Mobile SDK Development Guide

Salesforce Mobile SDK 3.2

@salesforcedocs

Last updated: April 27, 2015

CONTENTS

PREFACE

Mobile devices have radically changed the way we work and play. People consume, create, and share data on a wide range of connected devices. Workers use smart phones and tablets to stay in touch, connect with customers and peers, and engage on social networks and apps.

However, many companies continue to run their businesses on enterprise applications that don't work in the mobile world. These legacy applications remain locked away on corporate intranets and aren't available in employees' hands when they're needed. They don't provide a modern user experience, and they aren't wired into social graphs like consumer apps.

Yesterday's platforms were not designed to meet the demands of the mobile world. Big, monolithic stacks and rigid integration patterns lack the scalability and flexibility required by mobile technology. Techniques that evolved since the 1990s for web applications on PCs don't apply in mobile apps. Mobile applications require new architectures and software designs, and they need to run on platforms built for mobile application development and wireless connectivity. Today, outdated corporate applications are rapidly being replaced by mobile-ready cloud apps.

Table 1: Comparison of PC/Web applications and a modern mobile application

Category	Typical PC / Web application	Mobile / modern application
Connection and Availability	• Fast, reliable LAN • Low latency • High bandwidth • Connectivity assumed	• Varying connection • High latency • Low bandwidth • Offline operation required
User Interactions	• Keyboard and mouse • Long desktop interactions	• Touch screen • Quick, focused actions
Perimeter Security	• Corporate VPN or LAN access to applications	• Cumbersome to require VPN from mobile devices • IP restrictions ineffective with public mobile networks
Device Standardization	• Typically purchased and controlled by IT	• Often Bring Your Own Device (BYOD) • Multiple platforms
Form Factor	• Large (PC) screen	• Apps must support phone, tablet, and desktop
Social	• Typically siloed applications • Email-based collaboration	• Native user collaboration • Intuitively share and collaborate
Multi-device	• Client-server architectures with data stored on server (Web)	• Instant sharing between devices • Data propagation between devices
Device Interaction	• Applications rarely leverage telephony, camera, and other media devices	• Native use of mobile device's camera, contacts, calendar, and location

Category	Typical PC / Web application	Mobile / modern application
Location	• Rarely used in Web applications	• Commonly used both to associate data with a location and to filter data and services based on location

Salesforce provides a state–of–the–art cloud–based platform for building CRM mobile apps. The Salesforce Mobile SDK gives you advanced control over mobile device features, offline support, data synchronization, and mobile software design.

Salesforce Platform Mobile Services

Enterprise IT departments now face the daunting task of connecting their enterprise data and services with a mobile workforce. Salesforce faced this problem itself as it moved its enterprise CRM and service applications to the mobile world. This transformation required fundamental changes in the underlying technology and implementation to support Salesforce's applications across multiple platforms (iOS, Android) and multiple form factors (phone, tablet, and PC) with enterprise-grade reliability, availability, and security. The lessons learned and technology built to transform Salesforce's applications for mobile are now available for any company that uses the Salesforce cloud.

Salesforce Platform Mobile Services are designed to meet the challenges of mobile applications.

Salesforce Platform Mobile Services is the next-generation platform that powers Salesforce mobile applications, enabling enterprises to build their own Android, iPhone, and iPad applications. These services leverage the power of the Salesforce platform and its proven security, reliability, and scale for enterprise applications.

Salesforce Platform Mobile Services comprises three core components.

- Mobile Services in Force.com
- Salesforce Mobile SDK
- Identity

Mobile Services in Force.com

Mobile services in Force.com focus on developing and administering enterprise mobile applications.

- Mobile REST APIs provide access to enterprise data and services, leveraging standard Web protocols. Developers can quickly access their business data through REST APIs and leverage that data across phone, tablet, and web user interfaces. The REST APIs provide a single place to enforce access, security, common policy across all device types.
- Social (Chatter) REST APIs enable developers to quickly transform their applications with social networks and collaboration features. The Chatter REST API provides access to the feed, as well as the social graph of user connections. Mobile applications can easily consume or post items to a user or group, or leverage the social graph to enable instant collaboration between connected users.
- Mobile policy management enables administrators to enforce their enterprise security policy on mobile applications in a world without perimeter security. Administrators can enable security features such as two-factor authentication, device PIN protection, and password rotation. They can also enable and disable user access to mobile applications.
- Geolocation provides location-based information to enhance your online business processes with geospatial data. All objects in Salesforce include a compound geolocation field. The entire platform is location-ready, allowing radius-based searching and other spatial queries.

Salesforce Mobile SDK

Salesforce Mobile SDK lets you develop native Objective-C apps for iOS and Java apps for Android. You can also use it to provide a native container for hybrid apps written in HTML5 and JavaScript. Npm scripts for iOS and Android help you get started building native and hybrid apps. Salesforce Mobile SDK provides:

- Native device services. You can access device features such as the camera, GPS, and contacts across a broad range of iOS and Android devices.
- Secure offline storage and data synchronization. You can build applications which continue to function with limited or no network connectivity. The data stored on the device is securely encrypted and safe, even if the device is lost or stolen.
- Client OAuth authentication support. You're free from having to rebuild login pages and general authentication in mobile apps. Mobile SDK apps quickly and easily integrate with enterprise security management.

Identity

Identity provides a single enterprise identity and sign-on service to connect mobile devices with enterprise data and services. Identity provides the following advantages.

- Single sign-on across applications and devices, so users aren't forced to create multiple usernames and passwords.
- A trusted identity provider that you can leverage for any enterprise platform or application.
- A Cloud Directory that enables enterprises to white label identity services and use company-specific appearance and branding.
- The ability to utilize consumer identity providers, such as Facebook. This feature allows customer-facing applications to quickly engage with customer social data.

Customize Salesforce1, or Create a Custom App?

When it comes to developing functionality for your Salesforce mobile users, you have options. Although this book deals only with Mobile SDK development, Salesforce also provides the Salesforce1 Platform for mobile app development.

