` getter method to obtain the appropriate `RestRequest` object for your needs. For example, to get a description of a Salesforce object:

```
request = RestRequest.getRequestForDescribe(apiVersion, objectType);
```

5. Pass the `RestRequest` object you obtained in the previous step to `RestClient.sendAsync()` or `RestClient.sendSync()`. If you're on a UI thread and therefore calling `sendAsync()`:
   a. Implement the `ClientManager.AsyncRequestCallback` interface.

**b.** Pass an instance of your implementation to the `sendAsync()` method.

**c.** Receive the formatted response through your `ASyncRequestCallback.onSuccess()` method.

The following code implements and instantiates `ASyncRequestCallback` inline:

```
private void sendFromUIThread(RestRequest restRequest) {
 client.sendAsync(restRequest, new AsyncRequestCallback() {
  private long start = System.nanoTime();
  @Override
  public void onSuccess(RestRequest request, RestResponse result) {
   try
   {
    // Do something with the result
   }
   catch (Exception e) {
    printException(e);
   }
   EventsObservable.get().notifyEvent(EventType.RenditionComplete);
  }
  @Override
  public void onError(Exception exception)
  {
   printException(exception);
   EventsObservable.get().notifyEvent(EventType.RenditionComplete);
  }
 });
```

If you're calling the `sendSync()` method from a service, use the same procedure with the following changes:

1. To obtain a `RestClient` instance call `ClientManager.peekRestClient()` instead of `ClientManager.getRestClient()`.
2. Retrieve your formatted REST response from the `sendSync()` method's return value.

# Android Template App: Deep Dive

The TemplateApp sample project implements everything you need to create a basic Android app. Because it's a "bare bones" example, it also serves as the template that the Mobile SDK's create_native ant script uses to set up new native Android projects. You can gain a quick understanding of the native Android SDK by studying this project.

The TemplateApp project defines two classes, `TemplateApp` and `MainActivity`. The `TemplateApp` class extends `Application` and calls `SalesforceSDKManager.initNative()` in its `onCreate()` override. The `MainActivity` class subclasses the `SalesforceActivity` class. These two classes are all you need to create a running mobile app that displays a login screen and a home screen.

Despite containing only about 200 lines of code, TemplateApp is more than just a "Hello World" example. In its main activity, it retrieves Salesforce data through REST requests and displays the results on a mobile page. You can extend TemplateApp by adding more activities, calling other components, and doing anything else that the Android operating system, the device, and security restraints allow.

## TemplateApp Class

Every native Android app requires an instance of `android.app.Application`. Here's the entire class:

```
package com.salesforce.samples.templateapp;

import android.app.Application;

import com.salesforce.androidsdk.app.SalesforceSDKManager;
```

```
/**
 * Application class for our application.
 */
public class TemplateApp extends Application {

 @Override
 public void onCreate() {
  super.onCreate();
  SalesforceSDKManager.initNative(getApplicationContext(), new KeyImpl(), MainActivity.class);

 }
}
```

The TemplateApp class accomplishes two main tasks:

- Calls `initNative()` to initialize the app
- Passes in the app's implementation of `KeyInterface`

Most native Android apps can use similar code. For this small amount of work, your app gets free implementations of passcode and login/logout mechanisms, plus a few other benefits. See SalesforceActivity, SalesforceListActivity, and SalesforceExpandableListActivity Classes on page 39.

## MainActivity Class

In Mobile SDK apps, the main activity begins immediately after the user logs in. Once the main activity is running, it can launch other activities, which in turn can launch sub-activities. When the application exits, it does so by terminating the main activity. All other activities terminate in a cascade from within the main activity.

The `MainActivity` class for the Template app extends `com.salesforce.androidsdk.ui.sfnative.SalesforceActivity`. This superclass is the Mobile SDK's basic abstract activity class.`SalesforceActivity`, gives you free implementations of mandatory passcode and login protocols. If you use another base activity class instead, you're responsible for implementing those protocols. `MainActivity` initializes the app's UI and implements its UI buttons. The UI includes a list view that can show the user's Salesforce Contacts or Accounts. When the user clicks one of these buttons, the `MainActivity` object performs a couple of basic queries to populate the view. For example, to fetch the user's Contacts from Salesforce, the `onFetchContactsClick()` message handler sends a simple SOQL query:

```
public void onFetchContactsClick(View v) throws UnsupportedEncodingException {
       sendRequest("SELECT Name FROM Contact");
}
```

Internally, the private `sendRequest()` method formulates a server request using the `RestRequest` class and the given SOQL string:

```
private void sendRequest(String soql) throws UnsupportedEncodingException
{
 RestRequest restRequest = RestRequest.getRequestForQuery(getString(R.string.api_version),
  soql);
 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();
  }
 });
}
```

This method uses an instance of the `com.salesforce.androidsdk.rest.RestClient` class, `client`, to process its SOQL query. The `RestClient` class relies on two helper classes—`RestRequest` and `RestResponse`—to send the query and process its result. The `sendRequest()` method calls `RestClient.sendAsync()` to process the SOQL query asynchronously.

To support the `sendAsync()` call, the `sendRequest()` method constructs an instance of `com.salesforce.androidsdk.rest.RestRequest`, passing it the API version and the SOQL query string. The resulting object is the first argument for `sendAsync()`. The second argument is a callback object. When `sendAsync()` has finished running the query, it sends the results to this callback object. If the query is successful, the callback object uses the query results to populate a UI list control. If the query fails, the callback object displays a toast popup to display the error message.

***Java Note:***

In the call to`RestClient.sendAsync()` the code instantiates a new `AsyncRequestCallback` object as its second argument. However, the `AsyncRequestCallbackconstructor` is followed by a code block that overrides a couple of methods: `onSuccess()` and `onError()`. If that code looks strange to you, take a moment to see what's happening. `ASyncRequestCallback` is defined as an interface, so it has no implementation. In order to instantiate it, the code implements the two `ASyncRequestCallback` methods inline to create an anonymous class object. This technique gives `TemplateApp` an `sendAsync()` implementation of its own that can never be called from another object and doesn't litter the API landscape with a group of specialized class names.

## TemplateApp Manifest

A look at the `AndroidManifest.xml` file in the TemplateApp project reveals the components required for Mobile SDK native Android apps. Required components include:

| Name | Type | Description |
|---|---|---|
| `com.salesforce.androidsdk.auth.AuthenticatorService` | Service | Validates the user's credentials against the Salesforce OAuth module. |
| `MainActivity` | Activity | The first activity to be called after login. The name and the class are defined in the project. |
| `com.salesforce.androidsdk.ui.LoginActivity` | Activity | Displays the Salesforce login screen. |
| `com.salesforce.androidsdk.ui.PasscodeActivity` | Activity | Displays the passcode screen. Used only if the Salesforce administrator |

| Name | Type | Description |
|---|---|---|
| | | requires a passcode for the corresponding Connected App. This requirement can change at any time on the server, but the Mobile SDK checks the policy only during login. |
| `com.salesforce.androidsdk.ui.ServerPickerActivity` | Activity | Displays a list of Salesforce login servers from which the user can choose. This activity also lets users add custom servers. |
| `com.salesforce.androidsdk.ui.ManageSpaceActivity` | Activity | Displayed when the user clicks on **Manage Space** in the Settings app. Warns the user that clearing user data from Settings causes the user to be logged out. |

Because apps created by the `create_native` script are based on the TemplateApp project, you don't need to add these components to the manifest. As with any Android app, you can add other components, such as custom activities or services, using the Android Manifest editor in Eclipse.

In addition to component specifications, the manifest grants Android permissions to the app. Grants in TemplateApp include:

- `android.permission.INTERNET`
- `android.permission.MANAGE_ACCOUNTS`
- `android.permission.AUTHENTICATE_ACCOUNTS`
- `android.permission.GET_ACCOUNTS`
- `android.permission.USE_CREDENTIALS`
- `android.permission.ACCESS_NETWORK_STATE`

Most of these permissions provide access to Android user accounts. For details, search for manifest permissions in the Android SDK documentation.

# Android Sample Applications

RestExplorer is a sample app that demonstrates how to use the OAuth and REST API functions of the SalesforceSDK. It's also useful to investigate the various REST API actions from a Honeycomb tablet.

1. To run the application from your Eclipse workspace, right-click the **RestExplorer** project and choose **Run As** > **Android Application**.
2. To run the tests, right-click the **RestExplorerTest** project and choose **Run As** > **Android JUnit Test**.

NativeSqlAggregator is a sample app that demonstrates SQL aggregation with SmartSQL. As such, it also demonstrates a native implementation of SmartStore. To run the application from your Eclipse workspace, right-click the **NativeSqlAggregator** project and choose **Run As** > **Android Application**.

# Chapter 4

# Introduction to Hybrid Development

**In this chapter ...**



Hybrid apps combine the ease of HTML5 Web app development with the power and features of the native platform. They run within the Salesforce Mobile Container

, a native layer that translates the app into device-specific code.

Hybrid apps depend on HTML and JavaScript files. These files can be stored on the device or on the server.

- **Device**—Hybrid apps developed with `forcetk.mobilesdk` wrap a Web app inside the Salesforce Mobile Container. In this scenario, the JavaScript and HTML files are stored on the device.
- **Server** — Hybrid apps developed using Visualforce technology store their HTML and JavaScript files on the Salesforce server and are delivered through the Salesforce Mobile Container.

# iOS Hybrid Development

In order to develop hybrid applications, you'll need to meet some of the prerequisites for both the iOS native and the vanilla HTML5 scenarios.

1. Make sure you meet the HTML5 Development
2. Follow the installation instructions for iOS.

## iOS Hybrid Sample Application

The sample applications contained under the `hybrid/SampleApps` folder are designed around the PhoneGap SDK. PhoneGap is also known as Cordova. Salesforce Mobile SDK v. 1.4 and later include the Cordova libraries, so no separate installation is required. You can find documentation for the Cordova SDK in the Getting Started Guide.

Inside the `hybrid/SampleApps` folder, you can find sample projects:

- **AccountEditor**: Demonstrates how to use the SmartSync Data Framework to access Salesforce data.
- **ContactExplorer**: The `ContactExplorer` sample app uses PhoneGap (also known as Cordova) to retrieve local device contacts. It also uses the `forcetk.mobilesdk.js` toolkit to implement REST transactions with the Salesforce REST API. The app uses the OAuth2 support in Salesforce SDK to obtain OAuth credentials, then propagates those credentials to `forcetk.mobilesdk.js` by sending a JavaScript event.
- **VFConnector**: The VFConnector sample app demonstrates how to wrap a Visualforce page in a native container. This example assumes that your org has a Visualforce page called `BasicVFTest`. The app first obtains OAuth login credentials using the Salesforce SDK OAuth2 support, then uses those credentials to set appropriate webview cookies for accessing Visualforce pages.
- **SmartStoreExplorer**: Lets you explore SmartStore APIs.

# Android Hybrid Development

In order to develop hybrid applications, you'll need to meet some of the prerequisites for both the Android native and the vanilla HTML5 scenarios.

1. Make sure you meet the HTML5 Development.
2. Follow the installation instructions for Android Native.
3. After installing Mobile SDK for Android, create a new hybrid app as described in Creating a New Android Project on page 30. For the `apptype` parameter:
   - Use `--apptype="hybrid_local"` for a hybrid app with all code in the local project. Put your HTML and JavaScript files in `${target.dir}/assets/www/`.
   - Use `--apptype="hybrid_remote"` for a hybrid app with code in a Visualforce app on the server

## Hybrid Sample Applications

Inside the `./hybrid` folder, you can find sample projects and related test applications:

- **AccountEditor**: Demonstrates how to use the SmartSync Data Framework to access Salesforce data.
- **SampleApps/ContactExplorer**: The `ContactExplorer` sample app uses PhoneGap (also known as Cordova) to retrieve local device contacts. It also uses the `forcetk.mobilesdk.js` toolkit to implement REST transactions with the Salesforce

REST API. The app uses the OAuth2 support in Salesforce SDK to obtain OAuth credentials, then propagates those credentials to `forcetk.mobilesdk.js` by sending a javascript event.
- **SampleApps/test/ContactExplorerTest**: Tests for the ContactExplorer sample app.
- **SampleApps/VFConnector**: The VFConnector sample app demonstrates how to wrap a Visualforce page in a native container. This example assumes that your org has a Visualforce page called `BasicVFTest`. The app first obtains OAuth login credentials using the Salesforce SDK OAuth2 support, then uses those credentials to set appropriate webview cookies for accessing Visualforce pages.
- **SampleApps/test/VFConnectorTest**: Test for the VFConnector sample app.
- **SampleApps/SmartStoreExplorer**: Lets you explore SmartStore APIs.
- **SampleApps/test/SmartStoreExplorerTest**: Tests for the SmartStoreExplorer sample app.

# JavaScript Files for Hybrid Applications

In Salesforce Mobile SDK 2.0, we've refactored some JavaScript files and added new ones to support SmartSync. JavaScript files reside in the `forcedotcom/SalesforceMobileSDK-Shared` repository on GitHub.

## Refactored JavaScript Files

These files are now collected in the `cordova.force.js` file.

- `SFHybridApp.js`
- `SalesforceOAuthPlugin.js`
- `SmartStorePlugin.js`

## New JavaScript Files

These files are new in Mobile SDK 2.0.

| JavaScript File | Description |
|---|---|
| cordova.force.js | Contains plugins for hybrid apps using the Cordova libraries |
| SmartSync.js | The SmartSync Data Framework library |

## New External Dependencies

Mobile SDK 2.0 introduces new external dependencies.

| External JavaScript File | Description |
|---|---|
| jquery.js | Popular HTML utility library |
| underscore.js | SmartSync support |
| backbone.js | SmartSync support |

## Which JavaScript Files Do I Include?

Files that you include depend on the type of hybrid project. For each type described here, include all files in the list.

**For basic hybrid apps:**

- `cordova.js`
- `cordova.force.js`

**To make REST API calls from a basic hybrid app:**

- `cordova.js`
- `cordova.force.js`
- `forcetk.mobilesdk.js`

**To use SmartSync in a hybrid app:**

- `jquery.js`
- `underscore.js`
- `backbone.js`
- `cordova.js`
- `cordova.force.js`
- `forcetk.mobilesdk.js`
- `SmartSync.js`

# Versioning and Javascript Library Compatibility

In hybrid applications, client Javascript code interacts with native code through Cordova (formerly PhoneGap) and SalesforceSDK plugins. When you package your Javascript code with your mobile application, your testing assures that the code works with native code. However, when the Javascript code comes from the server—for example, when the application is written with VisualForce—harmful conflicts can occur. In such cases you must be careful to use Javascript libraries from the version of PhoneGap or Cordova that matches the Mobile SDK version you're using.

For example, suppose you shipped an application with Mobile SDK 1.2, which uses PhoneGap 1.2. Later, you ship an update that uses Mobile SDK 1.3. The 1.3 version of the Mobile SDK uses Cordova 1.8.1 rather than PhoneGap 1.2. You must make sure that the Javascript code in your updated application accesses native components only through the Cordova 1.8.1 and Mobile SDK 1.3 versions of the Javascript libraries. Using mismatched Javascript libraries can crash your application.

You can't force your customers to upgrade their clients, so how can you prevent crashes? First, identify the version of the client. Then, you can either deny access to the application if the client is outdated (for example, with a "Please update to the latest version" warning), or, preferably, serve compatible Javascript libraries.

The following table correlates Cordova and PhoneGap versions to Mobile SDK versions.

| Mobile SDK version | Cordova or PhoneGap version |
|---|---|
| 1.2 | PhoneGap 1.2 |
| 1.3 | Cordova 1.8.1 |
| 1.4 | Cordova 2.2 |
| 1.5 | Cordova 2.3 |
| 2.0 | Cordova 2.3 |

## Using the User Agent to Find the Mobile SDK Version

Fortunately, you can look up the Mobile SDK version in the user agent. The user agent starts with `SalesforceMobileSDK/<version>`. Once you obtain the user agent, you can parse the returned string to find the Mobile SDK version.

You can obtain the user agent on the server with the following Apex code:

```
userAgent = ApexPages.currentPage().getHeaders().get('User-Agent');
```

On the client, you can do the same in Javascript using the `navigator` object:

```
userAgent = navigator.userAgent;
```

### Detecting the Mobile SDK Version with the sdkinfo Plugin

In Javascript, you can also retrieve the Mobile SDK version and other information by using the `sdkinfo` plugin. This plugin, which is defined in the `cordova.force.js` file, offers one method:

```
getInfo(callback)
```

This method returns an associative array that provides the following information:

| Member name | Description |
|---|---|
| `sdkVersion` | Version of the Salesforce Mobile SDK used to build to the container. For example: “1.4”. |
| `appName` | Name of the hybrid application. |
| `appVersion` | Version of the hybrid application. |
| `forcePluginsAvailable` | Array containing the names of Salesforce plugins installed in the container. For example: "com.salesforce.oauth", "com.salesforce.smartstore", and so on. |

The following code retrieves the information stored in the `sdkinfo` plugin and displays it in alert boxes.

```
var sdkinfo = cordova.require("salesforce/plugin/sdkinfo");
sdkinfo.getInfo(new function(info) {
    alert("sdkVersion->" + info.sdkVersion);
    alert("appName->" + info.appName);
    alert("appVersion->" + info.appVersion);
    alert("forcePluginsAvailable->" + JSON.stringify(info.forcePluginsAvailable));
});
```

**See Also:**

*Example: Serving the Appropriate Javascript Libraries*

# Managing Sessions in Hybrid Applications

Mobile users expect their apps to just work. To help iron out common difficulties that plague many mobile apps, the Mobile SDK uses native containers for hybrid applications. These containers provide seamless authentication and session management by abstracting the complexity of web session management. However, as popular mobile app architectures evolve, this “one size fits all” approach proves to be too limiting in some cases. For example, if a mobile app uses JavaScript remoting in Visualforce, Salesforce cookies can be lost if the user lets the session expire. These cookies can be retrieved only when the user manually logs back in.

Mobile SDK 1.4 begins to transition hybrid apps away from predefined, proactive session management to more flexible, reactive session management. Rather than letting the hybrid container automatically control the session, developers can participate in the management by responding to session events. This change gives developers more control over managing sessions in the Salesforce Touch Platform.

To switch to reactive management, adjust your session management settings according to your app’s architecture. This table summarizes the behaviors and recommended approaches for common architectures.

| App Architecture | Proactive Behavior in SDK 1.3 and Earlier | Reactive Behavior in SDK 1.4 | Steps for Upgrading Code |
|---|---|---|---|
| REST API | Background session refresh | Refresh from JavaScript | No change for forcetk.mobilesdk.js. For other frameworks, add refresh code. |
| JavaScript Remoting in Visualforce | Restart app | Refresh session and CSRF token from JavaScript | Catch timeout, then either reload page or load a new iFrame. |
| JQuery Mobile | Restart app | Reload page | Catch timeout, then reload page. |

These sections provide detailed coding steps for each architecture.

## REST APIs (Including Apex2REST)

If you're writing or upgrading a hybrid app that leverages REST APIs, detect an expired session and request a new access token at the time the REST call is made. We encourage authors of apps based on this framework to leverage API wrapping libraries, such as forcetk.mobilesdk.js, to manage session retention.

The following code, from `index.html` in the ContactExplorer sample application, demonstrates the recommended technique. When the application first loads, call `getAuthCredentials()` on the Salesforce OAuth plugin, passing the handle to your refresh function (in this case, `salesforceSessionRefreshed`.) The OAuth plugin function calls your refresh function, passing it the session and refresh tokens. Use these returned values to initialize forcetk.mobilesdk.

- From the `onDeviceReady()` function:

```
cordova.require("salesforce/plugin/oauth").getAuthCredentials(salesforceSessionRefreshed,
 getAuthCredentialsError);
```

- `salesforceSessionRefreshed()` function:

```
function salesforceSessionRefreshed(credsData) {
        forcetkClient = new forcetk.Client(credsData.clientId, credsData.loginUrl);
        forcetkClient.setSessionToken(credsData.accessToken, apiVersion,
credsData.instanceUrl);
        forcetkClient.setRefreshToken(credsData.refreshToken);
        forcetkClient.setUserAgentString(credsData.userAgent);
    }
```

For the complete code, see the ContactExplorer sample application (`SalesforceMobileSDK-Android\hybrid\SampleApps\ContactExplorer`).

## JavaScript Remoting in Visualforce

For mobile apps that use JavaScript remoting to access Visualforce pages, incorporate the session refresh code into the method parameter list. In JavaScript, use the Visualforce remote call to check the session state and adjust accordingly.

```
<Controller>.<Method>(
    <params>,
    function(result, event) {
   if (hasSessionExpired(event)) {
     // Reload will try to redirect to login page, container will intercept
     // the redirect and refresh the session before reloading the origin page
     window.location.reload();
   } else {
     // Everything is OK. You can go ahead and use the result.
```

```
  },
    {escape: true}
);
```

This example defines `hasSessionExpired()` as:

```
function hasSessionExpired(event) {
    return (event.type == "exception" && event.message.indexOf("Logged in?") != -1);
}
```

*Advanced developers*: Reloading the entire page might not provide the optimal user experience. If you want to avoid reloading the entire page, you'll need to:

1. Refresh the access token
2. Refresh the Visualforce domain cookies
3. Finally, refresh the CSRF token

In `hasSessionExpired()`, instead of fully reloading the page as follows:

```
window.location.reload();
```

Do something like this:

```
// Refresh oauth token
cordova.require("salesforce/plugin/oauth").authenticate(
  function(creds) {
   // Reload hidden iframe that points to a blank page to
   // to refresh Visualforce domain cookies
   var iframe = document.getElementById("blankIframeId");
   iframe.src = src;

   // Refresh CSRF cookie
   <provider>.refresh(function() {
     <Retry call for a seamless user experience>;
   });

  },
  function(error) {
    console.log("Refresh failed");
  }
);
```

### JQuery Mobile

JQueryMobile makes Ajax calls to transfer data for rendering a page. If a session expires, a 302 error is masked by the framework. To recover, incorporate the following code to force a page refresh.

```
$(document).on('pageloadfailed', function(e, data) {
  console.log('page load failed');
  if (data.xhr.status == 0) {
    // reloading the VF page to initiate authentication
    window.location.reload();
  }
});
```

# Example: Serving the Appropriate Javascript Libraries

To provide the correct version of Javascript libraries, create a separate bundle for each Salesforce Mobile SDK version you use. Then, provide Apex code on the server that downloads the required version.

1. For each Salesforce Mobile SDK version that your application supports, do the following.
   a. Create a ZIP file containing the Javascript libraries from the intended SDK version.
   b. Upload the ZIP file to your org as a static resource.

   For example, if you ship a client that uses Salesforce Mobile SDK v. 1.3, add these files to your ZIP file:

   - `cordova.force.js`
   - `SalesforceOAuthPlugin.js`
   - `bootconfig.js`
   - `cordova-1.8.1.js`, which you should rename as `cordova.js`

   **Note:** In your bundle, it's permissible to rename the Cordova Javascript library as `cordova.js` (or `PhoneGap.js` if you're packaging a version that uses a `PhoneGap-x.x.js` library.)