Here are some differences between extending Salesforce1 and creating custom apps using the Mobile SDK. For more information on Salesforce1, see developer.salesforce.com/docs.

Customizing Salesforce1

- Has a pre-defined user interface.
- Has full access to Salesforce data.
- You can create an integrated experience with functionality developed in the Salesforce1 Platform.
- The Action Bar gives you a way to include your own apps/functionality.
- You can customize Salesforce1 with either point-and-click or programmatic customizations.
- Functionality can be added programmatically through Visualforce pages or Force.com Canvas apps.
- Salesforce1 customizations or apps adhere to the Salesforce1 navigation. So, for example, a Visualforce page can be called from the navigation menu or from the Action Bar.
- You can leverage existing Salesforce development experience, both point-and-click and programmatic.
- Included in all Salesforce editions and supported by Salesforce.

Building Custom Mobile Apps

Custom apps can be either free-standing apps you create with Salesforce Mobile SDK or browser apps using plain HTML5 and JQuery Mobile/Ajax. With custom apps, you can:

- Define a custom user experience.
- Access Salesforce data using REST APIs in native and hybrid local apps, or with Visualforce in hybrid apps using JavaScript Remoting. In HTML5 apps, do the same using JQueryMobile and Ajax.
- Brand your user interface for customer-facing exposure.
- Create standalone mobile apps, either with native APIs using Java for Android or Objective-C for iOS, or through a hybrid container using JavaScript and HTML5 (Mobile SDK only).
- Distribute apps through mobile industry channels, such as the Apple App Store or Google Play (Mobile SDK only).
- Configure and control complex offline behavior (Mobile SDK only).
- Use push notifications
- Design a custom security container using your own OAuth module (Mobile SDK only).
- Other important Mobile SDK considerations:
  - Open-source SDK, downloadable for free through npm installers as well as from GitHub. No licensing required.
  - Requires you to develop and compile your apps in an external development environment (Xcode for iOS, Eclipse or similar for Android).
  - Development costs range from $0 to $1M or more, plus maintenance costs.

Mobile SDK integrates Force.com cloud architecture into Android and iOS apps by providing:

- SmartSync Data Framework for accessing and syncing Salesforce data through JavaScript
- Implementation of Salesforce Connected App policy
- OAuth credentials management, including persistence and refresh capabilities
- Wrappers for Salesforce REST APIs
- Cordova-based containers for hybrid apps
- Data syncing for hybrid apps
- Secure offline storage with SmartStore
- Push notification support for native and **hybrid** apps
- Support for Salesforce Communities
- Support for multiple user logins

About This Book

This book introduces you to Salesforce Mobile SDK and teaches you how to design, develop, and manage mobile applications for the cloud. The chapters cover a wide range of development techniques for various skill sets, beginning with HTML5 and JavaScript, continuing through hybrid apps, and culminating in native iOS and Android development.

Each development paradigm is represented by a quick start tutorial. Most of these tutorials take you through the steps of creating a simple master-detail application that accesses Salesforce through REST APIs. Tutorials include:

- Running the ContactExplorer Hybrid Sample
- Tutorial: Creating a Native Android Warehouse Application
- Tutorial: Creating a Native iOS Warehouse App

- Tutorial: Creating a SmartSync Application

You'll also find pointers to Mobile SDK sample apps, tips and techniques for working with communities and managing hybrid apps, and descriptions of Mobile SDK features such as:

- Using SmartStore to Securely Store Offline Data
- Using SmartSync to Access Salesforce Objects
- Files and Networking
- Push Notifications and Mobile SDK

Enjoy your exploration of Salesforce Mobile SDK!

 Note: An online version of this book is available at developer.salesforce.com/docs.

Sending Feedback

Questions or comments about anything you see in this book? Suggestions for topics that you'd like to see covered in future versions? You can:

- Join the SalesforceMobileSDK community at `plus.google.com/communities`
- Post your thoughts on the Salesforce developer discussion forums at `https://developer.salesforce.com/forums`
- Email us directly at developerforce@salesforce.com

.

CHAPTER 1 Introduction to Salesforce Mobile Development

In this chapter ...

Salesforce Mobile SDK lets you harness the power of Force.com within stand-alone mobile apps.

Force.com provides a straightforward and productive platform for Salesforce cloud computing. Developers can use Force.com to define Salesforce application components—custom objects and fields, workflow rules, Visualforce pages, Apex classes, and triggers. They can then assemble those components into awesome, browser-based desktop apps.

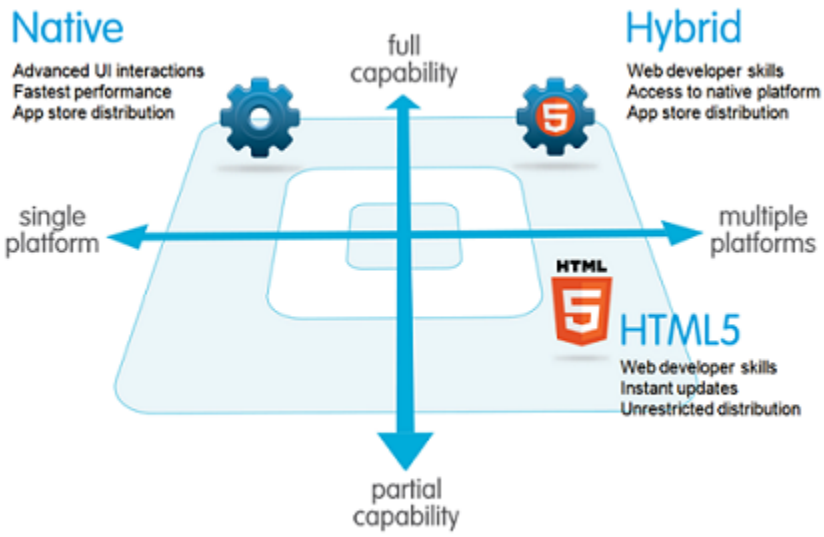
Unlike a desktop app, a Mobile SDK app accesses Salesforce data through a mobile device's native operating system rather than through a browser. To ensure a satisfying and productive mobile user experience, you can configure Mobile SDK apps to move seamlessly between online and offline states. Before you dive into this exciting world, look at how mobile development works, and also learn about essential Salesforce developer resources.

About Native, HTML5, and Hybrid Development

Salesforce Mobile SDK gives you options for how you'll develop your app. The option you choose depends on your development skills, device and technology requirements, goals, and schedule.

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 essentially 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).

Getting started with HTML5 is easier than with 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 and different operating systems.

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, what are the drawbacks?

- **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, query, search, and edit Salesforce data
- Securely cache Salesforce data for offline use
- Synchronize locally cached data 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.

Locally
: 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.

Remotely
: 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
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	Not secure; shared SQL, Key-Value stores	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.

- Hybrid Apps Quick Start on page 122
- iOS Native Quick Start on page 22
- Android Native Quick Start on page 69

CHAPTER 2 Getting Started With Mobile SDK

In this chapter ...

Let's get started creating custom mobile apps! If you haven't done so already, begin by signing up for Force.com and installing Mobile SDK development tools.

In addition to signing up, you need a Connected App definition, regardless of which development options you choose. For hybrid and native apps, install the Mobile SDK npm package for each platform you plan to support.

Developer Edition or Sandbox Environment?

Salesforce offers a range of environments for developers. The environment that's best for you depends on many factors, including:

- The type of application you're building
- Your audience
- Your company's resources

Development environments are used strictly for developing and testing apps. These environments contain test data that are not business critical. Development can be done inside your browser or with the Force.com IDE, which is based on the Eclipse development tool. There are two types of development environments: Developer Edition and Sandbox.

Types of Developer Environments

A *Developer Edition* (DE) environment is a free, fully-featured copy of the Enterprise Edition environment, with less storage and users. DE is a logically separate environment, ideal as your initial development environment. You can sign-up for as many DE organizations as you need. This allows you to build an application designed for any of the Salesforce production environments.

A *Partner Developer Edition* is a licensed version of the free DE that includes more storage, features, and licenses. Partner Developer Editions are free to enrolled Salesforce partners.

Sandbox is a nearly identical copy of your production environment available to Enterprise or Unlimited Edition customers. The sandbox copy can include data, configurations, or both. It is possible to create multiple sandboxes in your production environments for a variety of purposes without compromising the data and applications in your production environment.

Choosing an Environment

In this book, all exercises assume you're using a Developer Edition (DE) organization. However, in reality a sandbox environment can also host your development efforts. Here's some information that can help you decide which environment is best for you.

Developer Edition is ideal if:

- You are a partner who intends to build a commercially available Force.com app by creating a managed package for distribution through AppExchange and/or Trialforce.

  Note: Only Developer Edition or Partner Developer Edition environments can create managed packages.

- You are a salesforce.com customer with a Professional, Group, or Personal Edition, and you do not have access to Sandbox.
- You are a developer looking to explore the Force.com platform for FREE!

Partner Developer Edition is ideal if:

- You are developing in a team and you require a master environment to manage all the source code - in this case each developer would have a Developer Edition environment and check their code in and out of this master repository environment.
- You expect more than 2 developers to log in to develop and test.
- You require a larger environment that allows more users to run robust tests against larger data sets.

Sandbox is ideal if:

- You are a salesforce.com customer with Enterprise, Unlimited, or Force.com Edition, which includes Sandbox.
- You are developing a Force.com application specifically for your production environment.
- You are not planning to build a Force.com application that will be distributed commercially.
- You have no intent to list on the AppExchange or distribute through Trialforce.

Development Prerequisites

We recommend some background knowledge and system setup before you begin building Mobile SDK apps.

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

Familiarity with OAuth, login and passcode flows, and Salesforce connected apps is essential to designing and debugging Mobile SDK apps. See Authentication, Security, and Identity in Mobile Apps.

The following requirements apply to specific platforms and technologies:

- To build iOS applications (hybrid or native), see Native iOS Requirements.
- To build Android applications (hybrid or native), see Native Android Requirements.
- To build remote hybrid applications, you'll need an organization that has Visualforce.

Sign Up for Force.com

To access a wealth of tutorials, blogs, and support forums for all Salesforce developer programs, join Force.com.

1. In your browser go to `https://developer.salesforce.com/signup`.
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.salesforce.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.

Creating a Connected App

To enable your mobile app to connect to the Salesforce service, you need to create a connected app. The connected app includes a consumer key, a prerequisite to all development scenarios in this guide.

Create a Connected App

To create a connected app, you use the Salesforce app.

1. Log into your Force.com instance.
2. In Setup, navigate to **Create** > **Apps**.
3. Under Connected Apps, click **New**.
4. Perform steps for Basic Information.
5. Perform steps for API (Enable OAuth Settings).
6. Click **Save**.

If you plan to support push notifications, see Push Notifications and Mobile SDK on page 250 for additional connected app settings. You can add these settings later if you don't currently have the necessary information.

Note:

- The `Callback URL` provided for OAuth doesn't 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://`*.
- The detail page for your connected app displays a consumer key. It's a good idea to copy this key, as you'll need it later.
- After you create a new connected app, wait a few minutes for the token to propagate before running your app.

Basic Information

Specify basic information about your app in this section, including the app name, logo, and contact information.

1. Enter the `Connected App Name`. This name is displayed in the list of connected apps.

   Note: The name must be unique for the current connected apps in your organization. You can reuse the name of a deleted connected app if the connected app was created using the Spring '14 release or later. You cannot reuse the name of a deleted connected app if the connected app was created using an earlier release.