2. Create an Apex controller that determines which bundle to use. In your controller code, parse the user agent string to find which version the client is using.
   a. In your org, from Setup, click **Develop** > **Apex Class**.
   b. Create a new Apex controller named `SDKLibController` with the following definition.

```
public class SDKLibController {
    public String getSDKLib() {
        String userAgent = ApexPages.currentPage().getHeaders().get('User-Agent');

        if (userAgent.contains('SalesforceMobileSDK/1.3')) {
          return 'sdklib13';
        }
// Add additional if statements for other SalesforceSDK versions
// for which you provide library bundles.
    }
}
```

3. Create a Visualforce page for each library in the bundle, and use that page to redirect the client to that library. For example, for the SalesforceOAuthPlugin library:
   a. In your org, from Setup, click **Develop** > **Pages**.
   b. Create a new page called "SalesforceOAuthPlugin" with the following definition.

```
<apex:page controller="SDKLibController" action="{!URLFor($Resource[SDKLib],
 'SalesforceOAuthPlugin.js')}">
</apex:page>
```

   c. Reference the VisualForce page in a `<script>` tag in your HTML code. Be sure to point to the page you created in step 3b. For example:

```
<script type="text/javascript" src="/apex/SalesforceOAuthPlugin" />
```

   **Note:** Provide a separate `<script>` tag for each library in your bundle.

# Chapter 5

# HTML5 Development

**In this chapter ...**



HTML5 lets you create lightweight mobile interfaces without installing software on the target device. Any mobile, touch or desktop device can access these mobile interfaces.

You can create an HTML5 application that leverages the Force.com platform by:

- Using Visualforce to deliver the HTML content
- Using JavaScript remoting to invoke Apex controllers for fetching records from Force.com

# HTML5 Development Requirements

- You'll need a Force.com organization.
- Some knowledge of Apex and Visualforce is necessary.

**Note:** This type of development uses Visualforce. You can't use Database.com.

# Delivering HTML5 Content With Visualforce

Traditionally, you use Visualforce to create custom websites for the desktop environment. When combined with HTML5, however, Visualforce becomes a viable delivery mechanism for mobile web apps. These apps can leverage third-party UI widget libraries such as Sencha, or templating frameworks such as AngularJS and Backbone.js, that bind to data inside Salesforce.

To set up an HTML5 Apex page, change the `docType` attribute to "html-5.0", and use other settings similar to these:

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

</apex:page>
```

This code sets up an Apex page that can contain HTML5 content, but, of course, it produces an empty page. With the use of static resources and third-party libraries, you can add HTML and JavaScript code to build a fully interactive mobile app.

# Accessing Salesforce Data: Controllers vs. APIs

In an HTML5 app, you can access Salesforce data two ways:

- By using JavaScript remoting to invoke your Apex controller
- By accessing the Salesforce API with forcetk.js

## Using JavaScript Remoting to Invoke Your Apex Controller

Like `apex:actionFunction`, JavaScript remoting lets you invoke methods in your Apex controller through JavaScript code hosted on your Visualforce page.

JavaScript remoting offers several advantages.

- It offers greater flexilibity and better performance than `apex:actionFunction`.
- It supports parameters and return types in the Apex controller method, with automatic mapping between Apex and JavaScript types.
- It uses an asynchronous processing model with callbacks.
- Unlike `apex:actionFunction`, the AJAX request does not include the view state for the Visualforce page. This results in a faster round trip.

Compared to `apex:actionFunction`, however, JavaScript remoting requires you to write more code.

The following example inserts JavaScript code in a <script> tag on the Visualforce page. This code calls the `invokeAction()` method on the Visualforce remoting manager object. It passes `invokeAction()` the metadata needed to call a function named `getItemId()` on the Apex controller object `objName`. Because `invokeAction()` runs asynchronously, the code

also defines a callback function to process the value returned from `getItemId()`. In the Apex controller, the `@RemoteAction` annotation exposes the `getItemId()` function to external JavaScript code.

```
//Visualforce page code
<script type="text/javascript">
     Visualforce.remoting.Manager.invokeAction(
        '{!$RemoteAction.MyController.getItemId}',
        objName,
        function(result, event){
            //process response here
        },
        {escape: true}
    );
<script>

//Apex Controller code

@RemoteAction
global static String getItemId(String objectName) { ... }
```

See this Dreamforce 2012 session for a more detailed comparison between the JavaScript remoting and `actionFunction`. See http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_classes_annotation_RemoteAction.htm to read more about `@RemoteAction` annotations.

## Accessing the Salesforce API with Forcetk and JQuery

When you call Salesforce REST APIs from Visualforce, you're calling to a different domain. This separation violates same-origin browser policy, which causes the browser to refuse the connection. The forcetk JavaScript library works around same-origin policy restrictions by using the AJAX Proxy to give full access to the REST API. Since the AJAX proxy is present on all Visualforce hosts with an endpoint of the form https://<*abc*>.na1.visual.force.com/services/proxy, your Visualforce-hosted JavaScript can invoke it by passing the desired resource URL in an HTTP header.

To use the proxy service:

1. Send your request to `https://<domain>/services/proxy`, where <*domain*> is the domain of your current Visualforce page.
2. Use the following HTTP headers:

   **SalesforceProxy-Endpoint**

   URL of the request endpoint

   **SalesforceProxy-SID**

   Current user session ID

   For tips on accessing this proxy through JavaScript, see AJAX Proxy.

The following code sample uses the jQuery Mobile library for the user interface. To run this code, your Visualforce page must include jQuery and the forcetk toolkit. To add these resources:

1. Create an archive file, such as a ZIP file, that contains `app.js`, `forcetk.js`, `jquery.js`, and any other static resources your project requires.
2. In Salesforce, upload the archive file via **Your Name** > **App Setup** > **Develop** > **Static Resources**.

After obtaining an instance of the jQuery Mobile library, the sample code creates a forcetk client object and initializes it with a session ID. It then calls the asynchronous forcetk `query()` method to process a SOQL query. The query callback function uses jQuery Mobile to display the first Name field returned by the query as HTML in an object with ID "accountname." At the end of the Apex page, the HTML5 content defines the `accountname` element as a simple `<span>` tag.

```
<apex:page>
 <apex:includeScript value="{!URLFOR($Resource.static, 'jquery.js')}" />
 <apex:includeScript value="{!URLFOR($Resource.static, 'forcetk.js')}"  />
```

```
<script type="text/javascript">
    // Get a reference to jQuery that we can work with
    $j = jQuery.noConflict();

    // Get an instance of the REST API client and set the session ID
    var client = new forcetk.Client();
    client.setSessionToken('{!$Api.Session_ID}');

    client.query("SELECT Name FROM Account LIMIT 1", function(response){
        $j('#accountname').html(response.records[0].Name);
    });
</script>
<p>The first account I see is <span id="accountname"></span>.</p>
</apex:page>
```

**Note:**

- Using the REST API—even from a Visualforce page—consumes API calls.
- SalesforceAPI calls made through a Mobile SDK container or through a Cordova webview do not require proxy services. Cordova webviews disable same-origin policy, so you can make API calls directly. This exemption applies to all Mobile SDK hybrid and native apps.

## Additional Options

You can use the SmartSync Data Framework in HTML5 apps. Just include the required JavaScript libraries as static resources. Take advantage of the model and routing features. Offline access is disabled for this use case. See Using SmartSync to Access Salesforce Objects on page 63.

Salesforce Developer Marketing provides developer mobile packs that can help you get a quick start with HTML5 apps.

## Offline Limitations

Read these articles for tips on using HTML5 with Force.com in offline situations.

- http://blogs.developerforce.com/developer-relations/2011/06/using-html5-offline-with-forcecom.html
- http://blogs.developerforce.com/developer-relations/2013/03/using-javascript-with-force-com.html

# Chapter 6

# Using SmartSync to Access Salesforce Objects

**In this chapter ...**



The SmartSync Data Framework is a Mobile SDK library that represents Salesforce objects as JavaScript objects. Using SmartSync in a hybrid app, you can create models of Salesforce objects and manipulate the underlying records just by changing the model data. If you perform a SOQL or SOSL query, you receive the resulting records in a model collection rather than as a JSON string.

Underlying the SmartSync technology is the `backbone.js` open-source JavaScript library. `Backbone.js` defines an extensible mechanism for modeling data. To understand the basic technology behind the SmartSync Data Framework, browse the examples and documentation at backbonejs.org.

Three sample hybrid applications demonstrate SmartSync.

- Account Editor (`AccountEditor.html`)
- User Search (`UserSearch.html`)
- User and Group Search (`UserAndGroupSearch.html`)

You can find these sample apps in the `./hybrid/SampleApps/AccountEditor/assets/www` folder.

# About Backbone Technology

The SmartSync library, `SmartSync.js`, provides extensions to the open-source Backbone JavaScript library. The Backbone library defines key building blocks for structuring your web application:

- Models with key-value binding and custom events, for modeling your information
- Collections with a rich API of enumerable functions, for containing your data sets
- Views with declarative event handling, for displaying information in your models
- A router for controlling navigation between views

Salesforce SmartSync Data Framework extends the `Model` and `Collection` core Backbone objects to connect them to the Salesforce REST API. SmartSync also provides optional offline support through SmartStore, the secure storage component of the Mobile SDK.

To learn more about Backbone, see http://backbonejs.org/ and http://backbonetutorials.com/. You can also search online for "backbone javascript" to find a wealth of tutorials and videos.

# Models and Model Collections

Two types of objects make up the SmartSync Data Framework:

- Models
- Model collections

Definitions for these objects extend classes defined in `backbone.js`, a popular third-party JavaScript framework. For background information, see http://backbonetutorials.com.

# Models

Models on the client represent server records. In SmartSync, model objects are instances of `Force.SObject`, a subclass of the `Backbone.Model` class. `SObject` extends `Model` to work with Salesforce APIs and, optionally, with SmartStore.

You can perform the following CRUD operations on `SObject` model objects:

- Create
- Destroy
- Fetch
- Save
- Get/set attributes

In addition, model objects are observable: Views and controllers can receive notifications when the objects change.

## Properties

`Force.SObject` adds the following properties to `Backbone.Model`:

**`sobjectType`**

Required. The name of the Salesforce object that this model represents. This value can refer to either a standard object or a custom object.

**`fieldlist`**

Required. Names of fields to fetch, save, or destroy.

**`cacheMode`**

Offline behavior.

**`mergeMode`**

Conflict handling behavior.

**`cache`**

For updatable offline storage of records. The SmartSync Data Framework comes bundled with Force.StoreCache, a cache implementation that is backed by SmartStore.

**`cacheForOriginals`**

Contains original copies of records fetched from server to support conflict detection.

### Examples

You can assign values for model properties in several ways:

- As properties on a `Force.SObject` instance.
- As methods on a `Force.SObject` sub-class. These methods take a parameter that specifies the desired CRUD action ("create", "read", "update", or "delete").
- In the options parameter of the `fetch()`, `save()`, or `destroy()` function call.

For example, these code snippets are equivalent.

```
// As properties on a Force.SObject instance
acc = new Force.SObject({Id:"<some_id>"});
acc.sobjectType = "account";
acc.fieldlist = ["Id", "Name"];
acc.fetch();
```

```
// As methods on a Force.SObject sub-class
Account = Force.SObject.extend({
  sobjectType: "account",
  fieldlist: function(method) { return ["Id", "Name"];}
});
Acc = new Account({Id:"<some_id>"});
acc.fetch();
```

```
// In the options parameter of fetch()
acc = new Force.SObject({Id:"<some_id>"});
acc.sobjectType = "account";
acc.fetch({fieldlist:["Id", "Name"]);
```

## Model Collections

Model collections in the SmartSync Data Framework are containers for query results. Query results stored in a model collection can come from the server via SOQL, SOSL, or MRU queries. Optionally, they can also come from the cache via SmartSQL (if the cache is SmartStore), or another query mechanism if you use an alternate cache.

Model collection objects are instances of `Force.SObjectCollection`, a subclass of the `Backbone.Collection` class. `SObjectCollection` extends `Collection` to work with Salesforce APIs and, optionally, with SmartStore.

### Properties

`Force.SObjectCollection` adds the following properties to `Backbone.Collection`:

**`config`**

Required. Defines the records the collection can hold (using SOQL, SOSL, MRU or SmartSQL).

**`cache`**

For updatable offline storage of records. The SmartSync Data Framework comes bundled with Force.StoreCache, a cache implementation that's backed by SmartStore.

**`cacheForOriginals`**

Contains original copies of records fetched from server to support conflict detection.

### Examples

You can assign values for model collection properties in several ways:

- As properties on a `Force.SObject` instance
- As methods on a `Force.SObject` sub-class
- In the options parameter of the `fetch()`, `save()`, or `destroy()` function call

For example, these code snippets are equivalent.

```
// As properties on a Force.SObject instance
list = new Force.SObjectCollection({config:<valid_config>});
list.fetch();
```

```
// As methods on a Force.SObject sub-class
MyCollection = Force.SObjectCollection.extend({
  config: function() { return <valid_config>; }
});
list = new MyCollection();
list.fetch();
```

```
// In the options parameter of fetch()
list = new Force.SObjectCollection();
list.fetch({config:valid_config});
```

# Using the SmartSync Data Framework in JavaScript

To use SmartSync in a hybrid app, include:

- `jquery-x.x.x.min.js` (use version of file in `external/shared/jquery/`)
- `underscore-x.x.x.min.js` (use version of file in `external/shared/backbone/`)
- `backbone-x.x.x.min.js` (use version of file in `external/shared/backbone/`)
- `cordova.js`
- `cordova.force.js`
- `forcetk.mobilesdk.js`
- `SmartSync.js`

### Implementing a Model Object

To begin using SmartSync objects, define a model object to represent each `SObject` that you want to manipulate. The `SObjects` can be standard Salesforce objects or custom objects. For example, this code creates a model of the Account object that sets the two required properties—`sobjectType` and `fieldlist`—and defines a `cacheMode()` function.

```
app.models.Account = Force.SObject.extend({
    sobjectType: "Account",
```

```
    fieldlist: ["Id", "Name", "Industry", "Phone"],

    cacheMode: function(method) {
        if (app.offlineTracker.get("offlineStatus") == "offline") {
            return "cache-only";
        }
        else {
            return (method == "read" ? "cache-first" : "server-first");
        }
    }
});
```

Notice that the `app.models.Account` model object extends `Force.SObject`, which is defined in `SmartSync.js`. Also, the `cacheMode()` function queries a local `offlineTracker` object for the device's offline status. You can use the Cordova library to determine offline status at any particular moment.

SmartSync can perform a fetch or a save operation on the model. It uses the app's `cacheMode` value to determine whether to perform an operation on the server or in the cache. Your `cacheMode` member can either be a simple string property or a function returning a string.

## Implementing a Model Collection

The model collection for this sample app extends `Force.SObjectCollection`.

```
// The AccountCollection Model
app.models.AccountCollection = Force.SObjectCollection.extend({
    model: app.models.Account,
    fieldlist: ["Id", "Name", "Industry", "Phone"],

    setCriteria: function(key) {
        this.key = key;
    },

    config: function() {
        // Offline: do a cache query
        if (app.offlineTracker.get("offlineStatus") == "offline") {
            return {type:"cache", cacheQuery:{queryType:"like",
                indexPath:"Name", likeKey: this.key+"%",
                order:"ascending"}};
        }
        // Online
        else {
            // First time: do a MRU query
            if (this.key == null) {
                return {type:"mru", sobjectType:"Account",
                    fieldlist: this.fieldlist};
            }
            // Other times: do a SOQL query
            else {
                var soql = "SELECT " + this.fieldlist.join(",")
                    + " FROM Account"
                    + " WHERE Name like '" + this.key + "%'";
                return {type:"soql", query:soql};
            }
        }
    }
});
```

This model collection uses an optional key that is the name of the account to be fetched from the collection. It also defines a `config()` function that determines what information is fetched. If the device is offline, the `config()` function builds a cache query statement. Otherwise, if no key is specified, it queries the most recently used record ("mru"). If the key is specified and the device is online, it builds a standard SOQL query that pulls records for which the name matches the key. The fetch operation on the `Force.SObjectCollection` prototype transparently uses the returned configuration to automatically fill the model collection with query records.

See querySpec for information on formatting a cache query.

**Note:** These code examples are part of the Account Editor sample app. See Account Editor Sample for a sample description.

# Offline Caching

To provide offline support, your app must be able to cache its models and collections. SmartSync provides a configurable mechanism that gives you full control over caching operations.

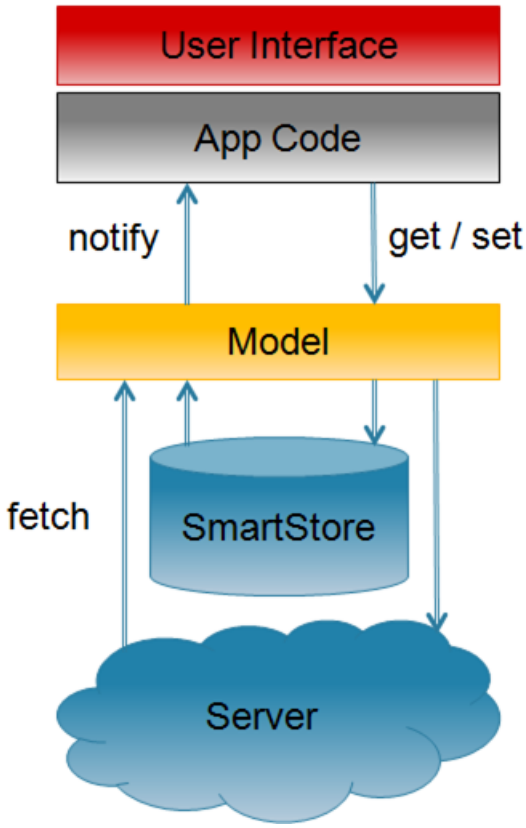
## Default Cache and Custom Cache Implementations

For its default cache, the SmartSync library defines StoreCache, a cache implementation that uses SmartStore. Both StoreCache and SmartStore are optional components for SmartSync apps. If your application runs in a browser instead of the Mobile SDK container, or if you don't want to use SmartStore, you must provide an alternate cache implementation. SmartSync requires cache objects to support these operations:

- retrieve
- save
- save all
- remove
- find

## SmartSync Caching Workflow

The SmartSync model performs all interactions with the cache and the Salesforce server on behalf of your app. Your app gets and sets attributes on model objects. During save operations, the model uses these attribute settings to determine whether to write changes to the cache or server, and how to merge new data with existing data. If anything changes in the underlying data or in the model itself, the model sends event notifications. Similarly, if you request a fetch, the model fetches the data and presents it to your app in a model collection.




SmartSync updates data in the cache transparently during CRUD operations. You can control the transparency level through optional flags. Cached objects maintain "dirty" attributes that indicate whether they've been created, updated, or deleted locally.

## Cache Modes

When you use a cache, you can specify a mode for each CRUD operation. Supported modes are:

| Mode | Constant | Description |
|---|---|---|
| “cache-only” | `Force.CACHE_MODE.CACHE_ONLY` | Read from, or write to, the cache. Do not perform the operation on the server. |
| “server-only” | `Force.CACHE_MODE.SERVER_ONLY` | Read from, or write to, the server. Do not perform the operation on the cache. |
| “cache-first” | `Force.CACHE_MODE.CACHE_FIRST` | For FETCH operations only. Fetch the record from the cache. If the cache doesn't contain the record, fetch it from the server and then update the cache. |
| “server-first” (default) | `Force.CACHE_MODE.SERVER_FIRST` | Perform the operation on the server, then update the cache. |

To query the cache directly, use a cache query. SmartStore provides query APIs as well as its own query language, Smart SQL. See Retrieving Data From a Soup.

# Implementing Offline Caching

To support offline caching, SmartSync requires you to supply your own implementations of a few tasks:

- Tracking offline status and specifying the appropriate cache control flag for CRUD operations, as shown in the `app.models.Account` example.
- Collecting records that were edited locally and saving their changes to the server when the device is back online. The following example uses a SmartStore cache query to retrieve locally changed records, then calls the `SyncPage` function to render the results in HTML.

```
sync: function() {
 var that = this;
 var localAccounts = new app.models.AccountCollection();
 localAccounts.fetch({
  config: {type:"cache", cacheQuery: {queryType:"exact",
       indexPath:"__local__", matchKey:true}},
  success: function(data) {
   that.slidePage(new app.views.SyncPage({model: data}).render());
  }
 });
}

app.views.SyncPage = Backbone.View.extend({

    template: _.template($("#sync-page").html()),

    render: function(eventName) {
        $(this.el).html(this.template(_.extend(
            {countLocallyModified: this.model.length},
            this.model.toJSON())));
        this.listView = new app.views.AccountListView({el: $("ul",
            this.el), model: this.model});
        this.listView.render();
        return this;
    },
...
});
```

# Using StoreCache For Offline Caching

The `SmartSync.js` library implements a cache named StoreCache that stores its data in SmartStore. Although SmartSync uses StoreCache as its default cache, StoreCache is a stand-alone component. Even if you don't use SmartSync, you can still leverage StoreCache for SmartStore operations.

**Note:** Although StoreCache is intended for use with SmartSync, you can use any cache mechanism with SmartSync that meets the requirements described in Offline Caching.

## Construction and Initialization

StoreCache objects work internally with SmartStore soups. To create a StoreCache object backed by the soup `soupName`, use the following constructor:

```
new Force.StoreCache(soupName [, additionalIndexSpecs, keyField])
```

**soupName**

Required. The name of the underlying SmartStore soup.

**additionalIndexSpecs**

Fields to include in the cache index in addition to default index fields. See Registering a Soup for formatting instructions.

**keyField**

Name of field containing the record ID. If not specified, StoreCache expects to find the ID in a field named "Id."

Soup items in a StoreCache object include four additional boolean fields for tracking offline edits:

- `__locally_created__`
- `__locally_updated__`
- `__locally_deleted__`
- `__local__` (set to true if any of the previous three are true)

These fields are for internal use but can also be used by apps. StoreCache indexes each soup on the `__local__` field and its ID field. You can use the `additionalIndexSpecs` parameter to specify additional fields to include in the index.

To register the underlying soup, call `init()` on the StoreCache object. This function returns a jQuery promise that resolves once soup registration is complete.

## StoreCache Methods

**init()**

Registers the underlying SmartStore soup. Returns a jQuery promise that resolves when soup registration is complete.

**retrieve(key [, fieldlist])**

Returns a jQuery promise that resolves to the record with key in the keyField returned by the SmartStore. The promise resolves to null when no record is found or when the found record does not include all the fields in the fieldlist parameter.

**key**

The key value of the record to be retrieved.

**fieldlist**

(Optional) A JavaScript array of required fields. For example:

```
["field1","field2","field3"]
```

**save(record [, noMerge])**

Returns a jQuery promise that resolves to the saved record once the SmartStore upsert completes. If `noMerge` is not specified or is false, the passed record is merged with the server record with the same key, if one exists.

**record**

The record to be saved, formatted as:

```
{<field_name1>:"<field_value1>"[,<field_name2>:"<field_value2>",...]}
```

For example:

```
{Id:"007", Name:"JamesBond", Mission:"TopSecret"}
```

**noMerge**

(Optional) Boolean value indicating whether the passed record is to be merged with the matching server record. Defaults to false.

**saveAll(records [, noMerge])**

Identical to `save()`, except that `records` is an array of records to be saved. Returns a jQuery promise that resolves to the saved records.

**records**

An array of records. Each item in the array is formatted as demonstrated for the `save()` function.

**noMerge**

(Optional) Boolean value indicating whether the passed record is to be merged with the matching server record. Defaults to false.

**remove(key)**

Returns a jQuery promise that resolves when the record with the given key has been removed from the SmartStore.

**key**

Key value of the record to be removed.

**find(querySpec)**

Returns a jQuery promise that resolves once the query has been run against the SmartStore. The resolved value is an object with the following fields:

| Field | Description |
|---|---|
| `records` | All fetched records |
| `hasMore` | Function to check if more records can be retrieved |
| `getMore` | Function to fetch more records |
| `closeCursor` | Function to close the open cursor and disable further fetch |

**querySpec**

A specification based on SmartStore query function calls, formatted as:

```
{queryType: "like" | "exact" | "range" | "smart"[, query_type_params]}
```

where `query_type_params` match the format of the related SmartStore query function call. See Retrieving Data From a Soup on page 106.

Here are some examples:

```
{queryType:"exact", indexPath:"<indexed_field_to_match_on>",
matchKey:<value_to_match>, order:"ascending"|"descending",
pageSize:<entries_per_page>}

{queryType:"range", indexPath:"<indexed_field_to_match_on>",
beginKey:<start_of_Range>, endKey:<end_of_range>, order:"ascending"|"descending",
 pageSize:<entries_per_page>}

{queryType:"like", indexPath:"<indexed_field_to_match_on>",
likeKey:"<value_to_match>", order:"ascending"|"descending",
pageSize:<entries_per_page>}
```

```
{queryType:"smart", smartSql:"<smart_sql_query>", order:"ascending"|"descending",
 pageSize:<entries_per_page>}
```

### Examples

The following example shows how to create, initialize, and use a StoreCache object.