2. Enter the `API Name`, used when referring to your app from a program. It defaults to a version of the name without spaces. Only letters, numbers, and underscores are allowed, so you'll need to edit the default name if the original app name contained any other characters.
3. Provide the `Contact Email` that Salesforce should use for contacting you or your support team. This address is not provided to administrators installing the app.
4. Provide the `Contact Phone` for Salesforce to use in case we need to contact you. This number is not provided to administrators installing the app.
5. Enter a `Logo Image URL` to display your logo in the list of connected apps and on the consent page that users see when authenticating. The URL must use HTTPS. The logo image can't be larger than 125 pixels high or 200 pixels wide, and must be in the GIF, JPG, or PNG file format with a 100 KB maximum file size. The default logo is a cloud. You have several ways to add a custom logo.
   - You can upload your own logo image by clicking **Upload logo image**. Select an image from your local file system that meets the size requirements for the logo. When your upload is successful, the URL to the logo appears in the `Logo Image URL` field. Otherwise, make sure the logo meets the size requirements.
   - You can also select a logo from the samples provided by clicking **Choose one of our sample logos**. The logos available include ones for Salesforce apps, third-party apps, and standards bodies. Click the logo you want, and then copy and paste the displayed URL into the `Logo Image URL` field.
   - You can use a logo hosted publicly on Salesforce servers by uploading an image that meets the logo file requirements (125 pixels high or 200 pixels wide, maximum, and in the GIF, JPG, or PNG file format with a 100 KB maximum file size) as a document using the Documents tab. Then, view the image to get the URL, and enter the URL into the `Logo Image URL` field.
6. Enter an `Icon URL` to display a logo on the OAuth approval page that users see when they first use your app. The logo should be 16 pixels high and wide, on a white background. Sample logos are also available for icons.

   You can select an icon from the samples provided by clicking **Choose one of our sample logos**. Click the icon you want, and then copy and paste the displayed URL into the `Icon URL` field.
7. If there is a a Web page with more information about your app, provide a `Info URL`.
8. Enter a `Description` to be displayed in the list of connected apps.

Prior to Winter '14, the `Start URL` and `Mobile Start URL` were defined in this section. These fields can now be found under Web App Settings and Mobile App Settings below.

API (Enable OAuth Settings)

This section controls how your app communicates with Salesforce. Select `Enable OAuth Settings` to configure authentication settings.

1. Enter the `Callback URL` (endpoint) that Salesforce calls back to your application during OAuth; it's the OAuth `redirect_uri`. Depending on which OAuth flow you use, this is typically the URL that a user's browser is redirected to after successful authentication. As this URL is used for some OAuth flows to pass an access token, the URL must use secure HTTP (HTTPS) or a custom URI scheme. If you enter multiple callback URLs, at run time Salesforce matches the callback URL value specified by the application with one of the values in `Callback URL`. It must match one of the values to pass validation.
2. If you're using the JWT OAuth flow, select `Use Digital Signatures`. If the app uses a certificate, click **Choose File** and select the certificate file.
3. Add all supported OAuth scopes to `Selected OAuth Scopes`. These scopes refer to permissions given by the user running the connected app, and are followed by their OAuth token name in parentheses:

   Access and manage your Chatter feed (chatter_api)
   Allows access to Chatter REST API resources only.

   Access and manage your data (api)
   Allows access to the 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.

   Access your basic information (id, profile, email, address, phone)
   Allows access to the Identity URL service.

   Access custom permissions (custom_permissions)
   Allows access to the custom permissions in an organization associated with the connected app, and shows whether the current user has each permission enabled.

   Allow access to your unique identifier (openid)
   Allows access to the logged in user's unique identifier for OpenID Connect apps.

   Full access (full)
   Allows access to all data accessible by the logged-in user, and encompasses all other scopes. `full` does not return a refresh token. You must explicitly request the `refresh_token` scope to get a refresh token.

   Perform requests on your behalf at any time (refresh_token, offline_access)
   Allows a refresh token to be returned if you are eligible to receive one. This lets the app interact with the user's data while the user is offline. The `refresh_token` scope is synonymous with `offline_access`.

   Provide access to custom applications (visualforce)
   Allows access to Visualforce pages.

   Provide access to your data via the Web (web)
   Allows the ability to use the `access_token` on the Web. This also includes `visualforce`, allowing access to Visualforce pages.

If your organization had the `No user approval required for users in this organization` option selected on your remote access prior to the Spring '12 release, users in the same organization as the one the app was created in still have automatic approval for the app. The read-only `No user approval required for users in this organization` checkbox is selected to show this condition. For connected apps, the recommended procedure after you've created an app is for administrators

to install the app and then set `Permitted Users` to `Admin-approved users`. If the remote access option was not checked originally, the checkbox doesn't display.

SEE ALSO:

Scope Parameter Values

Installing Mobile SDK

Salesforce Mobile SDK provides two installation paths.

- (*Recommended*) You can install the SDK in a ready-made development setup using a Node Packaged Module (npm) script.
- You can download the Mobile SDK open source code from GitHub and set up your own development environment.

Mobile SDK npm Packages

Most mobile developers want to use Mobile SDK as a "black box" and begin creating apps as quickly as possible. For this use case Salesforce provides two npm packages: **forceios** for iOS, and **forcedroid** for Android.

Mobile SDK npm packages provide a static snapshot of an SDK release. For iOS, the npm package installs binary modules rather than uncompiled source code. For Android, the npm package installs a snapshot of the SDK source code rather than binaries. You use the npm scripts not only to install Mobile SDK, but also 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.

Do This First: Install Node.js and npm

To use the Mobile SDK npm installers, you first install Node.js. The Node.js installer automatically installs npm.

1. Download Node.js from www.nodejs.org/download.
2. Run the downloaded installer to install Node.js and npm. Accept all prompts that ask for permission to install.
3. Test your installation at a command prompt by typing *`npm`*, then pressing *`ENTER`* or *`RETURN`*. If you don't see a page of command usage information, revisit Step 2 to find out what's missing.

Now you're ready to download the npm scripts and install Salesforce Mobile SDK for Android and iOS.

iOS Installation

For the fastest, easiest route to iOS development, use the forceios npm package to install Salesforce Mobile SDK.

1. At a command prompt, use the forceios package to install the Mobile SDK either globally (recommended) or locally.
   a. **For global installation:** 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. **For local installation:** Change directories to your preferred installation 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.

Android Installation

For the fastest, easiest route to Android development, use the forcedroid npm package to install Salesforce Mobile SDK.

1. Use the forcedroid package to install the Mobile SDK either globally (recommended) or locally.
   a. **For global installation:** Append `-g` to the end of the command.

   The installer's `-g` option installs `forcedroid` in a global location so that you can run it from any directory.

   - In non-Windows environments, most users need the `sudo` option because you lack read-write permissions in `/usr/local`. Type the following command at a terminal window:

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

     The npm utility installs global packages under `/usr/local/lib/node_modules`, and links binary modules in `/usr/local/bin`.
   - In Windows, type the following command at the Windows command prompt:

     ```
     npm install forcedroid -g
     ```

     The npm utility installs global packages in `%APPDATA%\npm\node_modules`, and links binaries in `%APPDATA%\npm`.

   b. **For local installation:** Change directories to your preferred installation folder 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 Mobile SDK npm Packages

If you need to uninstall an npm package, use the npm script.

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"
```

Uninstalling the Forceios Package

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"
```

Mobile SDK GitHub Repository

More adventurous developers can delve into the SDK, keep up with the latest changes, and possibly contribute to SDK development by cloning 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`

iOS: Cloning the Mobile SDK GitHub Repository (Optional)

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`

Android: Cloning the Mobile SDK GitHub Repository (Optional)

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`.

Creating Android Projects With the Cloned GitHub Repository

To create Android native and hybrid projects with the cloned SalesforceMobileSDK-Android repository, follow the instructions in `native/README.md` and `hybrid/README.md` files.

Creating iOS Projects With the Cloned GitHub Repository

To create native and hybrid projects with the cloned SalesforceMobileSDK-iOS repository, follow the instructions in `build.md` in the repository's root directory.

Mobile SDK Sample Apps

Salesforce Mobile SDK includes a wealth of sample applications that demonstrate its major features. Use the hybrid and native samples for iOS and Android as the basis for your own applications, or just study them for reference.

Installing the Sample Apps

In GitHub, sample apps live in the Mobile SDK repository for the target platform. You can access them there directly, or, for Android, you can import the shared source code from the SalesforceMobileSDK-Shared repository.

Accessing Sample Apps From the GitHub Platform Repositories

If you clone Mobile SDK directly from GitHub, all sample files are placed in `hybrid/SampleApps` and `native/SampleApps` directories. You can then build the Android samples by importing the Cordova project, `SalesforceSDK` project, `SmartStore` project, and the sample projects into your Eclipse workspace.

For Android: After cloning the repository, remember to run *`cscript install.vbs`* on Windows or *`./install.sh`* on Mac in the repository root folder. These scripts place the samples in the `native/SampleApps` and `hybrid/SampleApps` folders.

For iOS: After cloning the repository, remember to run `./install.sh` in the repository root folder. Open the `SalesforceMobileSDK-iOS/SalesforceMobileSDK.xcworkspace` file in Xcode to find the sample apps in the `Native SDK` and `Hybrid SDK` project folders.

Building Hybrid Sample Apps With Cordova

To build hybrid sample apps, you can use the Cordova command line and the source files from the SalesforceMobileSDK-Shared repository. For instructions, see Build Hybrid Sample Apps on page 128.

Android Sample Apps

Native

- **RestExplorer** demonstrates the OAuth and REST API functions of the SalesforceSDK. It's also useful for investigating REST API actions from a tablet.
- **NativeSqlAggregator** demonstrates SQL aggregation with SmartSQL. As such, it also demonstrates a native implementation of SmartStore.
- **FileExplorer** demonstrates the Files API as well as the underlying Google Volley networking enhancements.
- **SmartSyncExplorer** demonstrates the power of the native SmartSync library on Android. It resides in the Mobile SDK for Android under `native/SampleApps/SmartSyncExplorer`.

Hybrid

- **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.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.
- **HybridFileExplorer**: Demonstrates the Files API.
- **SimpleSync:**: Demonstrates how to use the SmartSync plugin.
- **SmartStoreExplorer**: Lets you explore SmartStore APIs.
- **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.

iOS Sample Apps

Native

- **RestAPIExplorer** exercises all native REST API wrappers. It resides in the Mobile SDK for iOS under `native/SampleApps/RestAPIExplorer`.
- **NativeSqlAggregator** shows SQL aggregation examples plus a native SmartStore implementation. It resides in the Mobile SDK for iOS under `native/SampleApps/NativeSqlAggregator`.

- **FileExplorer** demonstrates the Files API and the underlying network library. This sample resides in the Mobile SDK for iOS under `native/SampleApps/FileExplorer`.
- **SmartSyncExplorer** demonstrates the power of the native SmartSync library on iOS. It resides in the Mobile SDK for iOS under `native/SampleApps/SmartSyncExplorer`.

Hybrid

- **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.
- **HybridFileExplorer**: Demonstrates the Files API.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.
- **SimpleSync:**: Demonstrates how to use the SmartSync plugin.
- **SmartStoreExplorer**: Lets you explore SmartStore APIs.
- **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.

What's New

You can always find the list of new features for the current release of Mobile SDK at https://developer.salesforce.com/page/Mobile_SDK_Release_Notes.

CHAPTER 3 Native iOS Development

In this chapter ...

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

Two important features 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 that 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

iOS development with Mobile SDK 3.2 requires the following software.

- Xcode—Version 6.0 is the minimum, but we recommend the latest version.
- iOS 7.0 or higher.

On the Salesforce side, you'll need:

- Salesforce Mobile SDK 3.2 for iOS. See Install the Mobile SDK.
- A Salesforce Developer Edition organization with a connected app.

Creating an iOS Project

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.