```
var cache = new Force.StoreCache("agents", [{path:"Mission", type:"string"} ]);
// initialization of the cache / underlying soup
cache.init()
.then(function() {
    // saving a record to the cache
    return cache.save({Id:"007", Name:"JamesBond", Mission:"TopSecret"});
})
.then(function(savedRecord) {
    // retrieving a record from the cache
    return cache.retrieve("007");
})
.then(function(retrievedRecord) {
    // searching for records in the cache
    return cache.find({queryType:"like", indexPath:"Mission", likeKey:"Top%",
order:"ascending", pageSize:1});
})
.then(function(result) {
    // removing a record from the cache
    return cache.remove("007");
});
```

The next example shows how to use the `saveAll()` function and the results of the `find()` function.

```
// initialization
var cache = new Force.StoreCache("agents", [ {path:"Name", type:"string"}, {path:"Mission",
 type:"string"} ]);
cache.init()
.then(function() {
    // saving some records
   return cache.saveAll([{Id:"007", Name:"JamesBond"},{Id:"008", Name:"Agent008"}, {Id:"009",
 Name:"JamesOther"}]);
})
.then(function() {
    // doing an exact query
    return cache.find({queryType:"exact", indexPath:"Name", matchKey:"Agent008",
order:"ascending", pageSize:1});
})
.then(function(result) {
    alert("Agent mission is:" + result.records[0]["Mission"];
});
```

# Conflict Detection

Model objects support optional conflict detection to prevent unwanted data loss when the object is saved to the server. You can use conflict detection with any save operation, regardless of whether the device is returning from an offline state.

To support conflict detection, you specify a secondary cache to contain the original values fetched from the server. SmartSync keeps this cache for later reference. When you save or delete, you specify a *merge mode*. The following table summarizes the supported modes. To understand the mode descriptions, consider "theirs" to be the current server record, "yours" the current local record, and "base" the record that was originally fetched from the server.

| Mode | Constant | Description |
|---|---|---|
| “overwrite” | `Force.MERGE_MODE.OVERWRITE` | Write "yours" to the server, without comparing to "theirs" or "base”. (This is the same as not using conflict detection.) |
| “merge-accept-yours” | `Force.MERGE_MODE.MERGE_ACCEPT_YOURS` | Merge "theirs" and "yours". If the same field is changed both locally and remotely, the local value is kept. |
| “merge-fail-if-conflict” | `Force.MERGE_MODE.MERGE_FAIL_IF_CONFLICT` | Merge "theirs" and "yours". If the same field is changed both locally and remotely, the operation fails. |
| “merge-fail-if-changed” | `Force.MERGE_MODE.MERGE_FAIL_IF_CHANGED` | Merge "theirs" and "yours". If any field is changed remotely, the operation fails. |

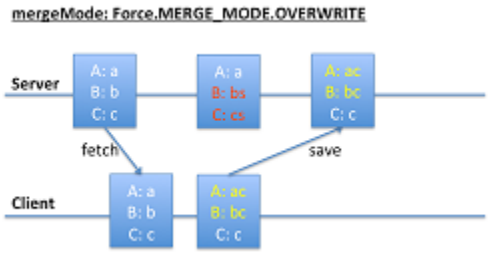
If a save or delete operation fails, you receive a report object with the following fields:

| Field Name | Contains |
|---|---|
| `base` | Originally fetched attributes |
| `theirs` | Latest server attributes |
| `yours` | Locally modified attributes |
| `remoteChanges` | List of fields changed between base and theirs |
| `localChanges` | List of fields changed between base and yours |
| `conflictingChanges` | List of fields changed both in theirs and yours, with different values |

Diagrams can help clarify how merge modes operate.

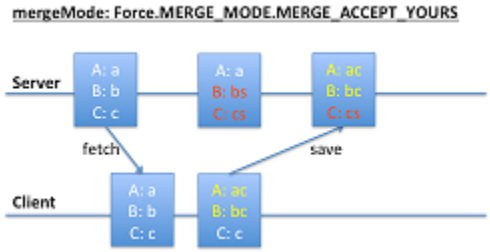
## MERGE_MODE.OVERWRITE

In the `MERGE_MODE.OVERWRITE` diagram, the client changes A and B, and the server changes B and C. Changes to B conflict, whereas changes to A and C do not. However, the save operation blindly writes all the client’s values to the server, overwriting any changes on the server.



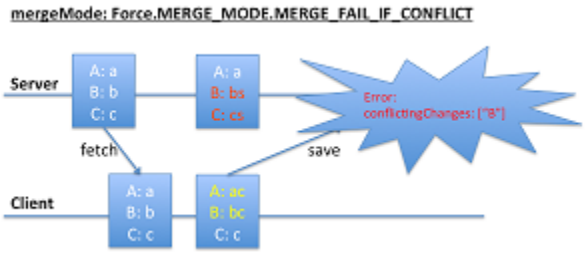

## MERGE_ACCEPT_YOURS

In the `MERGE_MODE.MERGE_ACCEPT_YOURS` diagram, the client changes A and B, and the server changes B and C. Client changes (A and B) overwrites corresponding fields on the server, regardless of whether conflicts exist. However, fields that the client leaves unchanged (C) do not overwrite corresponding server values.



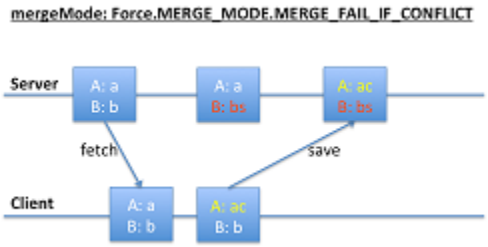

## MERGE_FAIL_IF_CONFLICT (Fails)

In the first `MERGE_MODE.MERGE_FAIL_IF_CONFLICT` diagram, both the client and the server change B. These conflicting changes cause the save operation to fail.




## MERGE_FAIL_IF_CONFLICT (Succeeds)

In the second `MERGE_MODE.MERGE_FAIL_IF_CONFLICT` diagram, the client changed A, and the server changed B. These changes don't conflict, so the save operation succeeds.



# Mini-Tutorial: Conflict Detection

The following mini-tutorial demonstrates how merge modes affect save operations under various circumstances. It takes the form of an extended example within an HTML context.

1. Set up the necessary caches:

```
var cache = new Force.StoreCache(soupName);
var cacheForOriginals = new Force.StoreCache(soupNameForOriginals);
var Account = Force.SObject.extend({sobjectType:"Account", fieldlist:["Id", "Name",
"Industry"], cache:cache, cacheForOriginals:cacheForOriginals});
```

2. Get an existing account:

```
var account = new Account({Id:<some actual account id>});
account.fetch();
```

3. Let's assume that the account has Name:"Acme" and Industry:"Software". Change the name to “Acme2.”

```
Account.set("Name", "Acme2");
```

4. Save to the server without specifying a merge mode, so that the default "overwrite" merge mode is used:

```
account.save(null);
```

   The account’s Name is now "Acme2" and its Industry is "Software" Let's assume that Industry changes on the server to "Electronics."

5. Change the account Name again:

```
Account.set("Name", "Acme3");
```

   You now have a change in the cache (Name) and a change on the server (Industry).

6. Save again, using "merge-fail-if-changed" merge mode.

```
account.save(null, {mergeMode: "merge-fail-if-changed", error: function(err) {
  // err will be a map of the form {base:…, theirs:…, yours:…, remoteChanges:["Industry"],
 localChanges:["Name"], conflictingChanges:[]}
});
```

   The error callback is called because the server record has changed.

7. Save again, using "merge-fail-if-conflict" merge mode. This merge succeeds because no conflict exists between the change on the server and the change on the client.

```
account.save(null, {mergeMode: "merge-fail-if-conflict"});
```

   The account’s Name is now "Acme3" (yours) and its Industry is "Electronics" (theirs). Let's assume that, meanwhile, Name on the server changes to "NewAcme" and Industry changes to "Services."

8. Change the account Name again:

```
Account.set("Name", "Acme4");
```

9. Save again, using "merge-fail-if-changed" merge mode. The error callback is called because the server record has changed.

```
account.save(null, {mergeMode: "merge-fail-if-changed", error: function(err) {
   // err will be a map of the form {base:…, theirs:…, yours:…, remoteChanges:["Name",
"Industry"], localChanges:["Name"], conflictingChanges:["Name"]}
});
```

10. Save again, using "merge-fail-if-conflict" merge mode:

```
account.save(null, {mergeMode: "merge-fail-if-changed", error: function(err) {
   // err will be a map of the form {base:…, theirs:…, yours:…, remoteChanges:["Name",
"Industry"], localChanges:["Name"], conflictingChanges:["Name"]}
});
```

The error callback is called because both the server and the cache change the Name field, resulting in a conflict:

11. Save again, using "merge-accept-yours" merge mode. This merge succeeds because your merge mode tells the `save()` function which Name value to accept. Also, since you haven't changed Industry, that field doesn't conflict.

```
account.save(null, {mergeMode: "merge-accept-yours"});
```

Name is "Acme4" (yours) and Industry is "Services" (theirs), both in the cache and on the server.

# Tutorial: Creating a SmartSync Application

This tutorial demonstrates how to create a local hybrid app that uses the SmartSync Data Framework. It recreates the User Search sample application that ships with Mobile SDK 2.0. User Search lets you search for User records in a Salesforce organization and see basic details about them.

This sample uses the following web technologies:

- Backbone.js
- Ratchet
- HTML5
- JavaScript

## Set Up Your Project

First, make sure you've installed Salesforce Mobile SDK using the NPM installer. For iOS instructions, see Installing and Uninstalling Salesforce Mobile SDK for iOS on page 9. For Android instructions, see Installing and Uninstalling Salesforce Mobile SDK for Android on page 28.

Also, download the `ratchet.css` file from http://maker.github.io/ratchet/.

1. Once you've installed Mobile SDK, create a local hybrid project for your platform.
   a. For **iOS:** At the command terminal, enter the following command:

```
forceios create --apptype=hybrid_local --appname=UserSearch
--companyid=com.acme.UserSearch --organization=Acme --outputdir=.
```

The forceios script creates your project at `./UserSearch/UserSearch.xcode.proj`.

**b.** For **Android:** At the command terminal or the Windows command prompt, enter the following command:

```
forcedroid create -–apptype="hybrid_local" --appname="UserSearch" --targetdir=.
--packagename="com.acme.usersearch"
```

The forcedroid script creates the project at `./UserSearch`.

2. Follow the onscreen instructions to open the new project in Eclipse (for Android) or Xcode (for iOS).
3. Open the `www` folder.
4. Remove the `inline.js` file from the project.
5. Create a new folder. Name it `css`.
6. Copy the `ratchet.css` file into your new `css` folder.
7. In the `www` folder, open `index.html` in your code editor and delete all of its contents.

## Edit the Application HTML File

To create your app's basic structure, define an empty HTML page that contains references, links, and code infrastructure.

1. In Xcode, edit `index.html` and add the following basic structure:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

2. In the `<head>` element:
   **a.** Turn off scaling to make the page look like an app rather than a web page.

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

   **b.** Set the content type.

```
<meta http-equiv="Content-type" content="text/html; charset=utf-8">
```

   **c.** Add a link to the ratchet.css file to provide the mobile look:

```
<link rel="stylesheet" href="css/ratchet.css"/>
```

   **d.** Include the necessary JavaScript files.

```
<script src="jquery/jquery-2.0.0.min.js"></script>
<script src="backbone/underscore-1.4.4.min.js"></script>
<script src="backbone/backbone-1.0.0.min.js"></script>
<script src="cordova-2.3.0.js"></script>
<script src="forcetk.mobilesdk.js"></script>
<script src="cordova.force.js"></script>
<script src="SmartSync.js"></script>
```

3. Now let's start adding content to the body. In the `<body>` block, add a `div` tag to contain the app UI.

```
<body>
<div id="content"></div>
```

   It's good practice to keep your objects and classes in a namespace. In this sample, we use the `app` namespace to contain our models and views.

4. In a `<script>` tag, create an application namespace. Let's call it `app`.

```
<script>
var app = {
    models: {},
    views: {}
}
```

   For the remainder of this procedure, continue adding your code in the `<script>` block.

5. Add an event listener and handler to wait for jQuery, and then call Cordova to start the authentication flow. Also, specify a callback function, `appStart`, to handle the user's credentials.

```
jQuery(document).ready(function() {
    document.addEventListener("deviceready", onDeviceReady,false);
});

function onDeviceReady() {
    cordova.require("salesforce/plugin/oauth").getAuthCredentials(appStart);
}
```

   Once the application has initialized and authentication is complete, the Salesforce OAuth plugin calls `appStart()` and passes it the user's credentials. The `appStart()` function passes the credentials to SmartSync by calling `Force.init()`, which initializes SmartSync. The `appStart()` function also creates a Backbone `Router` object for the application.

6. Add the `appStart()` function definition at the end of the `<script>` block.

```
function appStart(creds) {
    Force.init(creds, null, null,
       cordova.require("salesforce/plugin/oauth").forcetkRefresh);
    app.router = new app.Router();
    Backbone.history.start();
}
```

Here's the complete application to this point.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0,
      maximum-scale=1.0; user-scalable=no" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="css/ratchet.css"/>
    <script src="jquery/jquery-2.0.0.min.js"></script>
    <script src="backbone/underscore-1.4.4.min.js"></script>
    <script src="backbone/backbone-1.0.0.min.js"></script>
    <script src="cordova-2.3.0.js"></script>
    <script src="forcetk.mobilesdk.js"></script>
    <script src="cordova.force.js"></script>
    <script src="SmartSync.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script id="search-page" type="text/template">
      <header class="bar-title">
```

```
        <h1 class="title">Users</h1>
      </header>

      <div class="bar-standard bar-header-secondary">
        <input type="search" class="search-key" placeholder="Search"/>
      </div>

      <div class="content">
        <ul class="list"></ul>
      </div>
    </script>


    <script id="user-list-item" type="text/template">
      <img src="<%= SmallPhotoUrl %>" class="small-img" />
      <div class="details-short">
        <b><%= FirstName %> <%= LastName %></b><br/>
        Title<%= Title %>
      </div>
    </script>

  <script>
  var app = {
      models: {},
      views: {}
  };

  jQuery(document).ready(function() {
    document.addEventListener("deviceready", onDeviceReady,false);
  });

  function onDeviceReady() {
    cordova.require("salesforce/plugin/oauth").getAuthCredentials(appStart);
  }

  function appStart(creds) {
      console.log(JSON.stringify(creds));
      Force.init(creds, null, null,
cordova.require("salesforce/plugin/oauth").forcetkRefresh);
      app.router = new app.Router();
      Backbone.history.start();
  }    </script>
  </body>
</html>
```

## Create a SmartSync Model and a Collection

Now that we've configured the HTML infrastructure, let's get started using SmartSync by extending two of its primary objects:

- `Force.SObject`
- `Force.SObjectCollection`

These objects extend `Backbone.Model`, so they support the `Backbone.Model.extend()` function. To extend an object using this function, pass it a JavaScript object containing your custom properties and functions.

1. In the `<body>` tag, create a model object for the Salesforce User sObject. Extend `Force.SObject` to specify the sObject type and the fields we are targeting.

```
app.models.User = Force.SObject.extend({
    sobjectType: "User",
    fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title", "Email",
        "MobilePhone","City"]
})
```

2. Immediately after setting the User object, create a collection to hold user search results.
   Extend`Force.SObjectCollection` to indicate your new model (`app.models.User`) as the model for items in the collection.

```
app.models.UserCollection = Force.SObjectCollection.extend({
    model: app.models.User
});
```

Here's the complete model code.

```
// Models
app.models.User = Force.SObject.extend({
    sobjectType: "User",
    fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title", "Email",
"MobilePhone","City"]
});

app.models.UserCollection = Force.SObjectCollection.extend({
    model: app.models.User
});
```

## Create a Template

Templates let you describe an HTML layout within another HTML page. You can define an inline template in your HTML page by using a `<script>` tag of type "text/template". Your JavaScript code can use the template as the page design when it instantiates a new HTML page at runtime.

The search page template is simple. It includes a header, a search field, and a list to hold the search results.

1. Add a new script block. Place the block within the `<body>` block just after the "content" `<div>` tag.

```
<script id="search-page" type="text/template">
</script>
```

2. In the new `<script>` block, define the search page HTML template using Ratchet styles.

```
<script id="search-page" type="text/template">
  <header class="bar-title">
    <h1 class="title">Users</h1>
  </header>

  <div class="bar-standard bar-header-secondary">
    <input type="search" class="search-key" placeholder="Search"/>
  </div>

  <div class="content">
    <ul class="list"></ul>
  </div>
</script>
```

## Add the Search View

To create the view for a screen, you extend `Backbone.View`. In the search view extension, you load the template, define sub-views and event handlers, and implement the functionality for rendering the views and performing a SOQL search query.

1. In the `<body>` block, create a `Backbone.View` extension named `SearchPage` in the `app.views` array.

```
app.views.SearchPage = Backbone.View.extend({
});
```

   For the remainder of this procedure, add all code to the `extend({})` block.

2. Load the search-page template by calling the `_.template()` function. Pass it the raw HTML content of the `search-page` script tag.

```
template: _.template($("#search-page").html()),
```

3. Instantiate a sub-view named `UserListView` to contain the list of search results. (You'll define the `app.views.UserListView` view later.)

```
initialize: function() {
    this.listView = new app.views.UserListView({model: this.model});
},
```

4. Create a `render()` function for the search page view. Rendering the view consists simply of loading the template as the app's HTML content. Restore any criteria previously typed in the search field and render the sub-view inside the `<ul>` element.

```
render: function(eventName) {
    $(this.el).html(this.template());
    $(".search-key", this.el).val(this.model.criteria);
    this.listView.setElement($("ul", this.el)).render();
    return this;
},
```

5. Add a `keyup` event handler that performs a search when the user types a character in the search field.

```
events: {
    "keyup .search-key": "search"
},

search: function(event) {
    this.model.criteria = $(".search-key", this.el).val();
   var soql = "SELECT Id, FirstName, LastName, SmallPhotoUrl, Title FROM User WHERE Name
 like '" + this.model.criteria + "%' ORDER BY Name LIMIT 25 ";
    this.model.fetch({config: {type:"soql", query:soql}});
}
```

   This function defines a SOQL query. It then uses the backing model to send that query to the server and fetch the results.

Here's the complete extension.

```
app.views.SearchPage = Backbone.View.extend({
    template: _.template($("#search-page").html()),

    initialize: function() {
        this.listView = new app.views.UserListView({model: this.model});
    },

    render: function(eventName) {
        $(this.el).html(this.template());
        $(".search-key", this.el).val(this.model.criteria);
        this.listView.setElement($("ul", this.el)).render();
        return this;
    },
```

```
    events: {
        "keyup .search-key": "search"
    },

    search: function(event) {
        this.model.criteria = $(".search-key", this.el).val();
        var soql = "SELECT Id, FirstName, LastName, SmallPhotoUrl, Title FROM User WHERE
Name like '" + this.model.criteria + "%' ORDER BY Name LIMIT 25 ";
        this.model.fetch({config: {type:"soql", query:soql}});
    }
});
```

## Add the Search Result List View

The view for the search result list doesn't need a template. It is simply a container for list item views. It keeps track of these views in the `listItemViews` member. If the underlying collection changes, it renders itself again.

1. In the `<body>` block, create the view for the search result list by extending `Backbone.View`. Let's add an array for list item views as well as an `initialize()` function.

   ```
   app.views.UserListView = Backbone.View.extend({
       listItemViews: [],
       initialize: function() {
           this.model.bind("reset", this.render, this);
       },
   ```

   For the remainder of this procedure, add all code to the `extend({})` block.

2. Create the `render()` function to clean up any existing list item views by calling `close()` on each one.

   ```
       render: function(eventName) {
           _.each(this.listItemViews, function(itemView) { itemView.close(); });
   ```

3. In the `render()` function, create a new set of list item views for the records in the underlying collection. Each of these views is just an entry in the list. You'll define the `app.views.UserListItemView` later.

   ```
   this.listItemViews = _.map(this.model.models, function(model) { return new
       app.views.UserListItemView({model: model}); });
   ```

4. Append the list item views to the root DOM element.

   ```
   $(this.el).append(_.map(this.listItemViews, function(itemView) { return
   itemView.render().el;} ));
   return this;
       }
   ```

Here's the complete extension:

```
app.views.UserListView = Backbone.View.extend({

    listItemViews: [],

    initialize: function() {
        this.model.bind("reset", this.render, this);
    },
    render: function(eventName) {
        _.each(this.listItemViews, function(itemView) { itemView.close(); });
        this.listItemViews = _.map(this.model.models, function(model) {
             return new app.views.UserListItemView({model: model}); });
```

```
        $(this.el).append(_.map(this.listItemViews, function(itemView) {
             return itemView.render().el;} ));
        return this;
    }

});
```

## Add the Search Result List Item View

To define the search result list item view, you design and implement the view of a single row in a list. Each list item displays the following User fields:

- SmallPhotoUrl
- FirstName
- LastName
- Title

1. In the `<body>` block, create a template for a search result list item.

```
<script id="user-list-item" type="text/template">
  <img src="<%= SmallPhotoUrl %>" class="small-img" />
  <div class="details-short">
    <b><%= FirstName %> <%= LastName %></b><br/>
    Title<%= Title %>
  </div>
</script>
```

2. Immediately after the template, create the view for the search result list item. Once again, subclass`Backbone.View` and indicate that the whole view should be rendered as a list by defining the `tagName` member. For the remainder of this procedure, add all code in the `extend({})` block.

```
app.views.UserListItemView = Backbone.View.extend({
    tagName: "li",
});
```

3. Load template by calling `_.template()` with the raw content of the `user-list-item` script.

```
template: _.template($("#user-list-item").html()),
```

4. In the `render()` function, simply render the template using data from the model.

```
render: function(eventName) {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
},
```

5. Add a `close()` method to be called from the list view to do necessary cleanup and avoid memory leaks.

```
close: function() {
    this.remove();
    this.off();
}
```

Here's the complete extension.

```
app.views.UserListItemView = Backbone.View.extend({
    tagName: "li",
    template: _.template($("#user-list-item").html()),
    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    },
    close: function() {
        this.remove();
        this.off();
    }

});
```

## Router

A Backbone router defines navigation paths among views. To learn more about routers, see What is a router?

1. Just before the closing tag of the `<body>` block, define the application router by extending `Backbone.Router`.

   ```
   app.Router = Backbone.Router.extend({

   });
   ```

   For the remainder of this procedure, add all code in the `extend({})` block.

2. Because the app supports only one screen, you need only one "route". Add a `routes` object.

   ```
   routes: {
       "": "list"
   },
   ```

3. Define an `initialize()` function that creates the search result collections and search page view.

   ```
   initialize: function() {
       Backbone.Router.prototype.initialize.call(this);

       // Collection behind search screen
       app.searchResults = new app.models.UserCollection();
       app.searchView = new app.views.SearchPage({model: app.searchResults});
   },
   ```

4. Define the `list()` function to handle the only item in this route. When the list screen displays, fetch the search results and render the search view.

   ```
   list: function() {
      app.searchResults.fetch();
      $('#content').html(app.searchView.render().el);
   }
   ```

5. Run the application by double-clicking `index.html` to open it in a browser.

You've finished! Here's the entire application:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0;
```

```
 user-scalable=no" />
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="css/ratchet.css"/>
    <script src="jquery/jquery-2.0.0.min.js"></script>
    <script src="backbone/underscore-1.4.4.min.js"></script>
    <script src="backbone/backbone-1.0.0.min.js"></script>
    <script src="cordova-2.3.0.js"></script>
    <script src="forcetk.mobilesdk.js"></script>
    <script src="cordova.force.js"></script>
    <script src="SmartSync.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script id="search-page" type="text/template">
      <header class="bar-title">
        <h1 class="title">Users</h1>
      </header>

      <div class="bar-standard bar-header-secondary">
        <input type="search" class="search-key" placeholder="Search"/>
      </div>

      <div class="content">
        <ul class="list"></ul>
      </div>
    </script>


    <script id="user-list-item" type="text/template">
      <img src="<%= SmallPhotoUrl %>" class="small-img" />
      <div class="details-short">
        <b><%= FirstName %> <%= LastName %></b><br/>
        Title<%= Title %>
      </div>
    </script>

        <script>
var app = {
    models: {},
    views: {}
};

jQuery(document).ready(function() {
  document.addEventListener("deviceready", onDeviceReady,false);
});

function onDeviceReady() {
  cordova.require("salesforce/plugin/oauth").getAuthCredentials(appStart);
}

function appStart(creds) {
    console.log(JSON.stringify(creds));
    Force.init(creds, null, null, cordova.require("salesforce/plugin/oauth").forcetkRefresh);

    app.router = new app.Router();
    Backbone.history.start();
}


// Models
app.models.User = Force.SObject.extend({
    sobjectType: "User",
    fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title", "Email",
"MobilePhone","City"]
});

app.models.UserCollection = Force.SObjectCollection.extend({
    model: app.models.User
});
```

```
// Views

app.views.SearchPage = Backbone.View.extend({
    template: _.template($("#search-page").html()),

    initialize: function() {
        this.listView = new app.views.UserListView({model: this.model});
    },

    render: function(eventName) {
        $(this.el).html(this.template());
        $(".search-key", this.el).val(this.model.criteria);
        this.listView.setElement($("ul", this.el)).render();
        return this;
    },

    events: {
        "keyup .search-key": "search"
    },

    search: function(event) {
        this.model.criteria = $(".search-key", this.el).val();
        var soql = "SELECT Id, FirstName, LastName, SmallPhotoUrl, Title
            FROM User WHERE Name like '" + this.model.criteria + "%'
            ORDER BY Name LIMIT 25 ";
        this.model.fetch({config: {type:"soql", query:soql}});
    }
});

app.views.UserListView = Backbone.View.extend({

    listItemViews: [],

    initialize: function() {
        this.model.bind("reset", this.render, this);
    },
    render: function(eventName) {
        _.each(this.listItemViews, function(itemView) { itemView.close(); });
        this.listItemViews = _.map(this.model.models, function(model) { return new
app.views.UserListItemView({model: model}); });
        $(this.el).append(_.map(this.listItemViews, function(itemView) { return
itemView.render().el;} ));
        return this;
    }

});

app.views.UserListItemView = Backbone.View.extend({
    tagName: "li",
    template: _.template($("#user-list-item").html()),
    render: function(eventName) {
        $(this.el).html(this.template(this.model.toJSON()));
        return this;
    },
    close: function() {
        this.remove();
        this.off();
    }

});

// Router
app.Router = Backbone.Router.extend({
    routes: {
        "": "list"
    },

    initialize: function() {
        Backbone.Router.prototype.initialize.call(this);
```

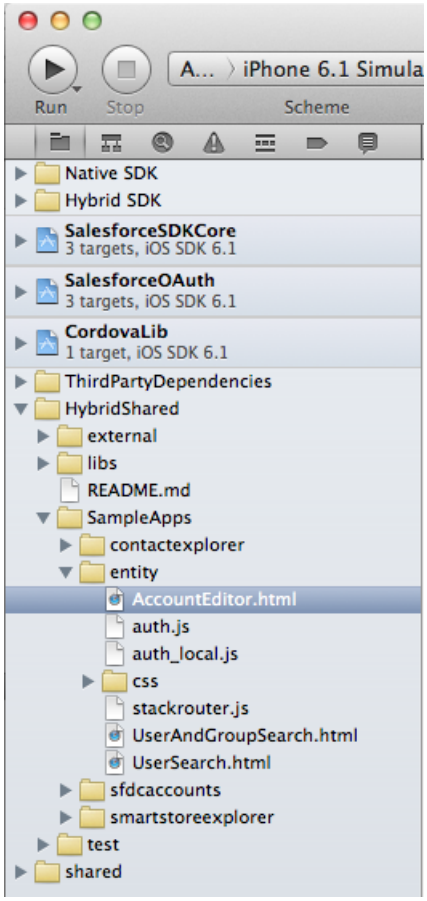
```
        // Collection behind search screen
        app.searchResults = new app.models.UserCollection();
        app.searchView = new app.views.SearchPage({model: app.searchResults});
console.log("here");
    },

    list: function() {
        app.searchResults.fetch();
        $('#content').html(app.searchView.render().el);
    }
});

    </script>
  </body>
</html>
```

# SmartSync Sample Apps

Salesforce Mobile SDK provides sample apps that demonstrate how to use SmartSync in hybrid apps. Account Editor is the most full-featured of these samples. You can switch to one of the simpler samples by changing the `startPage` property in the `bootconfig.json` file.

## Running the Samples in iOS

In your Salesforce Mobile SDK for iOS installation directory, double-click the `SalesforceMobileSDK.xcworkspace` to open it in Xcode. In Xcode, open `HybridShared/sampleApps/smartsync/AccountEditor.html`.



### Running the Samples in Android

In Android, you can run the sample from the command prompt. In your Salesforce Mobile SDK for Android installation directory, change to the `hybrid/SampleApps/AccountEditor` directory and run:

```
ant debug
ant installd
```

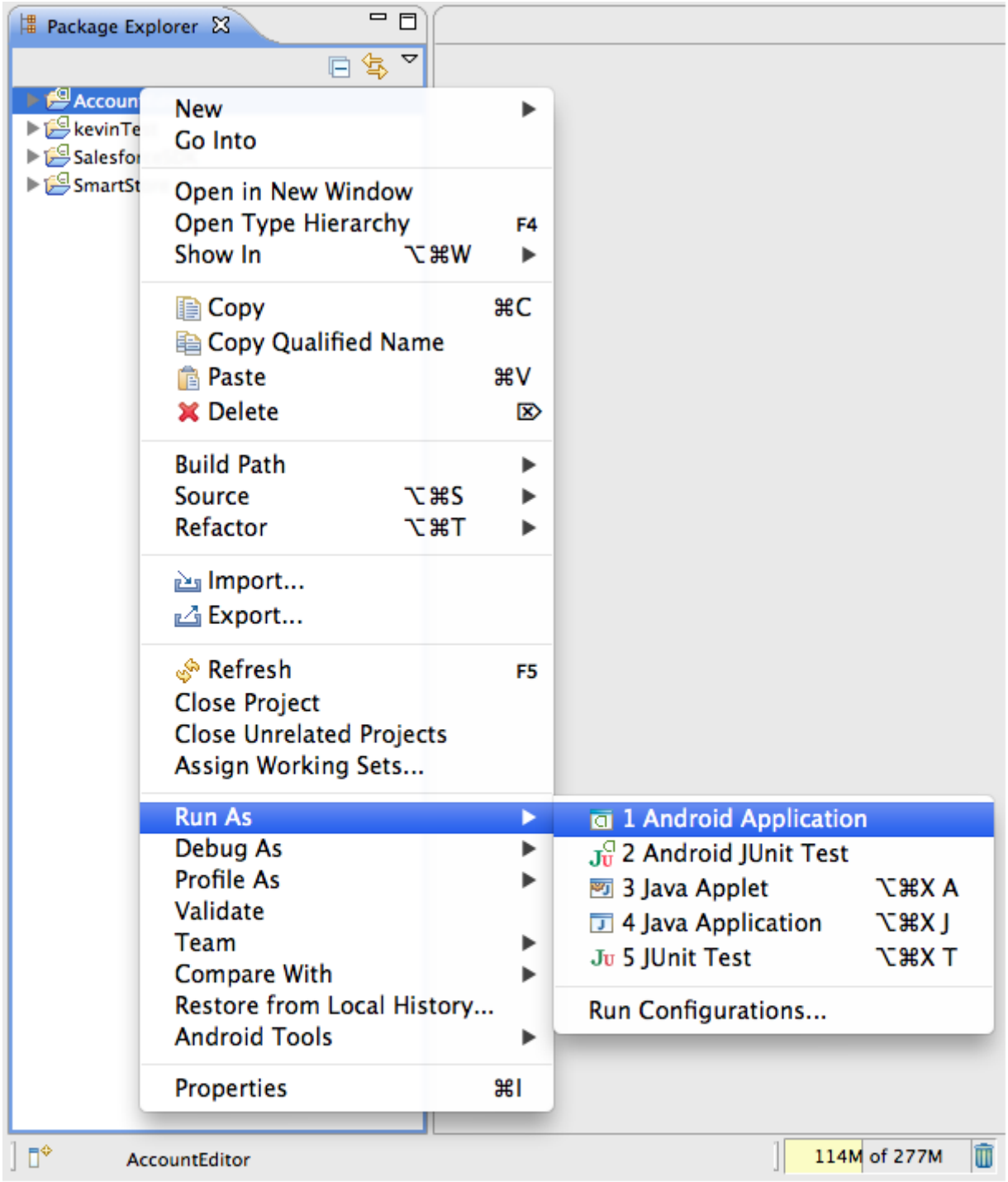
**Note:** If you get any errors saying that the `local.properties` file does not exist, run the following command from the directory shown in the error message:

```
%ANDROID_SDK%/tools/android update project -p .
```

To run the sample in Eclipse, import the following projects into your workspace:

- `forcedroid/native/SalesforceSDK`
- `forcedroid/hybrid/SmartStore`
- `forcedroid/hybrid/SampleApps/AccountEditor`

After Eclipse finishes building, Control-click or right-click **AccountEditor** in the Package Explorer, then click **Run As** > **Android application**.


Package Explorer
Accoun
kevinTe
Salesfo
SmartSt
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Build Path
Source ⌥⌘S
Refactor ⌥⌘T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Profile As
Validate
Team
Compare With
Restore from Local History...
Android Tools
Properties ⌘I
1 Android Application
2 Android JUnit Test
3 Java Applet ⌥⌘X A
4 Java Application ⌥⌘X J
5 JUnit Test ⌥⌘X T
Run Configurations...
AccountEditor
114M of 277M

# User and Group Search Sample

User and group search is the simplest SmartSync sample app. Its single screen lets you search users and collaboration groups and display matching records in a list.

To run the sample, edit `external/shared/sampleApps/smartsync/bootconfig.json`. Change `startPage` to `UserAndGroupSearch.html`:

```
{
    "remoteAccessConsumerKey":
"3MVG9Iu66FKeHhINkB1l7xt7kR8czFcCTUhgoA8Ol2Ltf1eYHOU4SqQRSEitYFDUpqRWcoQ2.dBv_a1Dyu5xa",
    "oauthRedirectURI": "testsfdc:///mobilesdk/detect/oauth/done",
    "oauthScopes": ["api","web"],
    "isLocal": true,
    "startPage": "UserAndGroupSearch.html",
    "errorPage": "error.html",
    "shouldAuthenticate": true,
    "attemptOfflineLoad": true
}
```

To run the app from Xcode in iOS, click **Run** to launch the AccountEditor project. After you've logged in, type at least two characters in the search box to see matching results.

## Looking Under the Hood

Open `UserAndGroupSearch.html` in your favorite editor. Here are the key sections of the file:

- Script includes
- Templates
- Models
- Views
- Router

## Script Includes

This sample includes the standard list of libraries for SmartSync applications.

- jQuery—See http://jquery.com/.
- Underscore—Utility-belt library for JavaScript, required by backbone) See http://underscorejs.org/
- Backbone—Gives structure to web applications. Used by SmartSync Data Framework. See http://backbonejs.org/.
- `cordova-2.3.0.js`—Required for all hybrid application used the SalesforceMobileSDK.
- `forcetk.mobilesdk.js`—Force.com JavaScript library for making Rest API calls. Required by SmartSync.
- `cordova.force.js`—As of Mobile SDK 2.0, this file combines all Force.com Cordova plugins. Replaces the `SFHybridApp.js`, `SalesforceOAuthPlugin.js`, and `SmartStorePlugin.js` files.
- `SmartSync.js`—The Mobile SDK SmartSync Data Framework.
- `fastclick.js`—Library used to eliminate the 300 ms delay between physical tap and firing of a click event. See https://github.com/ftlabs/fastclick.
- `stackrouter.js` and `auth.js`—Helper JavaScript libraries used by all three sample applications.

## Templates

Templates for this application include:

- `search-page`—template for the entire search page

- `user-list-item`—template for user list items
- `group-list-item`—template for collaboration group list items

### Models

This application defines a `SearchCollection` model.

`SearchCollection` subclasses the `Force.SObjectCollection` class, which in turn subclasses the `Collection` class from the Backbone library. Its only method configures the SOSL query used by the `fetch()` method to populate the collection.

```
app.models.SearchCollection = Force.SObjectCollection.extend({
    setCriteria: function(key) {
        this.config = {type:"sosl", query:"FIND {" + key + "*} IN ALL FIELDS RETURNING "
                        + "CollaborationGroup (Id, Name, SmallPhotoUrl, MemberCount), "
                      + "User (Id, FirstName, LastName, SmallPhotoUrl, Title ORDER BY Name)
 "
                        + "LIMIT 25"
                       };
    }
});
```

### Views

User and Group Search defines three views:

**SearchPage**

The search page expects a `SearchCollection` as its model. It watches the search input field for changes and updates the model accordingly.

```
events: {
    "keyup .search-key": "search"
},

search: function(event) {
    var key = $(".search-key", this.el).val();
    if (key.length >= 2) {
        this.model.setCriteria(key);
        this.model.fetch();
    }
}
```

**ListView**

The list portion of the search screen. `ListView` also expects a `Collection` as its model and creates `ListItemView` objects for each record in the `Collection`.

**ListItemView**

Shows details of a single list item, choosing the User or Group template based on the data.

### Router

The router does very little because this application defines only one screen.

## User Search Sample

User Search is a more elaborate sample than User and Group search. Instead of a single screen, it defines two screens. If your search returns a list of matches, User Search lets you tap on each of them to see a basic detail screen. Because it defines more than one screen, this sample also demonstrates the use of a router.

To run the sample, edit `external/shared/sampleApps/smartsync/bootconfig.json`. Change `startPage` to `UserSearch.html`:

```
{
    "remoteAccessConsumerKey":
"3MVG9Iu66FKeHhINkB1l7xt7kR8czFcCTUhgoA8Ol2Ltf1eYHOU4SqQRSEitYFDUpqRWcoQ2.dBv_a1Dyu5xa",
    "oauthRedirectURI": "testsfdc:///mobilesdk/detect/oauth/done",
    "oauthScopes": ["api","web"],
    "isLocal": true,
    "startPage": "UserSearch.html",
    "errorPage": "error.html",
    "shouldAuthenticate": true,
    "attemptOfflineLoad": true
}
```

In Xcode or Eclipse, launch the AccountEditor. Log in if prompted to do so. Unlike the User and Group Search example, you need to type only a single character in the search box to begin seeing search results. That's because this application uses SOQL, rather than SOSL, to query the server.

When you tap an entry in the search results list, you see a basic detail screen.

## Looking Under the Hood

Open the UserSearch.html file in your favorite editor. Here are the key sections of the file:

- Script includes
- Templates
- Models
- Views
- Router

## Script Includes

This sample includes the standard list of libraries for SmartSync applications.

- jQuery—See http://jquery.com/.
- Underscore—Utility-belt library for JavaScript, required by backbone) See http://underscorejs.org/
- Backbone—Gives structure to web applications. Used by SmartSync Data Framework. See http://backbonejs.org/.
- `cordova-2.3.0.js`—Required for all hybrid application used the SalesforceMobileSDK.
- `forcetk.mobilesdk.js`—Force.com JavaScript library for making Rest API calls. Required by SmartSync.
- `cordova.force.js`—As of Mobile SDK 2.0, this file combines all Force.com Cordova plugins. Replaces the `SFHybridApp.js`, `SalesforceOAuthPlugin.js`, and `SmartStorePlugin.js` files.
- `SmartSync.js`—The Mobile SDK SmartSync Data Framework.
- `fastclick.js`—Library used to eliminate the 300 ms delay between physical tap and firing of a click event. See https://github.com/ftlabs/fastclick.
- `stackrouter.js` and `auth.js`—Helper JavaScript libraries used by all three sample applications.

## Templates

Templates for this application include:

- `search-page`—template for the whole search page
- `user-list-item`—template for user list items
- `user-page`—template for user detail page

## Models

This application defines two models: `UserCollection` and `User`.

`UserCollection` subclasses the `Force.SObjectCollection` class, which in turn subclasses the `Collection` class from the Backbone library. Its only method configures the SOQL query used by the `fetch()` method to populate the collection.

```
app.models.UserCollection = Force.SObjectCollection.extend({
    model: app.models.User,
    fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title"],

    setCriteria: function(key) {
        this.key = key;
        this.config = {type:"soql", query:"SELECT " + this.fieldlist.join(",")
                        + " FROM User"
                        + " WHERE Name like '" + key + "%'"
                        + " ORDER BY Name "
                        + " LIMIT 25 "
                       };
    }
});
```

`User` subclasses SmartSync's `Force.SObject` class. The `User` model defines:

- An `sobjectType` field to indicate which type of sObject it represents (User, in this case).
- A `fieldlist` field that contains the list of fields to be fetched from the server

Here's the code:

```
app.models.User = Force.SObject.extend({
    sobjectType: "User",
    fieldlist: ["Id", "FirstName", "LastName", "SmallPhotoUrl", "Title", "Email",
"MobilePhone","City"]
});
```

## Views

This sample defines four views:

**SearchPage**

View for the entire search page. It expects a `UserCollection` as its model. It watches the search input field for changes and updates the model accordingly in the `search()` function.

```
events: {
    "keyup .search-key": "search"
},

search: function(event) {
    this.model.setCriteria($(".search-key", this.el).val());
    this.model.fetch();
}
```

**UserListView**

View for the list portion of the search screen. It also expects a `UserCollection` as its model and creates `UserListItemView` objects for each user in the `UserCollection` object.

**UserListItemView**

View for a single list item.

**UserPage**

View for displaying user details.

### Router

The `router` class handles navigation between the app's two screens. This class uses a `routes` field to map those view to `router` class method.

```
routes: {
    "": "list",
    "list": "list",
    "users/:id": "viewUser"
},
```

The list page calls `fetch()` to fill the search result collections, then brings the search page into view.

```
list: function() {
    app.searchResults.fetch();
    // Show page right away - list will redraw when data comes in
    this.slidePage(app.searchPage);
},
```

The user detail page calls `fetch()` to fill the user model, then brings the user detail page into view.