Specifying Application Options Directly

If you prefer, you can specify the forceios parameters directly at the command line. 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 name)
    --startpage=<App Start Page> (The start page of your remote app.
      Only required for hybrid_remote)
```

```
    [--outputdir=<Output directory> (Defaults to 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"
```

Open 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.

.

Run 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.

Use CocoaPods with Mobile SDK

You can use the CocoaPods to add Mobile SDK to an existing iOS app.

CocoaPods provide a convenient mechanism for merging external code packages or modules into existing Xcode projects. If you've never heard of or used CocoaPods, start by reading the documentation at www.cocoapods.org.

Mobile SDK for iOS provides a CocoaPods pod specification, or *podspec*, in the `forcedotcom/SalesforceMobileSDK-iOS` GitHub repo. The podspec, `SalesforceMobileSDK-iOS.podspec`, defines a master spec named `SalesforceMobileSDK-iOS` that installs all modules of Salesforce Mobile SDK for iOS. It also defines subspecs for individual SDK modules.

When you use your Podfile to merge one of these subspecs rather than the master spec, CocoaPods also merges all dependent modules.

To use CocoaPods with Mobile SDK, follow these steps.

1. Be sure you've installed the `cocoapods` Ruby gem as described at www.cocoapods.org.

2. In your project's Podfile, reference the Mobile SDK podspec that you intend to merge into your app. For example, to add all modules to your app, add the following declaration to your Podfile:

```
pod 'SalesforceMobileSDK-iOS'
```

3. In a Terminal window, run `pod install` from your project directory. CocoaPods downloads the dependencies for your requested pods, merges them into your project, and creates a workspace containing the newly merged project.

   Important: After running CocoaPods, always access your project only from the new workspace that `pod install` creates. For example, instead of opening `MyProject.xcodeproj`, open `MyProject.xcworkspace`.

4. To use Mobile SDK APIs in your merged app, remember these important tips.

   a. Import header files using angle brackets ("<" and ">") rather than double quotes. For example:

   ```
   #import <SFRestAPI.h>
   ```

   b. Implement the `SalesforceSDKManager` workflow in your `AppDelegate` class. The easiest way to do this task is to copy the bootstrap logic from the `AppDelegate` class in any Mobile SDK native sample app.

   c. Add authentication at a logical point in your app before you call any Mobile SDK API that accesses the network.

   Consult the Mobile SDK native sample apps for fully realized examples.

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 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 and OAuth2 Authentication Flow.

About Memory Management

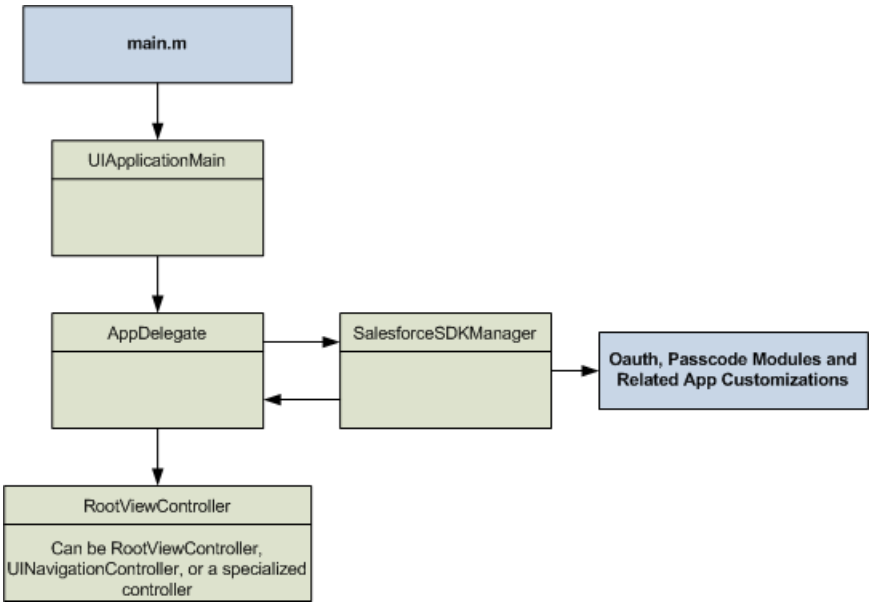
Beginning in Mobile SDK 2.0, native iOS apps use Automatic Reference Counting (ARC) to manage object memory. You don't have to allocate and then remember to deallocate your objects. See the Mac Developer Library at https://developer.apple.com for ARC syntax, guidelines, and best practices.

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.

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` uses a Mobile SDK service object, `SalesforceSDKManager`, to organize the application launch flow. This service coordinates Salesforce authentication and passcode activities. After the user is authenticated, `AppDelegate` passes control to the `RootViewController` object.



Note: The workflow demonstrated by the template app is merely an example. You can tailor your `AppDelegate` and supporting classes to achieve your desired workflow. For example, you can postpone Salesforce authentication until a later point. 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 explicitly terminates it, or until the device is rebooted.

SEE ALSO:

SalesforceSDKManager Class

SalesforceSDKManager Class

The `SalesforceSDKManager` class combines app identity and bootstrap configuration in a single component. It manages complex interactions between authentication and passcodes using configuration provided by the app developer. In effect, `SalesforceSDKManager` shields developers from having to control the bootstrap process.

The Mobile SDK template application uses the `SalesforceSDKManager` class to implement most of the Salesforce-specific startup functionality for you. `SalesforceSDKManager` manages and coordinates all objects involved in app launching, including PIN code, OAuth configuration, and other bootstrap processes. Using `SalesforceSDKManager` ensures that the complicated interactions between these processes occur in the proper sequence, while still providing you freedom to customize individual parts of the launch flow. Beginning with Mobile SDK 3.0, all iOS native apps must use `SalesforceSDKManager` to manage application launch behavior.

Note: The `SalesforceSDKManager` class, which is new in Mobile SDK 3.0, does not replace existing authentication management objects or events. Rather, it's a super-manager of the existing boot management objects. Existing code should continue to work as it did in earlier releases.

Life Cycle

`SalesforceSDKManager` is a singleton object that you access by sending the `sharedManager` class message:

```
[SalesforceSDKManager sharedManager]
```

This shared object is created exactly once, the first time your app calls `[SalesforceSDKManager sharedManager]`. It serves as a delegate for three other Mobile SDK manager objects:

- `SFUserAccountManager`
- `SFAuthenticationManager`
- `SFPasscodeManager`

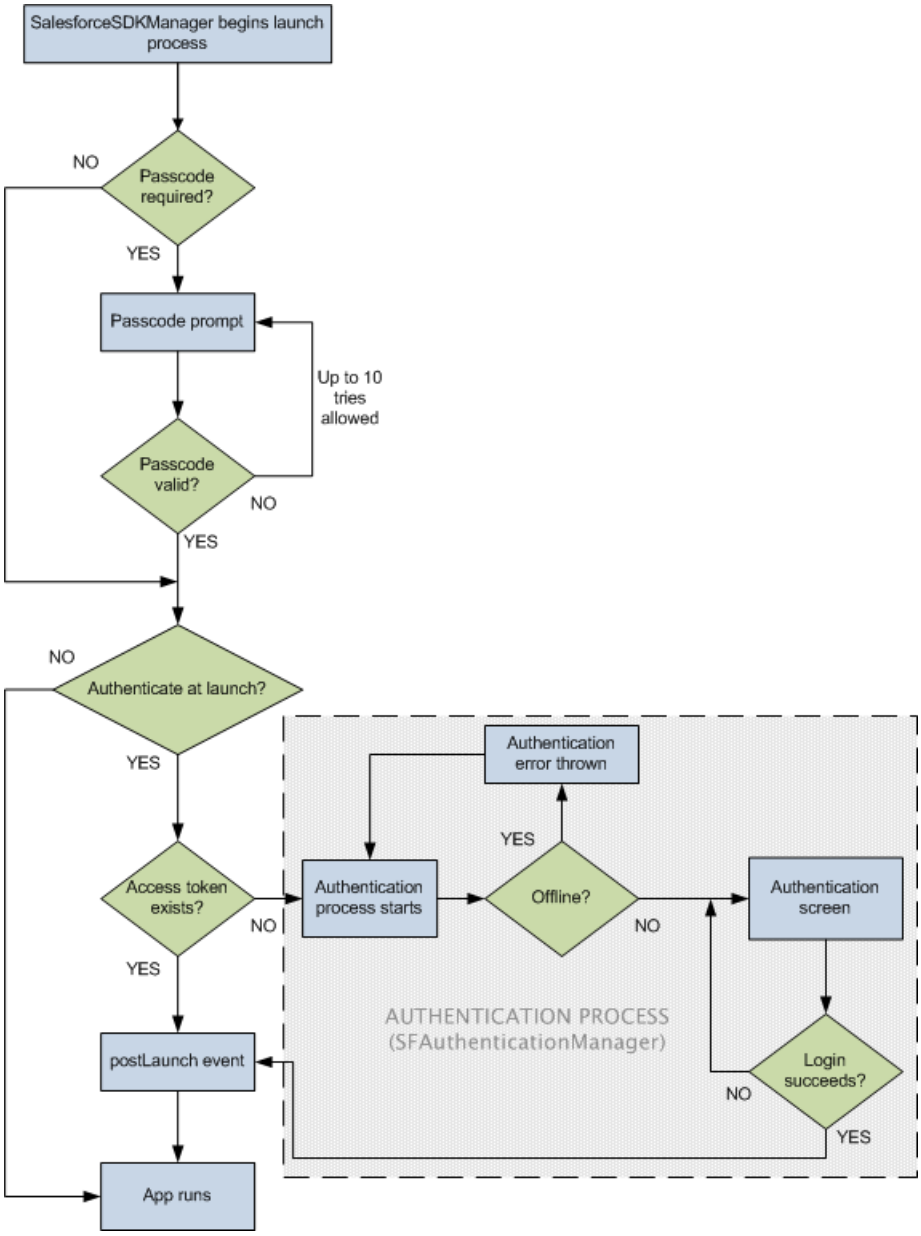
Your app uses the `SalesforceSDKManager` object in two scenarios:

1. At application startup, in the `init` and `application:didFinishLaunchingWithOptions:` methods of `AppDelegate`
2. Anytime the current user's OAuth tokens become invalid—either through logout, token expiration, or token revocation—while the app continues to run

The events in the first scenario happen only once during the app life cycle. The second scenario, though, can happen anytime. When the Mobile SDK detects that the current user's tokens are invalid, it re-runs the `SalesforceSDKManager` application launch flow, including any related event handlers that your app provides. Be sure to code these event handlers defensively so that you don't suffer unwanted losses of data or state if the app is reinitialized.

Application Launch Flow

When the `application:didFinishLaunchingWithOptions:` message arrives, you launch your app by sending the `launch` message to the shared `SalesforceSDKManager` instance. If a passcode is required by the app's connected app, `SalesforceSDKManager` displays the passcode verification screen to the user before continuing with the bootstrap process. The following diagram shows this flow.



Key points demonstrated in this diagram are:

- If the OAuth settings in the connected app definition do not require a passcode, the flow proceeds directly to Salesforce authentication.
- If a valid access token is found, the flow bypasses Salesforce authentication.
- If no access token is found and the device is offline, the authentication module throws an error and returns the user to the login screen. `SalesforceSDKManager` does not reflect this event to the app.
- The `postLaunch` event occurs only after all credentials and passcode challenges are verified.

Besides what's shown in the diagram, the `SalesforceSDKManager` launch process also delegates user switching and push notification setup to the app if the app supports those features. If the user fails or cancels either the passcode challenge or Salesforce login, a `postLogout` event fires, after which control returns to `AppDelegate`.

After the `postLaunch` event, the `SalesforceSDKManager` object doesn't reappear until a user logout or user switch event occurs. For either of these events, `SalesforceSDKManager` notifies your app. At that point, you can reset your app's Mobile SDK state and restart your app.

SalesforceSDKManager Launch Events

`SalesforceSDKManager` directs the app's bootstrap process according to the state of the app and the device. During the bootstrap process, several events fire at important points in the launch sequence. You can use these events to run your own logic after the `SalesforceSDKManager` flow is complete. For foregrounding, be sure to wait until your app receives the `postAppForeground` event before you resume your app's logic.

Table 2: Launch Events

Event	Description
`postLaunch`	Arrives after all launch activities have completed. The app can proceed with its business processes.
`launchError`	Sent if fatal errors occur during the launch process.
`postLogout`	Arrives after the current user has logged out, or if the user fails the passcode test or the login authentication.
`postAppForeground`	Arrives after the app returns to the foreground and the passcode (if applicable) has been verified. This event indicates that authentication is valid. After your app receives this event, you can take additional actions to handle foregrounding.
`switchUser`	Arrives after the current user has changed.

Certain events supersede others. For example, if passcode validation fails during launch, the `postLogout` event fires, but the `postLaunch` event does not. Between priority levels, the higher ranking event fires in place of the lower ranking event. Here is the list of priorities, with 1 as the highest priority level:

Table 3: Launch Event Priority Levels

Priority Level	Events	Comments
1	`postLogout`, `switchUser`	These events supersede all others.
2	`postLaunch`, `launchError`	It's important to note that these events always supersede `postAppForeground`. For instance, if you send the app to the background and then return it to the foreground during login, `postLaunch` fires if login succeeds, but `postAppForeground` does not.
3	`postAppForeground`	This lowest ranking event can be supplanted by any of the other events.

SalesforceSDKManager Properties

You configure your app's launch behavior by setting `SalesforceSDKManager` properties in the `init` method of `AppDelegate`. These properties contain your app's startup configuration, including:

- Connected app identifiers
- Required OAuth scopes
- Authentication behavior and associated customizations

You're required to specify at least the connected app and OAuth scopes settings.

You also use `SalesforceSDKManager` properties to define handler blocks for launch events. Event handler properties are optional. If you don't define them, the app logs a runtime warning when the event occurs. In general, it's a good idea to provide implementations for these blocks so that you have better control over the app flow.

Another especially useful property is the optional `authenticateAtLaunch`. Set this property to `NO` if you need to defer Salesforce authentication until some point after the app has started running. You can run the authentication process at any point by sending the `authenticate` message to `SalesforceSDKManager`. However, you must still always set the launch properties in the `init` method of `AppDelegate` and send the `launch` message to `SalesforceSDKManager` in the `application:didFinishLaunchingWithOptions:` method.

The following table describes `SalesforceSDKManager` properties.

Table 4: SalesforceSDKManager Properties

Property	Description
`connectedAppId`	(**Required**) The consumer ID from the associated Salesforce connected app.
`connectedAppCallbackUri`	(**Required**) The Callback URI from the associated Salesforce connected app.
`authScopes`	(**Required**) The OAuth scopes required for the app.
`postLaunchAction`	(**Required**) Controls how the app resumes functionality after launch completes.
`authenticateAtLaunch`	(Optional) Indicates whether `SalesforceSDKManager` should attempt authentication at launch. Defaults to YES. Set this value to NO if you want to defer authentication to a different stage of your application, and send the `authenticate` message to `SalesforceSDKManager` to initiate authentication at the appropriate time.
`launchErrorAction`	(Optional) If defined, this block responds to any errors that occur during the launch process.
`postLogoutAction`	(Optional) If defined, this block is executed when the current user has logged out.
`switchUserAction`	(Optional) If defined, this block handles a switch from the current user to an existing or new user. Note: This property is required if your app supports user switching.

Property	Description
`postAppForegroundAction`	(Optional) If defined, this block is executed after Mobile SDK finishes its post-foregrounding tasks.
`useSnapshotView`	(Optional) Set to YES if you plan to use a snapshot view when your app is in the background. This view obscures sensitive content in the app preview screen that displays when the user browses background apps from the home screen. Default is YES.
`snapshotView`	(Optional) Specifies the view that obscures sensitive app content from home screen browsing when your app is in the background. The default view is a white opaque screen.
`preferredPasscodeProvider`	(Optional) You can configure a different passcode provider to use a different passcode encryption scheme. Default is the Mobile SDK PBKDF2 provider.

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. It initializes Mobile SDK by using the shared `SalesforceSDKManager` object to oversee the app launch flow.

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 subview.

Setup

To customize the `AppDelegate` template, start by resetting the following static variables to values from your Force.com Connected Application:

- `RemoteAccessConsumerKey`