```
viewUser: function(id) {
    var that = this;
    var user = new app.models.User({Id: id});
    user.fetch({
        success: function() {
            app.userPage.model = user;
            that.slidePage(app.userPage);
        }
    });
}
```

## Account Editor Sample

Account Editor is the most complex SmartSync-based sample application in Mobile SDK 2.0. It allows you to create/edit/update/delete accounts online and offline, and also demonstrates conflict detection.

To run the sample:

1. If you've made changes to `external/shared/sampleApps/smartsync/bootconfig.json`, revert it to its origin content.
2. Launch Account Editor.

This application contains three screens:

- Accounts search
- Accounts detail
- Sync

When the application first starts, you see the Accounts search screen listing the most recently used accounts. In this screen, you can:

- Type a search string to find accounts whose names contain the given string.
- Tap an account to launch the account detail screen.
- Tap **Create** to launch an empty account detail screen.
- Tap **Online** to go offline. If you are already offline, you can tap the **Offline** button to go back online. (You can also go offline by putting the device in airplane mode.)

To launch the Account Detail screen, tap an account record in the Accounts search screen. The detail screen shows you the fields in the selected account. In this screen, you can:

- Tap a field to change its value.
- Tap **Save** to update or create the account. If validation errors occur, the fields with problems are highlighted.

  If you're online while saving and the server's record changed since the last fetch, you receive warnings for the fields that changed remotely.

  Two additional buttons, **Merge** and **Overwrite**, let you control how the app saves your changes. If you tap **Overwrite,** the app saves to the server all values currently displayed on your screen. If you tap **Merge**, the app saves to the server only the fields you changed, while keeping changes on the server in fields you did not change.
- Tap **Delete** to delete the account.
- Tap **Online** to go offline, or tap **Offline** to go online.

To see the Sync screen, tap **Online** to go offline, then create, update, or delete an account. When you tap **Offline** again to go back online, the Sync screen shows all accounts that you modified on the device.

Tap **Process *n* records** to try to save your local changes to the server. If any account fails to save, it remains in the list with a notation that it failed to sync. You can tap any account in the list to edit it further or, in the case of a locally deleted record, to undelete it.

## Looking Under the Hood

To view the source code for this sample, open `AccountEditor.html` in an HTML or text editor.

Here are the key sections of the file:

- Script includes
- Templates
- Models
- Views
- Router

## Script Includes

This sample includes the standard list of libraries for SmartSync applications.

- jQuery—See http://jquery.com/.
- Underscore—Utility-belt library for JavaScript, required by backbone. See http://underscorejs.org/.
- Backbone—Gives structure to web applications. Used by SmartSync Data Framework. See http://backbonejs.org/.
- `cordova-2.3.0.js`—Required for hybrid applications using the Salesforce Mobile SDK.
- `forcetk.mobilesdk.js`—Force.com JavaScript library for making REST API calls. Required by SmartSync.
- `cordova.force.js`—As of Mobile SDK 2.0, this file combines all Force.com Cordova plugins. Replaces the `SFHybridApp.js`, `SalesforceOAuthPlugin.js`, and `SmartStorePlugin.js` files.
- `SmartSync.js`—The Mobile SDK SmartSync Data Framework.
- `fastclick.js`—Library used to eliminate the 300 ms delay between physical tap and firing of a click event. See https://github.com/ftlabs/fastclick.
- `stackrouter.js` and `auth.js`—Helper JavaScript libraries used by all three sample applications.

## Templates

Templates for this application include:

- search-page
- sync-page

- account-list-item
- edit-account-page (for the Account detail page)

## Models

This sample defines three models: `AccountCollection`, `Account` and `OfflineTracker`.

`AccountCollection` is a subclass of SmartSync's `Force.SObjectCollection` class, which is a subclass of the Backbone framework's `Collection` class.

The `AccountCollection.config()` method returns an appropriate query to the collection. The query mode can be:

- Most recently used (MRU) if you are online and haven't provided query criteria
- SOQL if you are online and have provided query criteria
- SmartSQL when you are offline

When the app calls `fetch()` on the collection, the `fetch()` function executes the query returned by `config()`. It then uses the results of this query to populate `AccountCollection` with Account objects from either the offline cache or the server.

`AccountCollection` uses the two global caches set up by the AccountEditor application: `app.cache` for offline storage, and `app.cacheForOriginals` for conflict detection. The code shows that the `AccountCollection` model:

- Contains objects of the `app.models.Account` model (`model` field)
- Specifies a list of fields to be queried (`fieldlist` field)
- Uses the sample app's global offline cache (`cache` field)
- Uses the sample app's global conflict detection cache (`cacheForOriginals` field)
- Defines a `config()` function to handle online as well as offline queries

Here's the code (shortened for readability):

```
app.models.AccountCollection = Force.SObjectCollection.extend({
    model: app.models.Account,
    fieldlist: ["Id", "Name", "Industry", "Phone", "Owner.Name", "LastModifiedBy.Name",
"LastModifiedDate"],
    cache: function() { return app.cache},
    cacheForOriginals: function() { return app.cacheForOriginals;},

    config: function() {
        // Offline: do a cache query
        if (!app.offlineTracker.get("isOnline")) {
...
        }
        // Online
        else {
...
        }
    }
});
```

`Account` is a subclass of SmartSync's `Force.SObject` class, which is a subclass of the Backbone framework's `Model` class. Code for the `Account` model shows that it:

- Uses a `sobjectType` field to indicate which type of sObject it represents (Account, in this case).
- Defines `fieldlist` as a method rather than a field, because the fields that it retrieves from the server are not the same as the ones it sends to the server.
- Uses the sample app's global offline cache (`cache` field).
- Uses the sample app's global conflict detection cache (`cacheForOriginals` field).
- Supports a `cacheMode()` method that returns a value indicating how to handle caching based on the current offline status.

Here's the code:

```
app.models.Account = Force.SObject.extend({
    sobjectType: "Account",
    fieldlist: function(method) {
        return method == "read"
            ? ["Id", "Name", "Industry", "Phone", "Owner.Name", "LastModifiedBy.Name",
"LastModifiedDate"]
            : ["Id", "Name", "Industry", "Phone"];
    },
    cache: function() { return app.cache;},
    cacheForOriginals: function() { return app.cacheForOriginals;},
    cacheMode: function(method) {
        if (!app.offlineTracker.get("isOnline")) {
            return Force.CACHE_MODE.CACHE_ONLY;
        }
        // Online
        else {
            return (method == "read" ? Force.CACHE_MODE.CACHE_FIRST :
Force.CACHE_MODE.SERVER_FIRST);
        }
    }
});
```

`OfflineTracker` is a subclass of Backbone's `Model` class. This class tracks the offline status of the application by observing the browser's offline status. It automatically switches the app to offline when it detects that the browser is offline. However, it goes online only when the user requests it.

Here's the code:

```
app.models.OfflineTracker = Backbone.Model.extend({
    initialize: function() {
        var that = this;
        this.set("isOnline", navigator.onLine);
        document.addEventListener("offline", function() {
            console.log("Received OFFLINE event");
            that.set("isOnline", false);
        }, false);
        document.addEventListener("online", function() {
            console.log("Received ONLINE event");
            // User decides when to go back online
        }, false);
    }
});
```

## Views

This sample defines five views:

- SearchPage
- AccountListView
- AccountListItemView
- EditAccountView
- SyncPage

A view typically provides a template field to specify its design template, an `initialize()` function, and a `render()` function.

Each view can also define an `events` field. This field contains an array whose key/value entries specify the event type and the event handler function name. Entries use the following format:

```
"<event-type>[ <control>]": "<event-handler-function-name>"
```

For example:

```
events: {
    "click .button-prev": "goBack",
    "change": "change",
    "click .save": "save",
    "click .merge": "saveMerge",
    "click .overwrite": "saveOverwrite",
    "click .toggleDelete": "toggleDelete"
},
```

**SearchPage**

View for the entire search screen. It expects an `AccountCollection` as its model. It watches the search input field for changes (the `keyup` event) and updates the model accordingly in the `search()` function.

```
events: {
    "keyup .search-key": "search"
},
search: function(event) {
    this.model.setCriteria($(".search-key", this.el).val());
    this.model.fetch();
}
```

**AcountListView**

View for the list portion of the search screen. It expects an `AccountCollection` as its model and creates `AccountListItemView` object for each account in the `AccountCollection` object.

**AccountListItemView**

View for an item within the list.

**EditAccountPage**

View for account detail page. This view monitors several events:

| Event Type | Target Control | Handler function name |
|---|---|---|
| click | button-prev | goBack |
| change | Not set (can be any edit control) | change |
| click | save | save |
| click | merge | saveMerge |
| click | overwrite | saveOverwrite |
| click | toggleDelete | toggleDelete |

A couple of event handler functions deserve special attention. The `change()` function shows how the view uses the event target to send user edits back to the model:

```
change: function(evt) {
    // apply change to model
    var target = event.target;
    this.model.set(target.name, target.value);
    $("#account" + target.name + "Error", this.el).hide();
}
```

The `toggleDelete()` function handles a toggle that lets the user delete or undelete an account. If the user clicks to undelete, the code sets an internal `__locally_deleted__` flag to false to indicate that the record is no longer deleted in the cache. Else, it attempts to delete the record on the server by destroying the local model.

```
toggleDelete: function() {
    if (this.model.get("__locally_deleted__")) {
        this.model.set("__locally_deleted__", false);
        this.model.save(null, this.getSaveOptions(null, Force.CACHE_MODE.CACHE_ONLY));

    }
    else {
        this.model.destroy({
            success: function(data) {
                app.router.navigate("#", {trigger:true});
            },
            error: function(data, err, options) {
                var error = new Force.Error(err);
                alert("Failed to delete account: " + (error.type === "RestError" ?
                          error.details[0].message :
                          "Remote change detected - delete aborted"));
            }
        });
    }
}
```

**SyncPage**

View for the sync page. This view monitors several events:

| Event Type | Control | Handler function name |
|---|---|---|
| click | button-prev | goBack |
| click | sync | sync |

To see how the all screen is rendered, look at the render method:

```
render: function(eventName) {

    $(this.el).html(this.template(_.extend(
        {countLocallyModified: this.model.length},
        this.model.toJSON())));

    this.listView.setElement($("ul", this.el)).render();

    return this;

},
```

Let's take a look at what happens when the user taps **Process** (the sync control).

The `sync()` function looks at the first locally modified Account in the view's collection and tries to save it to the server. If the save succeeds and there are no more locally modified records, the app navigates back to the search screen. Otherwise, the app marks the account as having failed locally and then calls `sync()` again.

```
sync: function(event) {
    var that = this;
    if (this.model.length == 0 || this.model.at(0).get("__sync_failed__")) {
        // we push sync failures back to the end of the list -
        // if we encounter one, it means we are done
        return;
    }
    else {
```

```
        var record = this.model.shift();

        var options = {
            mergeMode: Force.MERGE_MODE.MERGE_FAIL_IF_CHANGED,
            success: function() {
                if (that.model.length == 0) {
                    app.router.navigate("#", {trigger:true});
                }
                else {
                    that.sync();
                }
            },
            error: function() {
                record = record.set("__sync_failed__", true);
                that.model.push(record);
                that.sync();
            }
        };
        return record.get("__locally_deleted__") ? record.destroy(options) :
            record.save(null, options);
    }
});
```

## Router

When the router is initialized, it sets up the two global caches used throughout the sample.

```
setupCaches: function() {
    // Cache for offline support
    app.cache = new Force.StoreCache("accounts", [ {path:"Name", type:"string"} ]);

    // Cache for conflict detection
    app.cacheForOriginals = new Force.StoreCache("original-accounts");

    return $.when(app.cache.init(), app.cacheForOriginals.init());
},
```

Once the global caches are set up, it also sets up two `AccountCollection` objects: One for the search screen, and one for the sync screen.

```
// Collection behind search screen
app.searchResults = new app.models.AccountCollection();

// Collection behind sync screen
app.localAccounts = new app.models.AccountCollection();
app.localAccounts.config = {type:"cache", cacheQuery: {queryType:"exact",
indexPath:"__local__", matchKey:true, order:"ascending", pageSize:25}};
```

Finally, it creates the view objects for the Search, Sync, and EditAccount screens.

```
// We keep a single instance of SearchPage / SyncPage and EditAccountPage
app.searchPage = new app.views.SearchPage({model: app.searchResults});
app.syncPage = new app.views.SyncPage({model: app.localAccounts});
app.editPage = new app.views.EditAccountPage();
```

The router has a `routes` field that maps actions to methods on the router class.

```
routes: {
    "": "list",
    "list": "list",
    "add": "addAccount",
    "edit/accounts/:id": "editAccount",
```

```
    "sync":"sync"
},
```

The `list` action fills the search result collections by calling `fetch()` and brings the search page into view.

```
list: function() {
    app.searchResults.fetch();
    // Show page right away - list will redraw when data comes in
    this.slidePage(app.searchPage);
},
```

The `addAccount` action creates an empty account object and bring the edit page for that account into view.

```
addAccount: function() {
    app.editPage.model = new app.models.Account({Id: null});
    this.slidePage(app.editPage);
},
```

The `editAccount` action fetches the specified Account object and brings the account detail page into view.

```
editAccount: function(id) {
    var that = this;
    var account = new app.models.Account({Id: id});
    account.fetch({
        success: function(data) {
            app.editPage.model = account;
            that.slidePage(app.editPage);
        },
        error: function() {
            alert("Failed to get record for edit");
        }
    });
}
```

The sync action computes the `localAccounts` collection by calling fetch and brings the sync page into view.

```
sync: function() {
    app.localAccounts.fetch();
    // Show page right away - list will redraw when data comes in
    this.slidePage(app.syncPage);
}
```

# Chapter 7

# Securely Storing Data Offline

**In this chapter ...**



Mobile devices can lose connection at any time, and environments such as hospitals and airplanes often prohibit connectivity. To handle these situations, it's important that your mobile apps continue to function when they go offline.

The Mobile SDK uses SmartStore, a secure offline storage solution on your device. SmartStore allows you to continue working even when the device is not connected to the Internet. SmartStore stores data as JSON documents in a data structure called a *soup*. A soup is a simple one-table database of "entries" which can be indexed in different ways and queried by a variety of methods.

Mobile SDK 2.0 provides a StoreCache mechanism that works with SmartStore soups to provide offline synchronization and conflict resolution services. You can control these services by providing simple configuration settings. We recommend that you use StoreCache to manage operations on Salesforce data. See Using StoreCache For Offline Caching on page 70 and Conflict Detection on page 73

**Note:** Pure HTML5 apps store offline information in a browser cache. Browser caching isn't part of the Mobile SDK, and we don't document it here. SmartStore uses storage functionality on the device. This strategy requires a native or hybrid development path.

## Sample Objects

The code snippets in this chapter use two objects, Account and Opportunity, which come predefined with every Salesforce organization. Account and Opportunity have a master-detail relationship; an Account can have more than one Opportunity.

# Accessing SmartStore in Hybrid Apps

Hybrid apps access SmartStore from JavaScript. In order to enable offline access in a hybrid mobile application, you need to include a couple of JavaScript and CSS files in your Visualforce or HTML page.

- `cordova-x.x.x.js` — The Cordova library (formerly PhoneGap).
- `cordova.force.js` – Contains the JavaScript portion of Salesforce OAuth and SmartStore plugins. Also includes methods that perform utility tasks, such as determining whether you're offline.

# Adding SmartStore to Android Apps

In Android apps, SmartStore is an optional component. It is not optional in iOS apps.

To use SmartStore in an Android app, you need to configure your project to include it. When you create a new Android project with the forcedroid utility, include SmartStore by setting the optional `– –usesmartstore=true` parameter. See Creating a New Android Project on page 30 for examples.

To add SmartStore to an existing Android project (hybrid or native):

1. Add the SmartStore library project to your project. In Eclipse, choose Properties from the Project menu. Select Android from the left panel, then click Add on the right-hand side. Choose the hybrid/SmartStore project.
2. In your *projectname*App.java file, import the SalesforceSDKManagerWithSmartStore class instead of SalesforceSDKManager. Replace this statement:

   ```
   import com.salesforce.androidsdk.app.SalesforceSDKManager
   ```

   with this one:

   ```
   import com.salesforce.androidsdk.smartstore.app.SalesforceSDKManagerWithSmartStore
   ```

3. In your *projectname*App.java file, change your App class to extend the SalesforceSDKManagerWithSmartStore class rather than SalesforceSDKManager.

# Offline Hybrid Development

Developing a hybrid application inside the container requires a build/deploy step for every change. For that reason, we recommend you develop your hybrid application directly in a browser, and only run your code in the container in the final stages of testing. JavaScript development in a browser is easier because there is no build/compile step. Whenever you make changes to the code, you can refresh the browser to see your changes.

We recommend using the Google Chrome browser because it comes bundled with developer tools that let you access the internals of the your web applications. For more information, see Chrome Developer Tools: Overview.

# SmartStore Soups

You store your offline data in SmartStore in one or more *soups*. A soup, conceptually speaking, is a logical collection of data records—represented as JSON objects—that you want to store and query offline. In the Force.com world, a soup will typically map to a standard or custom object that you wish to store offline, but that is not a hard and fast rule. You can store as many soups as you want in an application, but remember that soups are meant to be self-contained data sets; there is no direct

correlation between them. In addition to storing the data itself, you can also specify indices that map to fields within the data, for greater ease and customization of data queries.

**Note:**

SmartStore data is inherently volatile. Its lifespan is tied to the authenticated user as well as to OAuth token states. When the user logs out of the app, SmartStore deletes all soup data associated with that user. Similarly, when the OAuth refresh token is revoked or expires, the user's app state is reset, and all data in SmartStore is purged. Carefully consider the volatility of SmartStore data when designing your app. This warning is especially important if your org sets a short lifetime for the refresh token.

# Registering a Soup

In order to access a soup, you first need to register it. Provide a name, index specifications, and names of callback functions for success and error conditions:

```
navigator.smartstore.registerSoup(soupName, indexSpecs, successCallback, errorCallback)
```

If the soup does not already exist, this function creates it. If the soup already exists, registering gives you access to the existing soup. To find out if a soup already exists, use:

```
navigator.smartstore.soupExists(soupName, successCallback, errorCallback);
```

A soup is indexed on one or more fields found in its entries. Insert, update, and delete operations on soup entries are tracked in the soup indices. Always specify at least one index field when registering a soup. For example, if you are using the soup as a simple key/value store, use a single index specification with a string type.

### `indexSpecs`

The `indexSpecs` array is used to create the soup with predefined indexing. Entries in the `indexSpecs` array specify how the soup should be indexed. Each entry consists of a `path:type` pair. `path` is the name of an index field; `type` is either "string", "integer", or "floating". Index paths are case-sensitive and can include compound paths, such as Owner.Name.

**Note:** Performance can suffer if the index path is too deep. If index entries are missing any fields described in a particular `indexSpec`, they will not be tracked in that index.

```
"indexSpecs":[
    {
        "path":"Name",
        "type":"string"
    }
    {
        "path":"Id",
        "type":"string"
    }
    {
        "path":"ParentId",
        "type":"string"
    }
    {
        "path":"lastModifiedDate",
        "type":"integer"
    }
]
```

**Note:** Currently, the Mobile SDK supports three index types: “string”, “integer”, and “floating”. These types apply only to the index itself, and not to the way data is stored or retrieved. It’s OK to have a null field in an index column.

### successCallback

The success callback function for `registerSoup` takes one argument (the soup name).

```
function(soupName) { alert("Soup " + soupName + " was successfully created"); };
```

A successful creation of the soup returns a `successCallback` that indicates the soup is ready. Wait to complete the transaction and receive the callback before you begin any activity. If you register a soup under the passed name, the success callback function returns the soup.

### errorCallback

The error callback function for `registerSoup` takes one argument (the error description string).

```
function(err) { alert ("registerSoup failed with error:" + err); }
```

During soup creation, errors can happen for a number of reasons, including:

- An invalid or bad soup name
- No index (at least one index must be specified)
- Other unexpected errors, such as a database error

## Retrieving Data From a Soup

SmartStore provides a set of helper methods that build query strings for you. To query a specific set of records, call the `build*` method that suits your query specification. You can optionally define the index field, sort order, and other metadata to be used for filtering, as described in the following table:

| Parameter | Description |
|---|---|
| `indexPath` | This is what you’re searching for; for example a name, account number, or date. |
| `beginKey` | Optional. Used to define the start of a range query. |
| `endKey` | Optional. Used to define the end of a range query. |
| `order` | Optional. Either “ascending” or “descending.” |
| `pageSize` | Optional. If not present, the native plugin can return whatever page size it sees fit in the resulting `Cursor.pageSize`. |

**Note:**

All queries are single-predicate searches. Only SmartSQL queries support joins.

### Query Everything

`buildAllQuerySpec(indexPath, order, [pageSize])` returns all entries in the soup, with no particular order. Use this query to traverse everything in the soup.

`order` and `pageSize` are optional, and default to ascending and 10, respectively. You can specify:

- `buildAllQuerySpec(indexPath)`
- `buildAllQuerySpec(indexPath, order)`
- `buildAllQuerySpec(indexPath, order, [pageSize])`

However, you can't specify `buildAllQuerySpec(indexPath,[pageSize])`.

See Working With Cursors for information on page sizes.

**Note:** As a base rule, set `pageSize` to the number of entries you want displayed on the screen. For a smooth scrolling display, you might want to increase the value to two or three times the number of entries actually shown.

## Query with a Smart SQL SELECT Statement

`buildSmartQuerySpec(smartSql, [pageSize])` executes the query specified by `smartSql`. This function allows greater flexibility than other query factory functions because you provide your own Smart SQL SELECT statement. See Smart SQL Queries.

`pageSize` is optional and defaults to 10

Sample code, in various development environments, for a Smart SQL query that calls the SQL `COUNT` function:

**Javascript:**

```
var querySpec = navigator.smartstore.buildSmartQuerySpec("select count(*) from {employees}",
 1);
navigator.smartstore.runSmartQuery(querySpec, function(cursor) {
 // result should be [[ n ]] if there are n employees
});
```

**iOS native:**

```
SFQuerySpec* querySpec = [SFQuerySpec newSmartQuerySpec:@"select count(*) from {employees}"
 withPageSize:1];
NSArray* result = [_store queryWithQuerySpec:querySpec pageIndex:0];
// result should be [[ n ]] if there are n employees
```

**Android native:**

```
SmartStore store = SalesforceSDKManagerWithSmartStore.getInstance().getSmartStore();
JSONArray result = store.query(QuerySpec.buildSmartQuerySpec("select count(*) from
{employees}", 1), 0);
// result should be [[ n ]] if there are n employees
```

## Query by Exact

`buildExactQuerySpec(indexPath, matchKey, [pageSize])` finds entries that exactly match the given `matchKey` for the `indexPath` value. Use this to find child entities of a given ID. For example, you can find Opportunities by Status. However, you can't specify order in the results.

Sample code for retrieving children by ID:

```
var querySpec = navigator.smartstore.buildExactQuerySpec("sfdcId", "some-sfdc-id");
navigator.smartstore.querySoup("Catalogs", querySpec, function(cursor) {
       // we expect the catalog to be in: cursor.currentPageOrderedEntries[0]
});
```

Sample code for retrieving children by parent ID:

```
var querySpec = navigator.smartstore.buildExactQuerySpec("parentSfdcId", "some-sfdc-id);
navigator.smartstore.querySoup("Catalogs", querySpec, function(cursor) {});
```

### Query by Range

`buildRangeQuerySpec(indexPath, beginKey, endKey, [order, pageSize])` finds entries whose `indexPath` values fall into the range defined by `beginKey` and `endKey`. Use this function to search by numeric ranges, such as a range of dates stored as integers.

`order` and `pageSize` are optional, and default to ascending and 10, respectively. You can specify:

- `buildRangeQuerySpec(indexPath, beginKey, endKey)`
- `buildRangeQuerySpec(indexPath, beginKey, endKey, order)`
- `buildRangeQuerySpec(indexPath, beginKey, endKey, order, pageSize)`

However, you can't specify `buildRangeQuerySpec(indexPath, beginKey, endKey, pageSize)`.

By passing null values to `beginKey` and `endKey`, you can perform open-ended searches:

- Passing `null` to `endKey` finds all records where the field at `indexPath` is >= `beginKey`.
- Passing `null` to `beginKey` finds all records where the field at `indexPath` is <= `endKey`.
- Passing `null` to both `beginKey` and `endKey` is the same as querying everything.

### Query by Like

`buildLikeQuerySpec(indexPath, likeKey, [order, pageSize])` finds entries whose `indexPath` values are like the given `likeKey`. You can use "foo%" to search for terms that begin with your keyword, "%foo" to search for terms that end with your keyword, and "%foo%" to search for your keyword anywhere in the `indexPath` value. Use this function for general searching and partial name matches. `order` and `pageSize` are optional, and default to ascending and 10, respectively.

**Note:** Query by Like is the slowest of the query methods.

### Executing the Query

Queries run asynchronously and return a cursor to your JavaScript callback. Your success callback should be of the form `function(cursor)`. Use the `querySpec` parameter to pass your query specification to the `querySoup` method.

```
navigator.smartstore.querySoup(soupName,querySpec,successCallback,errorCallback);
```

### Retrieving Individual Soup Entries by Primary Key

All soup entries are automatically given a unique internal ID (the primary key in the internal table that holds all entries in the soup). That ID field is made available as the `_soupEntryId` field in the soup entry. Soup entries can be looked up by `_soupEntryId` by using the `retrieveSoupEntries` method. Note that the return order is not guaranteed, and if entries have been deleted they will be missing from the resulting array. This method provides the fastest way to retrieve a soup entry, but it's usable only when you know the `_soupEntryId`:

```
navigator.smartStore.retrieveSoupEntries(soupName, indexSpecs, successCallback, errorCallback)
```

# Smart SQL Queries

Beginning with Salesforce Mobile SDK version 2.0, SmartStore supports a Smart SQL query language for free-form SELECT statements. Smart SQL queries combine standard SQL SELECT grammar with additional descriptors for referencing soups and soup fields. This approach gives you maximum control and flexibility, including the ability to use joins. Smart SQL supports all standard SQL SELECT constructs.

## Smart SQL Restrictions

Smart SQL supports only SELECT statements and only indexed paths.

## Syntax

Syntax is identical to the standard SQL SELECT specification but with the following adaptations.

| Usage | Syntax |
|---|---|
| To specify a column | `{<soupName>:<path>}` |
| To specify a table | `{<soupName>}` |
| To refer to the entire soup entry JSON string | `{<soupName>:_soup}` |
| To refer to the internal soup entry ID | `{<soupName>:_soupEntryId}` |
| To refer to the last modified date | `{<soupName>:_soupLastModifiedDate}` |

## Sample Queries

Consider two soups: one named Employees, and another named Departments. The Employees soup contains standard fields such as:

- First name (`firstName`)
- Last name (`lastName`)
- Department code (`deptCode`)
- Employee ID (`employeeId`)
- Manager ID (`managerId`)

The Departments soup contains:

- Name (`name`)
- Department code (`deptCode`)

Here are some examples of basic Smart SQL queries using these soups:

```
select {employees:firstName}, {employees:lastName}
from {employees} order by {employees:lastName}

select {departments:name}
from {departments}
order by {departments:deptCode}
```

## Joins

Smart SQL also allows you to use joins. For example:

```
select {departments:name}, {employees:firstName} || ' ' || {employees:lastName}
from {employees}, {departments}
where {departments:deptCode} = {employees:deptCode}
order by {departments:name}, {employees:lastName}
```

You can even do self joins:

```
select mgr.{employees:lastName}, e.{employees:lastName}
from {employees} as mgr, {employees} as e
where mgr.{employees:employeeId} = e.{employees:managerId}
```

### Aggregate Functions

Smart SQL support the use of aggregate functions such as:

- COUNT
- SUM
- AVG

For example:

```
select {account:name},
    count({opportunity:name}),
    sum({opportunity:amount}),
    avg({opportunity:amount}),
    {account:id},
    {opportunity:accountid}
from {account},
    {opportunity}
where {account:id} = {opportunity:accountid}
group by {account:name}
```

The NativeSqlAggregator sample app delivers a fully implemented native implementation of SmartStore, including SmartSQL support for aggregate functions and joins. See NativeSqlAggregator Sample App: Using SmartStore in Native Apps on page 113.

## Working With Cursors

Queries can potentially have long result sets that are too large to load. Instead, only a small subset of the query results (a single page) is copied from the native realm to the JavaScript realm at any given time. When you perform a query, a cursor object is returned from the native realm that provides a way to page through a list of query results. The JavaScript code can then move forward and back through the pages, causing pages to be copied to the JavaScript realm.

**Note:** For advanced users: Cursors are not snapshots of data; they are dynamic. If you make changes to the soup and then start paging through the cursor, you will see those changes. The only data the cursor holds is the original query and your current position in the result set. When you move your cursor, the query runs again. Thus, newly created soup entries can be returned (assuming they satisfy the original query).

Use the following cursor functions to navigate the results of a query:

- `navigator.smartstore.moveCursorToPageIndex(cursor, newPageIndex, successCallback, errorCallback)`—Move the cursor to the page index given, where 0 is the first page, and the last page is defined by `totalPages - 1`.
- `navigator.smartstore.moveCursorToNextPage(cursor, successCallback, errorCallback)`—Move to the next entry page if such a page exists.
- `navigator.smartstore.moveCursorToPreviousPage(cursor, successCallback, errorCallback)`—Move to the previous entry page if such a page exists.
- `navigator.smartstore.closeCursor(cursor, successCallback, errorCallback)`—Close the cursor when you're finished with it.

**Note:** `successCallback` for those functions should expect one argument (the updated cursor).

# Manipulating Data

In order to track soup entries for insert, update, and delete, SmartStore adds a few fields to each entry:

- `_soupEntryId`—This field is the primary key for the soup entry in the table for a given soup.
- `_soupLastModifiedDate`—The number of milliseconds since 1/1/1970.
  - To convert to a JavaScript date, use `new Date(entry._soupLastModifiedDate)`
  - To convert a date to the corresponding number of milliseconds since 1/1/1970, use `date.getTime()`

When inserting or updating soup entries, SmartStore automatically sets these fields. When removing or retrieving specific entries, you can reference them by `_soupEntryId`.

## Inserting or Updating Soup Entries

If the provided soup entries already have the `_soupEntryId` slots set, then entries identified by that slot are updated in the soup. If an entry does not have a `_soupEntryId` slot, or the value of the slot doesn't match any existing entry in the soup, then the entry is added (inserted) to the soup, and the `_soupEntryId` slot is overwritten.

**Note:** You must not manipulate the `_soupEntryId` or `_soupLastModifiedDate` value yourself.

Use the `upsertSoupEntries` method to insert or update entries:

```
navigator.smartStore.upsertSoupEntries(soupName, entries[], successCallback, errorCallback)
```

where `soupName` is the name of the target soup, and `entries` is an array of one or more entries that match the soup's data structure. The `successCallback` and `errorCallback` parameters function much like the ones for `registerSoup`. However, the success callback for `upsertSoupEntries` indicates that either a new record has been inserted, or an existing record has been updated.

## Upserting with an External ID

If your soup entries mirror data from an external system, you might need to refer to those entities by their ID (primary key) in the external system. For that purpose, we support upsert with an external ID. When you perform an upsert, you can designate any index field as the external ID field. SmartStore will look for existing soup entries with the same value in the designated field with the following results:

- If no field with the same value is found, a new soup entry will be created.
- If the external ID field is found, it will be updated.
- If more than one field matches the external ID, an error will be returned.

When creating a new entry locally, use a regular upsert. Set the external ID field to a value that you can later query when uploading the new entries to the server.

When updating entries with data coming from the server, use the upsert with external ID. Doing so guarantees that you don't end up with duplicate soup entries for the same remote record.

In the following sample code, we chose the value `new` for the `id` field because the record doesn't yet exist on the server. Once we are online, we can query for records that exist only locally (by looking for records where `id == "new"`) and upload them to the server. Once the server returns the actual ID for the records, we can update their `id` fields locally. If you create products that belong to catalogs that have not yet been created on the server, you will be able to capture the relationship with the catalog through the `parentSoupEntryId` field. Once the catalogs are created on the server, update the local records' `parentExternalId` fields.

The following code contains sample scenarios. First, it calls `upsertSoupEntries` to create a new soup entry. In the success callback, the code retrieves the new record with its newly assigned soup entry ID. It then changes the description and calls `forcetk.mobilesdk` methods to create the new account on the server and then update it. The final call demonstrates the upsert with external ID. To make the code more readable, no error callbacks are specified. Also, because all SmartStore calls are asynchronous, real applications should do each step in the callback of the previous step.

```
// Specify data for the account to be created
var acc = {id: "new", Name: "Cloud Inc", Description: "Getting started"};

// Create account in SmartStore
// This upsert does a "create" because the acc has no _soupEntryId field
navigator.smartstore.upsertSoupEntries("accounts", [ acc ], function(accounts) {
    acc = accounts[0];
    // acc should now have a _soupEntryId field (and a _lastModifiedDate as well)
});

// Update account's description in memory
acc["Description"] = "Just shipped our first app ";

// Update account in SmartStore
// This does an "update" because acc has a _soupEntryId field
navigator.smartstore.upsertSoupEntries("accounts", [ acc ], function(accounts) {
   acc = accounts[0];
});

// Create account on server (sync client -> server for entities created locally)
forcetkClient.create("account", {"Name": acc["Name"], "Description": acc["Description"]},
function(result) {
   acc["id"] = result["id"];
   // Update account in SmartStore
   navigator.smartstore.upsertSoupEntries("accounts", [ acc ]);
});

// Update account's description in memory
acc["Description"] = "Now shipping for iOS and Android";

// Update account's description on server
// Sync client -> server for entities existing on server
forcetkClient.update("account", acc["id"], {"Description": acc["Description"]});

///// Later, there is an account (with id: someSfdcId) that you want to get locally

///// There might be an older version of that account in the SmartStore already

// Update account on client
// sync server -> client for entities that might or might not exist on client
forcetkClient.retrieve("account", someSfdcId, "id,Name,Description", function(result) {
   // Create or update account in SmartStore (looking for an account with the same sfdcId)
   navigator.smartstore.upsertSoupEntriesWithExternalId("accounts", [ result ], "id");
});
```

## Removing Soup Entries

Entries are removed from the soup asynchronously and your callback is called with success or failure. The `soupEntryIds` is a list of the `_soupEntryId` values from the entries you wish to delete.

```
navigator.smartStore.removeFromSoup(soupName, soupEntryIds, successCallback, errorCallback)
```

## Removing a Soup

To remove a soup, call `removeSoup()`. Note that once a user signs out, the soups get deleted automatically.

```
navigator.smartstore.removeSoup(soupName,successCallback,errorCallback);
```

# Using the Mock SmartStore

To facilitate developing and testing code that makes use of the SmartStore while running outside the container, you can use an emulated SmartStore. The MockSmartStore is a JavaScript implementation of the SmartStore that stores the data in local storage (or optionally just in memory).

**Note:** The MockSmartStore doesn't encrypt data and is not meant to be used in production applications.

Inside the `PhoneGap` directory, there's a local directory containing the following files:

- `MockCordova.js`—A minimal implementation of Cordova functions meant only for testing plugins outside the container.
- `MockSmartStore.js`—A JavaScript implementation of the SmartStore meant only for development and testing outside the container.
- `MockSmartStorePlugin.js`—A JavaScript helper class that intercepts SmartStore Cordova plugin calls and handles them using a MockSmartStore.
- `CordovaInterceptor.js`—A JavaScript helper class that intercepts Cordova plugin calls.

When writing an application using SmartStore, make the following changes to test your app outside the container:

- Include `MockCordova.js` instead of `cordova-x.x.x.js`.
- Include `MockSmartStore.js` after `cordova.force.js`.

To see a MockSmartStore example, check out `Cordova/local/test.html`.

### Same-origin Policies

Same-origin policy permits scripts running on pages originating from the same site to access each other's methods and properties with no specific restrictions; it also blocks access to most methods and properties across pages on different sites. Same-origin policy restrictions are not an issue when your code runs inside the container, because the container disables same-origin policy in the webview. However, if you call a remote API, you need to worry about same-origin policy restrictions.

Fortunately, browsers offer ways to turn off same-origin policy, and you can research how to do that with your particular browser. If you want to make XHR calls against Force.com from JavaScript files loaded from the local file system, you should start your browser with same-origin policy disabled. The following article describes how to disable same-origin policy on several popular browsers: Getting Around Same-Origin Policy in Web Browsers.

### Authentication

For authentication with MockSmartStore, you will need to capture access tokens and refresh tokens from a real session and hand code them in your JavaScript app. You'll also need these tokens to initialize the `forcetk.mobilesdk` JavaScript toolkit.

# NativeSqlAggregator Sample App: Using SmartStore in Native Apps

The NativeSqlAggregator app demonstrates how to use SmartStore in a native app. It also demonstrates the ability to make complex SQL-like queries, including aggregate functions, such as SUM and COUNT, and joins across different soups within SmartStore.

### Creating a Soup

The first step to storing a Salesforce object in SmartStore is to create a soup for the object. The function call to register a soup takes two arguments - the name of the soup, and the index specs for the soup. Indexing supports three types of data: string,

integer, and floating decimal. The following example illustrates how to initialize a soup for the Account object with indexing on Name, Id, and OwnerId fields.

**Android:**

```
SalesforceSDKManagerWithSmartStore sdkManager =
SalesforceSDKManagerWithSmartStore.getInstance();

SmartStore smartStore = sdkManager.getSmartStore();

IndexSpec[] ACCOUNTS_INDEX_SPEC = {
 new IndexSpec("Name", Type.string),
 new IndexSpec("Id", Type.string),
 new IndexSpec("OwnerId", Type.string)
};

smartStore.registerSoup("Account", ACCOUNTS_INDEX_SPEC);
```

**iOS:**

```
SFSmartStore *store = [SFSmartStore  sharedStoreWithName:kDefaultSmartStoreName];

NSArray *keys = [NSArray arrayWithObjects:@"path", @"type", nil];
NSArray *nameValues = [NSArray arrayWithObjects:@"Name", kSoupIndexTypeString, nil];
NSDictionary *nameDictionary = [NSDictionary dictionaryWithObjects:nameValues
forKeys:keys];
NSArray *idValues = [NSArray arrayWithObjects:@"Id", kSoupIndexTypeString, nil];
NSDictionary *idDictionary = [NSDictionary dictionaryWithObjects:idValues forKeys:keys];
NSArray *ownerIdValues = [NSArray arrayWithObjects:@"OwnerId", kSoupIndexTypeString,
nil];
NSDictionary *ownerIdDictionary = [NSDictionary dictionaryWithObjects:ownerIdValues
forKeys:keys];
NSArray *accountIndexSpecs = [[NSArray alloc] initWithObjects:nameDictionary,
idDictionary, ownerIdDictionary,  nil];

[store registerSoup:@"Account" withIndexSpecs:accountIndexSpecs];
```

## Storing Data in a Soup

Once the soup is created, the next step is to store data in the soup. In the following example, `account` represents a single record of the object Account. On Android, `account` is of type `JSONObject`. On iOS, its type is `NSDictionary`.

**Android:**

```
SmartStore smartStore = sdkManager.getSmartStore();
smartStore.upsert("Account", account);
```

**iOS:**

```
SFSmartStore *store = [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
[store upsertEntries:[NSArray arrayWithObject:account] toSoup:@"Account"];
```

## Running Queries Against SmartStore

Beginning with Mobile SDK 2.0, you can run advanced SQL-like queries against SmartStore that span multiple soups. The syntax of a SmartStore query is similar to standard SQL syntax, with a couple of minor variations. A colon (":") serves as the delimiter between a soup name and an index field. A set of curly braces encloses each `<soup-name>:<field-name>` pair. See Smart SQL Queries on page 108.

Here's an example of an aggregate query run against SmartStore:

```
SELECT {Account:Name},
    COUNT({Opportunity:Name}),
    SUM({Opportunity:Amount}),
    AVG({Opportunity:Amount}), {Account:Id}, {Opportunity:AccountId}
FROM {Account}, {Opportunity}
WHERE {Account:Id} = {Opportunity:AccountId}
GROUP BY {Account:Name}
```

This query represents an implicit join between two soups, Account and Opportunity. It returns:

- Name of the Account
- Number of opportunities associated with an Account
- Sum of all the amounts associated with each Opportunity of that Account
- Average amount associated with an Opportunity of that Account
- Grouping by Account name

The code snippet below demonstrates how to run such queries from within a native app. In this example, `smartSql` is the query and `pageSize` is the requested page size. The `pageIndex` argument specifies which page of results you want returned.

**Android:**

```
QuerySpec querySpec = QuerySpec.buildSmartQuerySpec(smartSql, pageSize);
JSONArray result = smartStore.query(querySpec, pageIndex);
```

**iOS:**

```
SFSmartStore *store = [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
SFQuerySpec *querySpec = [SFQuerySpec newSmartQuerySpec:queryString
withPageSize:pageSize];
NSArray *result = [store queryWithQuerySpec:querySpec pageIndex:pageIndex];
```

# Chapter 8

# Authentication, Security, and Identity in Mobile Apps

**In this chapter ...**



Secure authentication is essential for enterprise applications running on mobile devices. OAuth2 is the industry-standard protocol that allows secure authentication for access to a user's data, without handing out the username and password. It is often described as the valet key of software access: a valet key only allows access to certain features of your car: you cannot open the trunk or glove compartment using a valet key.

Mobile app developers can quickly and easily embed the Salesforce OAuth2 implementation. The implementation uses an HTML view to collect the username and password, which are then sent to the server. A session token is returned and securely stored on the device for future interactions.

A Salesforce *connected app* is the primary means by which a mobile device connects to Salesforce. A connected app gives both the developer and the administrator control over how the app connects and who has access. For example, a connected app can be restricted to certain users, can set or relax an IP range, and so forth.

# OAuth Terminology

**Access Token**

A value used by the consumer to gain access to protected resources on behalf of the user, instead of using the user's Salesforce credentials. The access token is a session ID, and can be used directly.

**Authorization Code**

A short-lived token that represents the access granted by the end user. The authorization code is used to obtain an access token and a refresh token.

**Connected App**

An application external to Salesforce that uses the OAuth protocol to verify both the Salesforce user and the external application. Replaces remote access application.

**Consumer Key**

A value used by the consumer to identify itself to Salesforce. Referred to as `client_id`.

**Refresh Token**

A token used by the consumer to obtain a new access token, without having the end user approve the access again.

**Remote Access Application (DEPRECATED)**

A *remote access application* is an application external to Salesforce that uses the OAuth protocol to verify both the Salesforce user and the external application. Remote access applications have been deprecated in favor of connected apps.

# Creating a Connected App

Before a mobile device can connect with the service, you'll need to create a connected app. The connected app includes a consumer key, a prerequisite to all development scenarios in this guide.

1. Log into your Database.com or Force.com instance.
2. In Setup, navigate to **Create** > **Apps**.
3. Under Connected Apps, click **New**.
4. For `Connected App Name`, enter a name, such as `Test Client`
5. Under Developer Name, enter your developer ID.
6. For `Callback URL`, enter `sfdc://success`

**Note:** The `Callback URL` does not have to be a valid URL; it only has to match what the app expects in this field. You can use any custom prefix, such as `sfdc://`.

7. For `Contact Email`, enter your email address.
8. For `Selected OAuth Scopes`, choose the permissions settings for your app. For descriptions, see Scope Parameter Values.
9. Click **Save**.

**Note:** After you create a new connected app, wait a few minutes for the token to propagate before running your app.

**Tip:** The detail page for your connected app displays a consumer key. It's a good idea to copy the key, as you'll need it later.

# Connected Apps

A Connected App is an application that integrates with salesforce.com using APIs. Connected Apps use standard SAML and OAuth protocols to authenticate, provide Single Sign-On, and provide tokens for use with Salesforce APIs. In addition to standard OAuth capabilities, Connected Apps allow administrators to set various security policies and have explicit control over who may use the applications.

Connected Apps begin with a developer defining OAuth metadata about the application, including:

- Basic descriptive and contact information for the Connected App
- The OAuth scopes and callback URL for the Connected App
- Optional IP ranges where the Connected App might be running
- Optional information about mobile policies the Connected App can enforce

In return, the developer is provided an OAuth client Id and client secret for the Connected App. The developer can then package the app and provide it to a Salesforce administrator.

Administrators can install the Connected App into their organization and use profiles, permission sets, and IP range restrictions to control which users can access the application. Management is done from a detail page for the Connected App. Administrators can also uninstall the Connected App and install a newer version. When the app is updated, the developer can notify administrators that there is a new version available for the app.

**Note:** Salesforce-managed packages like those for the Chatter mobile apps can't be uninstalled. They are automatically updated when the next user's session refreshes.

## About PIN Security

Salesforce Connected Apps have an additional layer of security via PIN protection on the app. This PIN protection is for the mobile app itself, and isn't the same as the PIN protection on the device or the login security provided by the Salesforce organization.

In order to use PIN protection, the developer must select the **Implements Screen Locking & Pin Protection** checkbox when creating the Connected App. Mobile app administrators then have the options of enforcing PIN protection, customizing timeout duration, and setting PIN length.

**Note:** Because PIN security is implemented in the mobile device's operating system, only native and hybrid mobile apps can use PIN protection; HTML5 Web apps can't use PIN protection.

In practice, PIN protection can be used so that the mobile app locks up if it's isn't used for a specified number of minutes. When a mobile app is sent to the background, the clock continues to tick.

To illustrate how PIN protection works:

1. User turns on phone and enters PIN for the device.
2. User starts mobile app (Connected App).
3. User enters login information for Salesforce organization.
4. User enters PIN code for mobile app.
5. User works in the app, then sends it to the background by opening another app (or receiving a call, and so on).

6. The mobile app times out.
7. User re-opens the app, and the app PIN screen displays (for the mobile app, not the device).
8. User enters app PIN and can resume working.

# OAuth2 Authentication Flow

The authentication flow depends on the state of authentication on the device.

### First Time Authentication Flow

1. User opens a mobile application.
2. An authentication dialog/window/overlay appears.
3. User enters username and password.
4. App receives session ID.
5. User grants access to the app.
6. App starts.

### Ongoing Authentication

1. User opens a mobile application.
2. If the session ID is active, the app starts immediately. If the session ID is stale, the app uses the refresh token from its initial authorization to get an updated session ID.
3. App starts.

### PIN Authentication (Optional)

1. User opens a mobile application after not using it for some time.
2. If the elapsed time exceeds the configured PIN timeout value, a passcode entry screen appears. User enters the PIN.

**Note:** PIN protection is a function of the mobile policy and is used only when it's enabled in the Salesforce connected app definition. It can be shown whether you are online or offline, if enough time has elapsed since you last used the application. See About PIN Security on page 118.

3. App re-uses existing session ID.
4. App starts.

# OAuth 2.0 User-Agent Flow

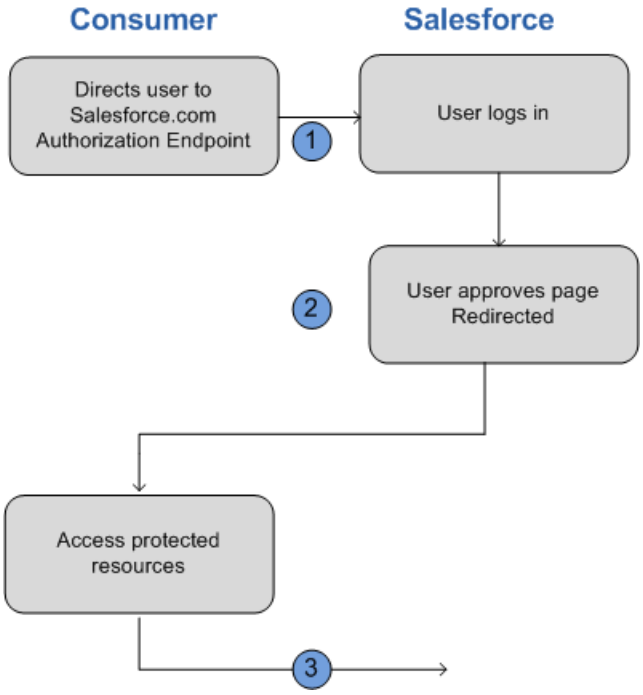
The user-agent authentication flow is used by client applications residing on the user's mobile device. The authentication is based on the user-agent's same-origin policy.

In the user-agent flow, the client application receives the access token in the form of an HTTP redirection. The client application requests the authorization server to redirect the user-agent to another web server or local resource accessible to the user-agent, which is capable of extracting the access token from the response and passing it to the client application. Note that the token response is provided as a hash (#) fragment on the URL. This is for security, and prevents the token from being passed to the server, as well as to other servers in referral headers.

This user-agent authentication flow doesn't utilize the client secret since the client executables reside on the end-user's computer or device, which makes the client secret accessible and exploitable.

**Warning:** Because the access token is encoded into the redirection URI, it might be exposed to the end-user and other applications residing on the computer or device.

If you are authenticating using JavaScript, call `window.location.replace();` to remove the callback from the browser's history.




1. The client application directs the user to Salesforce to authenticate and authorize the application.
2. The user must always approve access for this authentication flow. After approving access, the application receives the callback from Salesforce.

After obtaining an access token, the consumer can use the access token to access data on the end-user's behalf and receive a refresh token. Refresh tokens let the consumer get a new access token if the access token becomes invalid for any reason.

## OAuth 2.0 Refresh Token Flow

After the consumer has been authorized for access, they can use a refresh token to get a new access token (session ID.) This is only done after the consumer already has received a refresh token using either the Web server or user-agent flow. It is up to the consumer to determine when an access token is no longer valid, and when to apply for a new one. Bearer flows can only be used after the consumer has received a refresh token.

The following are the steps for the refresh token authentication flow. More detail about each step follows:

1. The consumer uses the existing refresh token to request a new access token.
2. After the request is verified, Salesforce sends a response to the client.

**Note:**

Mobile SDK apps can use the SmartStore feature to store data locally for offline use. SmartStore data is inherently volatile. Its lifespan is tied to the authenticated user as well as to OAuth token states. When the user logs out of the app, SmartStore deletes all soup data associated with that user. Similarly, when the OAuth refresh token is revoked or expires, the user's app state is reset, and all data in SmartStore is purged. Carefully consider the volatility of

SmartStore data when designing your app. This warning is especially important if your org sets a short lifetime for the refresh token.

## Scope Parameter Values

The `scope` parameter enables you to fine-tune what the client application can access in a Salesforce organization. The valid values for `scope` are:

| Value | Description |
|---|---|
| `api` | Allows access to the current, logged-in user's account using APIs, such as REST API and Bulk API. This value also includes `chatter_api`, which allows access to Chatter REST API resources. |
| `chatter_api` | Allows access to Chatter REST API resources only. |
| `full` | Allows access to all data accessible by the logged-in user. `full` does not return a refresh token. You must explicitly request the `refresh_token` scope to get a refresh token. |
| `id` | Allows access only to the identity URL service. |
| `refresh_token` | Allows a refresh token to be returned if you are eligible to receive one. |
| `visualforce` | Allows access to Visualforce pages. |
| `web` | Allows the ability to use the `access_token` on the Web. This also includes `visualforce`, allowing access to Visualforce pages. |

## Using Identity URLs

In addition to the access token, an identity URL is also returned as part of a token response, in the `id` parameter.

The identity URL is both a string that uniquely identifies a user, as well as a RESTful API that can be used to query (with a valid access token) for additional information about the user. Salesforce returns basic personalization information about the user, as well as important endpoints that the client can talk to, such as photos for the user, and API endpoints it can access.

The format of the URL is: `https://login.salesforce.com/id/`*`orgID`*`/`*`userID`*, where *`orgId`* is the ID of the Salesforce organization that the user belongs to, and *`userID`* is the Salesforce user ID.

**Note:** For a sandbox, `login.salesforce.com` is replaced with `test.salesforce.com`.

The URL must always be HTTPS.

### Identity URL Parameters

The following parameters can be used with the access token and identity URL. They are used in an authorization request header or in a request with the `oauth_token` parameter. For more details, see "Using the Access Token" in the Salesforce Help.

| Parameter | Description |
|---|---|
| Access token | See "Using the Access Token" in the Salesforce Help. |

| Parameter | Description |
| --- | --- |
| Format | This parameter is optional. Specify the format of the returned output. Valid values are:<br>• `urlencoded`<br>• `json`<br>• `xml`<br>Instead of using the `format` parameter, the client can also specify the returned format in an accept-request header using one of the following:<br>• `Accept: application/json`<br>• `Accept: application/xml`<br>• `Accept: application/x-www-form-urlencoded`<br>Note the following:<br>• Wildcard accept headers are allowed. `*/*` is accepted and returns JSON.<br>• A list of values is also accepted and is checked left-to-right. For example: `application/xml,application/json,application/html,*/*` returns XML.<br>• The `format` parameter takes precedence over the accept request header. |
| Version | This parameter is optional. Specify a SOAP API version number, or the literal string, `latest`. If this value isn't specified, the returned API URLs contains the literal value `{version}`, in place of the version number, for the client to do string replacement. If the value is specified as `latest`, the most recent API version is used. |
| PrettyPrint | This parameter is optional, and is only accepted in a header, not as a URL parameter. Specify the output to be better formatted. For example, use the following in a header: `X-PrettyPrint:1`. If this value isn't specified, the returned XML or JSON is optimized for size rather than readability. |
| Callback | This parameter is optional. Specify a valid JavaScript function name. This parameter is only used when the format is specified as JSON. The output is wrapped in this function name (JSONP.) For example, if a request to `https://server/id/orgid/userid/` returns `{"foo":"bar"}`, a request to `https://server/id/orgid/userid/?callback=baz` returns `baz({"foo":"bar"});`. |

## Identity URL Response

After making a valid request, a **302 redirect** to an instance URL is returned. That subsequent request returns the following information in JSON format:

- `id`—The identity URL (the same URL that was queried)
- `asserted_user`—A boolean value, indicating whether the specified access token used was issued for this identity
- `user_id`—The Salesforce user ID

- `username`—The Salesforce username
- `organization_id`—The Salesforce organization ID
- `nick_name`—The community nickname of the queried user
- `display_name`—The display name (full name) of the queried user
- `email`—The email address of the queried user
- `status`—The user's current Chatter status.
  - `created_date`:`xsd datetime` value of the creation date of the last post by the user, for example, 2010-05-08T05:17:51.000Z
  - `body`: the body of the post
- `photos`—A map of URLs to the user's profile pictures

  **Note:** Accessing these URLs requires passing an access token. See "Using the Access Token" in the Salesforce Help.

  - `picture`
  - `thumbnail`
- `urls`—A map containing various API endpoints that can be used with the specified user.

  **Note:** Accessing the REST endpoints requires passing an access token. See "Using the Access Token" in the Salesforce Help.

  - `enterprise` (SOAP)
  - `metadata` (SOAP)
  - `partner` (SOAP)
  - `profile`
  - `feeds` (Chatter)
  - `feed-items` (Chatter)
  - `groups` (Chatter)
  - `users` (Chatter)
  - `custom_domain`—This value is omitted if the organization doesn't have a custom domain configured and propagated
- `active`—A boolean specifying whether the queried user is active
- `user_type`—The type of the queried user
- `language`—The queried user's language
- `locale`—The queried user's locale
- `utcOffset`—The offset from UTC of the timezone of the queried user, in milliseconds
- `last_modified_date`—`xsd datetime` format of last modification of the user, for example, 2010-06-28T20:54:09.000Z

The following is a response in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<user xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<id>http://na1.salesforce.com/id/00Dx0000001T0zk/005x0000001S2b9</id>
<asserted_user>true</asserted_user>
<user_id>005x0000001S2b9</user_id>
<organization_id>00Dx0000001T0zk</organization_id>
<nick_name>admin1.2777578168398293E12foofoofoofoo</nick_name>
<display_name>Alan Van</display_name>
<email>admin@2060747062579699.com</email>
<status>
   <created_date xsi:nil="true"/>
   <body xsi:nil="true"/>
</status>
```

```
<photos>
   <picture>http://na1.salesforce.com/profilephoto/005/F</picture>
   <thumbnail>http://na1.salesforce.com/profilephoto/005/T</thumbnail>
</photos>
<urls>
   <enterprise>http://na1.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk
   </enterprise>
   <metadata>http://na1.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk
   </metadata>
   <partner>http://na1.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk
   </partner>
   <rest>http://na1.salesforce.com/services/data/v{version}/
   </rest>
   <sobjects>http://na1.salesforce.com/services/data/v{version}/sobjects/
   </sobjects>
   <search>http://na1.salesforce.com/services/data/v{version}/search/
   </search>
   <query>http://na1.salesforce.com/services/data/v{version}/query/
   </query>
   <profile>http://na1.salesforce.com/005x0000001S2b9
   </profile>
</urls>
<active>true</active>
<user_type>STANDARD</user_type>
<language>en_US</language>
<locale>en_US</locale>
<utcOffset>-28800000</utcOffset>
<last_modified_date>2010-06-28T20:54:09.000Z</last_modified_date>
</user>
```

The following is a response in JSON format:

```
{"id":"http://na1.salesforce.com/id/00Dx0000001T0zk/005x0000001S2b9",
"asserted_user":true,
"user_id":"005x0000001S2b9",
"organization_id":"00Dx0000001T0zk",
"nick_name":"admin1.2777578168398293E12foofoofoofoo",
"display_name":"Alan Van",
"email":"admin@2060747062579699.com",
"status":{"created_date":null,"body":null},
"photos":{"picture":"http://na1.salesforce.com/profilephoto/005/F",
   "thumbnail":"http://na1.salesforce.com/profilephoto/005/T"},
"urls":
   {"enterprise":"http://na1.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk",
   "metadata":"http://na1.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk",
   "partner":"http://na1.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk",
   "rest":"http://na1.salesforce.com/services/data/v{version}/",
   "sobjects":"http://na1.salesforce.com/services/data/v{version}/sobjects/",
   "search":"http://na1.salesforce.com/services/data/v{version}/search/",
   "query":"http://na1.salesforce.com/services/data/v{version}/query/",
   "profile":"http://na1.salesforce.com/005x0000001S2b9"},
"active":true,
"user_type":"STANDARD",
"language":"en_US",
"locale":"en_US",
"utcOffset":-28800000,
"last_modified_date":"2010-06-28T20:54:09.000+0000"}
```

After making an invalid request, the following are possible responses from Salesforce:

| Request Problem | Error Code |
|---|---|
| HTTP | 403 (forbidden) — HTTPS_Required |
| Missing access token | 403 (forbidden) — Missing_OAuth_Token |
| Invalid access token | 403 (forbidden) — Bad_OAuth_Token |

| Request Problem | Error Code |
|---|---|
| Users in a different organization | 403 (forbidden) — Wrong_Org |
| Invalid or bad user or organization ID | 404 (not found) — Bad_Id |
| Deactivated user or inactive organization | 404 (not found) — Inactive |
| User lacks proper access to organization or information | 404 (not found) — No_Access |
| Request to the endpoint of a site | 404 (not found) — No_Site_Endpoint |
| Invalid version | 406 (not acceptable) — Invalid_Version |
| Invalid callback | 406 (not acceptable) — Invalid_Callback |

# Setting a Custom Login Server

For special cases--for example, if you're a Salesforce partner using Trialforce--you might need to redirect your customer login requests to a non-standard login URI. For iOS apps, you set the Custom Host in your app's iOS settings bundle. If you've configured this setting, it will be used as the default connection.

In Android, login hosts are known as server connections. Prior to Mobile SDK v. 1.4, server connections for Android apps were hard-coded in the SalesforceSDK project. In v. 1.4 and later, the host list is defined in the `res/xml/servers.xml` file. The SalesforceSDK library project uses this file to define production and sandbox servers.

You can add your servers to the runtime list by creating your own `res/xml/servers.xml` file in your application project. The root XML element for this file is `<servers>`. This root can contain any number of `<server>` entries. Each `<server>` entry requires two attributes: `name` (an arbitrary human-friendly label) and `url` (the web address of the login server.)

Here's an example of a servers.xml file.

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
  <server name="XYZ.com Login" url="https://<username>.cloudforce.com"/>
</servers>
```

## Server Whitelisting Errors

If you get a whitelist rejection error, you'll need to add your custom login domain to the `ExternalHosts` list for your project. This list is defined in the `<project_name>/<platform_path>/config.xml` file. Add those domains (e.g. cloudforce.com) to the app's whitelist in the following files:

For Mobile SDK 2.0:

- **iOS:** `/Supporting Files/config.xml`
- **Android:** `/res/xml/config.xml`

# Revoking OAuth Tokens

When a user logs out of an app, or the app times out or in other ways becomes invalid, the logged-in users' credentials are cleared from the mobile app. This effectively ends the connection to the server. Also, Mobile SDK revokes the refresh token from the server as part of logout.

### Revoking Tokens

To revoke OAuth 2.0 tokens, use the revocation endpoint:

```
https://login.salesforce.com/services/oauth2/revoke
```

Construct a POST request that includes the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body. For example:

```
POST /revoke HTTP/1.1
Host: https://login.salesforce.com/services/oauth2/revoke
Content-Type: application/x-www-form-urlencoded

token=currenttoken
```

If an access token is included, we invalidate it and revoke the token. If a refresh token is included, we revoke it as well as any associated access tokens.

The authorization server indicates successful processing of the request by returning an HTTP status code 200. For all error conditions, a status code 400 is used along with one of the following error responses.

- `unsupported_token_type`—token type not supported
- `invalid_token`—the token was invalid

For a sandbox, use `test.salesforce.com` instead of `login.salesforce.com`.

## Handling Refresh Token Revocation in Android Native Apps

Beginning with Salesforce Mobile SDK version 1.5, native Android apps can control what happens when a refresh token is revoked by an administrator. The default behavior in this case is to automatically log out the current user. As a result of this behavior:

- Any subsequent REST API calls your app makes will fail.
- The system discards your user's account information and cached offline data.
- The system forces the user to navigate away from your page.
- The user must log into Salesforce again to continue using your app.

These side effects provide a secure response to the administrator's action, but they might or might not be suitable for your application. In your code you can choose whether to accept the default behavior or implement your own response. In either case, continue reading to determine whether you need to adapt your code.

### Token Revocation Events

When a token revocation event occurs, the `ClientManager` object sends an Android-style notification. The intent action for this notification is declared in the `ClientManager.ACCESS_TOKEN_REVOKE_INTENT` constant. `TokenRevocationReceiver`, a utility class, is designed to respond to this intent action. To provide your own handler, you'll extend this class and override the `onReceive()` method. See Token Revocation: Active Handling.

`SalesforceActivity.java`, `SalesforceListActivity.java`, `SalesforceExpandableListActivity.java`, and `SalesforceDroidGapActivity.java` implement `ACCESS_TOKEN_REVOKE_INTENT` event listeners. These listeners automatically take logged out users to the login page when the refresh token is revoked. A toast message notifies the user of this occurrence.

## Token Revocation: Passive Handling

You can let the SDK handle all token revocation events with no active involvement on your part. However, even if you take this passive approach, you might still need to change your code. You do not need to change your code if:

- Your app contains any services, or
- All of your activities extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`.

If your app fails to satisfy at least one of these conditions, implement the following code changes.

1. (For legacy apps written before the Mobile SDK 1.5 release) In the `ClientManager` constructor, set the `revokedTokenShouldLogout` parameter to true.

   **Note:** This step is not necessary for apps that are new in Mobile SDK 1.5 or later.

2. In any activity that does not extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`, amend the code as follows.

   a. Declare a new variable:

   ```
   private TokenRevocationReceiver tokenRevocationReceiver;
   ```

   b. In the `onCreate()` method add the following code:

   ```
   tokenRevocationReceiver = new TokenRevocationReceiver(this);
   ```

   c. In the `onResume()` method add the following code:

   ```
   registerReceiver(tokenRevocationReceiver, new
   IntentFilter(ClientManager.ACCESS_TOKEN_REVOKE_INTENT));
   ```

   d. In the `onPause()` method add the following code:

   ```
   unregisterReceiver(tokenRevocationReceiver);
   ```

## Token Revocation: Active Handling

If you choose to implement your own token revocation event handler, be sure to fully analyze the security implications of your customized flow, and then test it thoroughly. Be especially careful with how you dispose of cached user data. Because the user's access has been revoked, that user should no longer have access to sensitive data.

To provide custom handling of token revocation events:

1. The starting point for implementing your own response is the `SalesforceSDKManager.shouldLogoutWhenTokenRevoked()` method. By default, this method returns true. Override this method to return false in your `SalesforceSDKManager` subclass.

   ```
   @Override
   public boolean shouldLogoutWhenTokenRevoked() {
    return false;
   }
   ```

2. The `ClientManager` constructor provides a boolean parameter, `revokedTokenShouldLogout`. Set this parameter to false. You can do this by calling `shouldLogoutWhenTokenRevoked()` on your `SalesforceSDKManager` subclass.
3. Implement your handler by extending `TokenRevocationReceiver` and overriding the `onReceive()` method.
4. Regardless of whether your activity subclasses `SalesforceActivity`, perform step 2 in Token Revocation: Passive Handling on page 127.

# Portal Authentication Using OAuth 2.0 and Force.com Sites

The Salesforce Spring '13 Release adds enhanced flexibility for portal authentication. If your app runs in a Salesforce portal, you can use OAuth 2.0 with a Force.com site to obtain API access tokens on behalf of portal users. In this configuration you can

- Authenticate portal users via Auth providers and SAML, rather than a SOAP API `login()` call
- Avoid handling user credentials in your app
- Customize the login screen provided by the Force.com site

Here's how to get started.

1. Associate a Force.com site with your portal. The site generates a unique URL for your portal. See Associating a Portal with Force.com Sites.
2. Create a custom login page on the Force.com site. See Managing Force.com Site Login and Registration Settings.
3. Use the unique URL that the site generates as the redirect domain for your users' login requests.

The OAuth2 service recognizes your custom host name and redirects the user to your Site login page if the user is not yet authenticated.

For example, rather than redirecting to `https://login.salesforce.com`:

```
    https://login.salesforce.com/services/oauth2/authorize?response_type=
        code&client_id=<your_client_id>&redirect_uri=<your_redirect_uri>
```

redirect to your unique Force.com Site url, such as `https://mysite.secure.force.com`:

```
    https://mysite.secure.force.com/services/oauth2/authorize?response_type=
        code&client_id=<your_client_id>&redirect_uri=<your_redirect_uri>
```

For more information and a demonstration video, see OAuth for Portal Users on the Force.com Developer Relations Blogs page.

# Chapter 9

# Migrating from the Previous Release

**In this chapter ...**



If you developed code with Salesforce Mobile SDK 1.5, follow these instructions to update your app to version 2.0.

# Migrating Android Applications

Perform these tasks to upgrade your Android applications from Salesforce Mobile SDK 1.5.3 to version 2.0.0.

## Upgrading Native Android Apps

- In your app's Eclipse workspace, replace the existing SalesforceSDK project with the 2.0 SalesforceSDK project. If your app uses SmartStore, replace the existing SmartStore project in Eclipse with the 2.0 SmartStore project.
  1. Right-click your project and select **Properties**.
  2. Click the `Android` tab and replace the existing SalesforceSDK entry at the bottom (in the library project section) with the new SalesforceSDK project in your workspace. Repeat this step with the SmartStore project if your app uses SmartStore.
- Change your class that extends `ForceApp` or `ForceAppWithSmartStore` to extend `Application` instead. We'll call this class `SampleApp` in the remaining steps.
- Create a new class that implements `KeyInterface`. Name it `KeyImpl` (or another name of your choice.) Move the `getKey()` implementation from `SampleApp` into `KeyImpl`.
- We've renamed `ForceApp` to `SalesforceSDKManager` and `ForceAppWithSmartStore` to `SalesforceSDKManagerWithSmartStore`.
  - ◊ Replace all occurrences of `ForceApp` with `SalesforceSDKManager`
  - ◊ Replace all occurrences of `ForceAppWithSmartStore` with `SalesforceSDKManagerWithSmartStore`.
  - ◊ Update the app's class imports to reflect this change.
  - ◊ Replace all occurrences of `ForceApp.APP` with `SalesforceSDKManager.getInstance()`.
  - ◊ Replace all occurrences of `ForceAppWithSmartStore.APP` with `SalesforceSDKManagerWithSmartStore.getInstance()`.
- In the `onCreate()` method of `SampleApp`, add the following line of code.

```
SalesforceSDKManager.initNative(getApplicationContext(), new KeyImpl(),
<mainActivityClass>.class);
```

  where `<mainActivityClass>` is the class to be launched when the login flow completes.

**Note:**

- ◊ If your app supplies its own login activity, you can pass it as an additional argument to the `initNative()` method call.
- ◊ If your app uses SmartStore, call `initNative()` on `SalesforceSDKManagerWithSmartStore` instead of `SalesforceSDKManager`.

- Remove overridden methods of `ForceApp` from `SampleApp`, such as `getKey()`, `getMainActivityClass()`, and any other overridden methods.
- You're no longer required to create a `LoginOptions` object. The Salesforce Mobile SDK now automatically reads these options from an XML file, `bootconfig.xml`, which resides in the `res/values` folder of your project.
  - ◊ Create a file called `bootconfig.xml` under the `res/values` folder of your project. Move your app's login options configuration from code to `bootconfig.xml`. See `res/values/bootconfig.xml` in the SalesforceSDK project or in one of the sample native apps for an example.

- `NativeMainActivity` has been renamed to `SalesforceActivity` and moved to a new package named `com.salesforce.androidsdk.ui.sfnative`.
  - ◊ If any of your app's classes extend `NativeMainActivity`, replace all references to `NativeMainActivity` with `SalesforceActivity`.
  - ◊ Update the app's class imports to reflect this change.
- We've moved SmartStore to a new package named `com.salesforce.androidsdk.smartstore`. If your app uses SmartStore project, update the app's class imports and other code references to reflect this change.

## Upgrading Hybrid Android Apps

- In your app's Eclipse workspace, replace the existing SalesforceSDK project with the 2.0 SalesforceSDK project. If your app uses SmartStore, replace the existing SmartStore project in Eclipse with the 2.0 SmartStore project.
  1. Right-click your project and select **Properties**.
  2. Click the `Android` tab and replace the existing SalesforceSDK entry at the bottom (in the library project section) with the new SalesforceSDK project in your workspace. Repeat this step with the SmartStore project if your app uses SmartStore.
- Change your class that extends `ForceApp` or `ForceAppWithSmartStore` to extend `Application` instead. We'll call this class `SampleApp` in the remaining steps.
- Create a new class that implements `KeyInterface`. Name it `KeyImpl` (or any other name of your choice.) Move the `getKey()` implementation from `SampleApp` into `KeyImpl`.
- We've renamed `ForceApp` to `SalesforceSDKManager` and `ForceAppWithSmartStore` to `SalesforceSDKManagerWithSmartStore`.
  - ◊ Replace all occurrences of `ForceApp` with `SalesforceSDKManager`
  - ◊ Replace all occurrences of `ForceAppWithSmartStore` with `SalesforceSDKManagerWithSmartStore`.
  - ◊ Update the app's class imports to reflect this change.
  - ◊ Replace all occurrences of `ForceApp.APP` with `SalesforceSDKManager.getInstance()`.
  - ◊ Replace all occurrences of `ForceAppWithSmartStore.APP` with `SalesforceSDKManagerWithSmartStore.getInstance()`.
- In the `onCreate()` method of `SampleApp`, add the following line of code.

```
SalesforceSDKManager.initHybrid(getApplicationContext(), new KeyImpl());
```

**Note:**

- ◊ If your app supplies its own login activity, you can pass it as an additional argument to the `initHybrid()` method call.
- ◊ If your app uses SmartStore, call `initHybrid()` on `SalesforceSDKManagerWithSmartStore` instead of `SalesforceSDKManager`.

- Remove overridden methods of `ForceApp` from `SampleApp`, such as `getKey()`, `getMainActivityClass()`, and any other overridden methods.
- You're no longer required to create a `LoginOptions` object. The Salesforce Mobile SDK now automatically reads these options from an XML file, `bootconfig.xml`, which resides in the `res/values` folder of your project.

    - Create a file called `bootconfig.xml` under the `res/values` folder of your project. Move your app's login options configuration from code to `bootconfig.xml`. See `res/values/bootconfig.xml` in the SalesforceSDK project or in one of the sample native apps for an example.

- `NativeMainActivity` has been renamed to `SalesforceActivity` and moved to a new package named `com.salesforce.androidsdk.ui.sfnative`.
    - If any of your app's classes extend `NativeMainActivity`, replace all references to `NativeMainActivity` with `SalesforceActivity`.
    - Update the app's class imports to reflect this change.
- We've moved SmartStore to a new package named `com.salesforce.androidsdk.smartstore`. If your app uses the SmartStore project, update the app's class imports and other code references to reflect this change.
- We've replaced `bootconfig.js` with `bootconfig.json`. Convert your existing `bootconfig.js` to the new `bootconfig.json` format. See the hybrid sample apps for examples.
- The SalesforceSDK Cordova plugins—`SFHybridApp.js`, `cordova.force.js`, and `SalesforceOAuthPlugin.js`—have been combined into a single file named `filecordova.force.js`.
    - Replace these Cordova plugin files with `cordova.force.js`.
    - Replace all references to `SFHybridApp.js`, `cordova.force.js`, and `SalesforceOAuthPlugin.js` with `cordova.force.js`.
- `forcetk.js` has now been renamed to `forcetk.mobilesdk.js`. Replace the existing copy of `forcetk.js` with the latest version of `forcetk.mobilesdk.js`. Update all references to `forcetk.js` to the new name.
- The `bootstrap.html` file is no longer required and can safely be removed.
- We've moved `SalesforceDroidGapActivity` and `SalesforceGapViewClient` to a new package named `com.salesforce.androidsdk.ui.sfhybrid`. If your app references these classes, update those references and related class imports.

# Migrating iOS Applications

Perform these tasks to upgrade your iOS applications from Salesforce Mobile SDK 1.5 to version 2.0.

## Upgrading Native iOS Apps

As with all upgrades, you have two choices for upgrading your existing app:

- Create a new project using the Mobile SDK 2.0 template app for your app type (native, hybrid), then move your existing code and artifacts into the new app.
- Incorporate Mobile SDK 2.0 artifacts into your existing app.

For 2.0, we strongly recommend that you take the first approach. Even if you opt for the second approach, you can profit from creating a sample app to see the change of work flow in the `AppDelegate` class. For both native and hybrid cases, the parent app delegate classes—`SFNativeRestAppDelegate` and `SFContainerAppDelegate`, respectively—are no longer supported. Your app's `AppDelegate` class now orchestrates the startup process.

- Remove `SalesforceHybridSDK.framework`, which has been replaced.
- Update your Mobile SDK library and resource dependencies, from the SalesforceMobileSDK-iOS-Package repo.
    - Remove `SalesforceSDK`
    - Add `SalesforceNativeSDK` (in the `Dependencies/` folder)
    - Add `SalesforceSDKCore` (in the `Dependencies/` folder)

- ◊ Update `SalesforceOAuth` (in the `Dependencies/` folder)
- ◊ Update `SalesforceSDKResources.bundle` (in the `Dependencies/` folder)
- ◊ Update `RestKit` (in the `Dependencies/ThirdParty/RestKit/` folder)
- ◊ Update `SalesforceCommonUtils` (in the `Dependencies/ThirdParty/SalesforceCommonUtils` folder)
- ◊ Update openssl (`libcrypto.a` and `libssl.a`, in the `Dependencies/ThirdParty/openssl` folder)
- ◊ Update sqlcipher (in the `Dependencies/ThirdParty/sqlcipher` folder)

- Update your `AppDelegate` class. Make your `AppDelegate.h` and `AppDelegate.m` files conform to the new design patterns. Here are some key points:
  - ◊ In `AppDelegate.h`, `AppDelegate` should no longer inherit from `SFNativeRestAppDelegate`.
  - ◊ In `AppDelegate.m`, `AppDelegate` now has primary responsibility for navigating the auth flow and root view controller staging. It also handles boundary events when the user logs out or switches login hosts.

    **Note:** The design patterns in the new `AppDelegate` are just suggestions. Mobile SDK no longer requires a specific flow. Use an authentication flow (with the updated `SFAuthenticationManager` singleton) that suits your needs, relative to your app startup and boundary use cases.)

  - ◊ The only prerequisites for using authentication are the `SFAccountManager` configuration settings at the top of `[AppDelegate init]`. Make sure that those settings match the values specified in your connected app. Also, make sure that this configuration is set before the first call to `[SFAuthenticationManager loginWithCompletion:failure:]`.

## Upgrading Hybrid iOS Apps

In Mobile SDK 2.0, hybrid configuration during bootstrap moves to native code. Take a look at `SFHybridViewController` to see the new configuration. (You can also see this change in `AppDelegate` in the hybrid template app.)

New app templates are now available through the forceios NPM package. To install the templates, first install node.js. See the forceios `README` at npmjs.org for more information on installing the templates and using them to create apps.

Even if you're not porting your previous contents into a 2.0 application shell, it's still a good idea to create a new hybrid app from the template and follow along.

- Remove `SalesforceHybridSDK.framework`. We've replaced this project.
- Update your Mobile SDK library and resource dependencies from the SalesforceMobileSDK-iOS-Package repo. The following modules are new additions to your Mobile SDK 1.5 application.
  - ◊ `SalesforceHybridSDK` (in the `Dependencies/` folder)
  - ◊ `SalesforceOAuth` (in the `Dependencies/` folder)
  - ◊ `SalesforceSDKCore` (in the `Dependencies/` folder)
  - ◊ `SalesforceSDKResources.bundle` (in the `Dependencies/` folder)
  - ◊ Cordova (in the `Dependencies/Cordova/` folder)
  - ◊ `SalesforceCommonUtils` (in the `Dependencies/ThirdParty/SalesforceCommonUtils` folder)
  - ◊ openssl (`libcrypto.a` and `libssl.a`, in the `Dependencies/ThirdParty/openssl` folder)
  - ◊ sqlcipher (in the `Dependencies/ThirdParty/sqlcipher` folder)
  - ◊ `libxml2.dylib` (System library)
- Update hybrid dependencies in your app's `www/` folder.

**Note:** If you're updating a Visualforce app, only the `bootconfig.js` change is required. Your hybrid app does not use the other files.

- Migrate your `bootconfig.js` configuration to the new `bootconfig.json` format.
- Remove `SalesforceOAuthPlugin.js`, `SFHybridApp.js`, `cordova.force.js`, and `forcetk.js`.
- If you're not using them, you can remove `SFTestRunnerPlugin.js`, `qunit.css`, and `qunit.js`.
- Add `cordova.force.js` (in the `HybridShared/libs/` folder).
- If you're using forceTk, add `forcetk.mobilesdk.js` (in the `HybridShared/libs/` folder).
- If you're using jQuery, update jQuery (in the `HybridShared/external/` folder).
- Add `SmartSync.js` (in the `HybridShared/libs/` folder).
- Add `backbone-1.0.0.min.js` and `underscore-1.4.4.min.js` (in the `HybridShared/external/backbone/` folder).
- Add jQuery if you haven't already (in the `HybridShared/external/jquery/` folder).
- If you'd like to use the new SmartSync Data Framework:
  - Add `SmartSync.js` (in the `HybridShared/libs/` folder).
  - Add `backbone-1.0.0.min.js` and `underscore-1.4.4.min.js` (in the `HybridShared/external/backbone/` folder).
  - If you haven't already, add jQuery, (in the `HybridShared/external/jquery/` folder).

- Update your `AppDelegate`—Make your `AppDelegate.h` and `AppDelegate.m` files conform to the new design patterns. If you've never changed your `AppDelegate` class, you can simply copy the new template app's `AppDelegate.h` and `AppDelegate.m` files over the old ones. Here are some key points:
  - In `AppDelegate.h`:
    - `AppDelegate` no longer inherits `SFContainerAppDelegate`.
    - There's a new `viewController` property on `SFHybridViewController`.
  - In `AppDelegate.m`, `AppDelegate` now assumes primary responsibility for navigating the bootstrapping and authentication flow. This responsibility includes handling boundary events when the user logs out or switches login hosts.

# Chapter 10

# Reference

**In this chapter ...**



Reference documentation is hosted on GitHub

- For iOS: http://forcedotcom.github.com/SalesforceMobileSDK-iOS/Documentation/SalesforceSDK/index.html
- For Android: http://forcedotcom.github.com/SalesforceMobileSDK-Android/index.html

# REST API Resources

The Salesforce Mobile SDK simplifies using the REST API by creating wrappers. All you need to do is call a method and provide the correct parameters; the rest is done for you. This table lists the resources available and what they do. For more information, see the REST API Developer's Guide.

| Resource Name | URI | Description |
|---|---|---|
| Versions | `/` | Lists summary information about each Salesforce version currently available, including the version, label, and a link to each version's root. |
| Resources by Version | `/vXX.X/` | Lists available resources for the specified API version, including resource name and URI. |
| Describe Global | `/vXX.X/sobjects/` | Lists the available objects and their metadata for your organization's data. |
| SObject Basic Information | `/vXX.X/sobjects/`***`SObject`***`/` | Describes the individual metadata for the specified object. Can also be used to create a new record for a given object. |
| SObject Describe | `/vXX.X/sobjects/`***`SObject`***`/describe/` | Completely describes the individual metadata at all levels for the specified object. |
| SObject Rows | `/vXX.X/sobjects/`***`SObject`***`/`***`id`***`/` | Accesses records based on the specified object ID. Retrieves, updates, or deletes records. This resource can also be used to retrieve field values. |
| SObject Rows by External ID | `/vXX.X/sobjects/`***`SObjectName/fieldName/fieldValue`*** | Creates new records or updates existing records (upserts records) based on the value of a specified external ID field. |
| SObject User Password | `/vXX.X/sobjects/User/`***`user id`***`/password`<br>`/vXX.X/sobjects/SelfServiceUser/`***`self service user id`***`/password` | Set, reset, or get information about a user password. |
| Query | `/vXX.X/query/?q=`***`soql`*** | Executes the specified SOQL query. |
| Search | `/vXX.X/search/?s=`***`sosl`*** | Executes the specified SOSL search. The search string must be URL-encoded. |

# iOS Architecture

At a high level, the current facilities that the native SDK provides to consumers are:

- OAuth authentication capabilities
- REST API communication capabilities
- SmartStore secure storage and retrieval of app data

**Note:** SmartStore is not currently exposed to native template apps, but is included in the binary distribution.

The Salesforce native SDK is essentially one library, with dependencies on (and providing exposure to) the following additional libraries:

- `libRestKit.a` — Third-party underlying libraries for facilitating REST API calls.
  - RestKit in turn depends on `libxml2.dylib`, which is part of the standard iOS development environment
- `libSalesforceOAuth.a` — Underlying libraries for managing OAuth authentication.
- `libsqlite3.dylib` — Library providing access to SQLite capabilities. This is also a part of the standard iOS development environment.
- `fmdb` — Objective-C wrapper around SQLite.

**Note:** This wrapper is not currently exposed to native template apps, but is included in the binary distribution.

# Native iOS Objects

The following objects let you access Salesforce data in your native app:

- `SFRestAPI`
- `SFRestAPI` (Blocks)
- `SFRestRequest`

### `SFRestAPI`

`SFRestAPI` is the entry point for making REST requests, and is generally accessed as a singleton, via `[SFRestAPI sharedInstance]`.

You can easily create many standard canned queries from this object, such as:

```
SFRestRequest* request = [[SFRestAPI sharedInstance] requestForUpdateWithObjectType:@"Contact"

    objectId:contactId
    fields:updatedFields];
```

You can then initiate the request with the following:

```
[[SFRestAPI sharedInstance] send:request delegate:self];
```

### `SFRestAPI` (Blocks)

This is a category extension of the `SFRestAPI` class that allows you to specify blocks as your callback mechanism. For example:

```
NSMutableDictionary *fields = [NSMutableDictionary dictionaryWithObjectsAndKeys:
    @"John", @"FirstName",
    @"Doe", @"LastName",
    nil];
[[SFRestAPI sharedInstance] performCreateWithObjectType:@"Contact"
    fields:fields
    failBlock:^(NSError *e) {
    NSLog(@"Error: %@", e);
    }
```

```
    completeBlock:^(NSDictionary *d) {
        NSLog(@"ID value for object: %@", [d objectForKey:@"id"]);
    }];
```

#### SFRestRequest

In addition to the canned REST requests provided by SFRestAPI, you can also create your own:

```
NSString *path = @"/v23.0";
SFRestRequest* request = [SFRestRequest requestWithMethod:SFRestMethodGET path:path
queryParams:nil];
```

#### SFRestAPI (QueryBuilder)

This category extension provides utility methods for creating SOQL and SOSL query strings. Examples:

```
NSString *soqlQuery =
 [SFRestAPI SOQLQueryWithFields:[NSArray arrayWithObjects:@"Id", @"Name", @"Company",
@"Status", nil]
                             sObject:@"Lead"
                               where:nil
                               limit:10];

NSString *soslQuery =
 [SFRestAPI SOSLSearchWithSearchTerm:@"all of these will be escaped:~{]"
         objectScope:[NSDictionary dictionaryWithObject:@"WHERE isactive=true ORDER BY
lastname
                   asc limit 5"
                    forKey:@"User"]];
```

### Other Objects

Though you won't likely leverage these objects directly, their purpose in the SDK is worth noting.

- `RKRequestDelegateWrapper`—The intermediary between `SFRestAPI` and the `RestKit` libraries. `RKRequestDelegateWrapper` wraps the functionality of `RestKit` communications, providing convenience methods for determining the type of HTTP post, handling data transformations, and interpreting responses.
- `SFSessionRefresher`—Tightly-coupled with `SFRestAPI`, providing an abstraction around functionality for automatically refreshing a session if any REST requests fail due to session expiration.

# Android Architecture

The SalesforceSDK is provided as a library project. You need to reference the SalesforceSDK project from your application project. See the Android developer documentation.

## Java Code

Java sources are under `/src`.

### Java Code

| Package Name | Description |
|---|---|
| `com.salesforce.androidsdk.app` | SDK application classes (SalesforceSDKManager) |
| `com.salesforce.androidsdk.auth` | OAuth support classes |
| `com.salesforce.androidsdk.phonegap` | Native implementation of Salesforce Mobile SDK PhoneGap plugin |

| Package Name | Description |
|---|---|
| com.salesforce.androidsdk.rest | Classes for REST requests/responses |
| com.salesforce.androidsdk.security | Security-related helper classes (e.g. passcode manager) |
| com.salesforce.androidsdk.smartstore | SmartStore and supporting classes |
| com.salesforce.androidsdk.ui | Activities (e.g. login) |
| com.salesforce.androidsdk.ui.sfhybrid | App activity base classes |
| com.salesforce.androidsdk.ui.sfnative | App activity base classes |
| com.salesforce.androidsdk.util | Miscellaneous utility classes |

### com.salesforce.androidsdk.app

| Class | Description |
|---|---|
| SalesforceSDKManager | Abstract subclass of application; you must supply a concrete subclass in your project. |
| UpgradeManager | Helper class for upgrades |
| UUIDManager | Helper class for UUID generation |

### com.salesforce.androidsdk.auth

| Class | Description |
|---|---|
| AccountWatcher | Watcher responsible for cleanup when account is removed from settings application |
| AuthenticatorService | Service taking care of authentication |
| HttpAccess | Generic HTTP access layer |
| LoginServerManager | Manages login hosts |
| OAuth2 | Helper class for common OAuth2 requests |

### com.salesforce.androidsdk.phonegap

| Class | Description |
|---|---|
| ForcePlugin | Abstract super class for all Salesforce plugins |
| JavaScriptPluginVersion | Helper class to encapsulate the version reported by the JavaScript code |
| SalesforceOAuthPlugin | PhoneGap plugin for Salesforce OAuth |
| SDKInfoPlugin | PhoneGap plugin to get information about the SDK container |
| TestRunnerPlugin | PhoneGap plugin to run javascript tests in container |

### com.salesforce.androidsdk.rest

| Class | Description |
|---|---|
| ClientManager | Factory of RestClient, kicks off login flow if needed |
| RestClient | Authenticated client to talk to a Force.com server |
| RestRequest | Force.com REST request wrapper |
| RestResponse | REST response wrapper |

### com.salesforce.androidsdk.security

| Class | Description |
|---|---|
| Encryptor | Helper class for encryption/decryption/hash computations |
| PasscodeManager | Inactivity timeout manager, kicks off passcode screen if needed |

### com.salesforce.androidsdk.smartstore.app

This package is part of the SmartStore library project.

| Class | Description |
|---|---|
| SalesforceSDKManagerWithSmartStore | Super class for all force applications that use the SmartStore (lives in SmartStore library project) |
| UpgradeManagerWithSmartStore | Upgrade manager for applications that use the SmartStore (lives in SmartStore library project) |

### com.salesforce.androidsdk.smartstore.phonegap

This package is part of the SmartStore library project.

| Class | Description |
|---|---|
| SmartStorePlugin | PhoneGap plugin for SmartStore |
| StoreCursor | Represents a query cursor |

### com.salesforce.androidsdk.smartstore.store

This package is part of the SmartStore library project.

| Class | Description |
|---|---|
| DBHelper | Helper class to access the database underlying SmartStore |
| DBOpenHelper | Helper class to manage regular database creation and version management |
| IndexSpec | Represents an index specification |
| QuerySpec | Represents a query specification |
| SmartSqlHelper | Helper class for parsing and running SmartSql |
| SmartStore | Searchable/secure store for JSON documents |

### com.salesforce.androidsdk.ui

| Class | Description |
|---|---|
| CustomServerUrlEditor | Custom dialog allowing user to pick a different login host |
| LoginActivity | Login screen |
| SalesforceActivity | Main activity of native application should extend or duplicate the functionality of this class |
| OAuthWebviewHelper | Helper class to manage a WebView instance that is going through the OAuth login process |
| PasscodeActivity | Passcode (PIN) screen |
| SalesforceDroidGapActivity | Main activity for hybrid applications |
| SalesforceGapViewClient | WebView client used in hybrid applications |
| SalesforceR | Class that allows references to resources defined outside the SDK |
| ServerPickerActivity | Choose login host screen |

### com.salesforce.androidsdk.ui.sfhybrid

| Class | Description |
|---|---|
| SalesforceDroidGapActivity | Defines the main activity for a Cordova-based application |
| SalesforceGapViewClient | Defines the web view client for a Cordova-based application |

### com.salesforce.androidsdk.ui.sfnative

| Class | Description |
|---|---|
| SalesforceActivity | Main activity of native applications. All native activities are encouraged to extend one of the classes in this package, or else duplicate the functionality of one of these classes. |
| SalesforceListActivity | Main activity of native applications, based on the Android `ListActivity` class. All native activities are encouraged to extend one of the classes in this package, or else duplicate the functionality of one of these classes. |
| SalesforceExpandableListActivity | Main activity of native applications, based on the Android `ExpandableListActivity` class. All native activities are encouraged to extend one of the classes in this package, or else duplicate the functionality of one of these classes. |

### com.salesforce.androidsdk.util

| Class | Description |
|---|---|
| BaseActivityInstrumentationTestCase | Super class for activty test classes |
| EventsListenerQueue | Class to track activity events using a queue, allowing for tests to wait for certain events to turn up |

| Class | Description |
|---|---|
| `EventsObservable` | Used to register and receive events generated by the SDK (used primarily in tests) |
| `EventsObserver` | Observer of SDK events |
| `SalesforceSDKManagerInstrumentationTestCase` | Super class for tests of an application using the Salesforce Mobile SDK |
| `HybridInstrumentationTestCase` | Super class for tests of hybrid application |
| `JSTestCase` | Super class to run tests written in JavaScript |
| `JUnitReportTestRunner` | Test runner that runs tests using a time run cap |
| `LogUtil` | Helper methods for logging |
| `NativeInstrumentationTestCase` | Super class for tests of native application |
| `TimeLimitedTestRunner` | Test runner that limits the lifetime of the test run |

## Libraries

Libraries are under `/libs`.

| Library Name | Description |
|---|---|
| `cordova-2.3.0.jar` | Open source mobile development framework; used in hybrid applications (*) |
| `sqlcipher.jar` | Open source extension to SQLite that provides transparent 256-bit AES encryptiong of database files (**) |
| `x86/*.so` | Native libraries required by sqlcipher on Intel-based devices |
| `armeabi/*.so` | Native libaries required by sqlcipher on ARM-based devices (**) |
| `commons-code.jar, guava-r09.jar` | Java libraries required by sqlcipher |

(*) denotes files required for hybrid application.

(**) denotes files required for SmartStore.

## Resources

Resources are under `/res`.

### `drawable-hdpi`

| File | Use |
|---|---|
| `sf__edit_icon.png` | Server picker screen |
| `sf__highlight_glare.png` | Login screen |
| `sf__icon.png` | Application icon |

### drawable-ldpi

| File | Use |
|---|---|
| sf__icon.png | Application icon |

### drawable-mdpi

| File | Use |
|---|---|
| sf__edit_icon.png | Server picker screen |
| sf__highlight_glare.png | Login screen |
| sf__ic_refresh_sync_anim0.png | Application icon |
| sf__icon.png | Application icon |

### drawable

| File | Use |
|---|---|
| sf__header_bg.png | Login screen |
| sf__progress_spinner.xml | Login screen |
| sf__toolbar_background.xml | Login screen |

### drawable-xlarge

| File | Use |
|---|---|
| sf__header_bg.png | Login screen (tablet) |
| sf__header_drop_shadow.xml | Login screen (tablet) |
| sf__header_left_border.xml | Login screen (tablet) |
| sf__header_refresh.png | Login screen (tablet) |
| sf__header_refresh_press.png | Login screen (tablet) |
| sf__header_refresh_states.xml | Login screen (tablet) |
| sf__header_right_border.xml | Login screen (tablet) |
| sf__login_content_header.xml | Login screen (tablet) |
| sf__nav_shadow.png | Login screen (tablet) |
| sf__oauth_background.png | Login screen (tablet) |
| sf__oauth_container_dropshadow.9.png | Login screen (tablet) |
| sf__progress_spinner.xml | Login screen (tablet) |
| sf__refresh_loader.png | Login screen (tablet) |
| sf__toolbar_background.xml | Login screen (tablet) |

### drawable-xlarge-port

| File | Use |
|---|---|
| sf__oauth_background.png | Login screen (tablet) |

### layout

| File | Use |
|---|---|
| sf__custom_server_url.xml | Server picker screen |
| sf__login.xml | Login screen |
| sf__passcode.xml | Pin screen |
| sf__server_picker.xml | Server picker screen (deprecated) |
| sf__server_picker_list.xml | Server picker screen |

### layout-land

| File | Use |
|---|---|
| sf__passcode.xml | PIN screen |

### layout-xlarge

| File | Use |
|---|---|
| sf__header_bottom.xml | Header (tablet) |
| sf__header_separator.xml | Header (tablet) |
| sf__login.xml | Login screen (tablet) |
| sf__login_header.xml | Login screen (tablet) |
| sf__passcode.xml | PIN screen (tablet) |
| sf__server_picker.xml | Server picker screen (tablet) |
| sf__server_picker_header.xml | Server picker screen (tablet) |

### menu

| File | Use |
|---|---|
| sf__clear_custom_url.xml | Add connection dialog |
| sf__login.xml | Login menu (phone) |

### values

| File | Use |
|---|---|
| bootconfig.xml | Connected app configuration settings |
| sf__colors.xml | Colors |

| File | Use |
|---|---|
| sf__dimens.xml | Dimensions |
| sf__strings.xml | SDK strings |
| sf__style.xml | Styles |
| strings.xml | Other strings (app name) |

### values-xlarge

| File | Use |
|---|---|
| sf__colors.xml | Colors (tablet) |
| sf__dimens.xml | Dimensions (tablet) |
| styles.xml | Styles (tablet) |

### xml

| File | Use |
|---|---|
| authenticator.xml | Preferences for account used by application |
| config.xml | Plugin configuration file for PhoneGap. Required for hybrid. |
| servers.xml | Server configuration. |

# Index

## J



## K



## L



## M



## N



## O



## P



## Q



## R

## S



## T



## U



## V



## W



## X