  ```
  static NSString * const RemoteAccessConsumerKey =
  @"3MVG9Iu66FKeHhINkB1l7xt7kR8...YFDUpqRWcoQ2.dBv_a1Dyu5xa";
  ```

  This variable corresponds to the Consumer Key in your connected app.

- `OAuthRedirectURI`

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

  This variable corresponds to the Callback URL in your connected app.

Initialization

The following listing shows the `init` method as implemented by the template app. It is followed by a call to the `launch` method of `SalesforceSDKManager` in the `application:didFinishLaunchingWithOptions:` method.

```
- (id)init
{
```

```
    self = [super init];
    if (self) {
        [SalesforceSDKManager sharedManager].connectedAppId = RemoteAccessConsumerKey;
        [SalesforceSDKManager sharedManager].connectedAppCallbackUri =
            OAuthRedirectURI;
        [SalesforceSDKManager sharedManager].authScopes = @[ @"web", @"api" ];
        __weak AppDelegate *weakSelf = self;
        [SalesforceSDKManager sharedManager].postLaunchAction =
            ^(SFSDKLaunchAction launchActionList) {
            [weakSelf log:SFLogLevelInfo
                   format:@"Post-launch: launch actions taken: %@",
                          [SalesforceSDKManager
                               launchActionsStringRepresentation:launchActionList]];
            [weakSelf setupRootViewController];
        };
        [SalesforceSDKManager sharedManager].launchErrorAction =
            ^(NSError *error, SFSDKLaunchAction launchActionList) {
                [weakSelf log:SFLogLevelError
                     format:@"Error during SDK launch: %@", [error localizedDescription]];

            [weakSelf initializeAppViewState];
            [[SalesforceSDKManager sharedManager] launch];
        };
        [SalesforceSDKManager sharedManager].postLogoutAction = ^{
            [weakSelf handleSdkManagerLogout];
        };
        [SalesforceSDKManager sharedManager].switchUserAction =
            ^(SFUserAccount *fromUser, SFUserAccount *toUser) {
                [weakSelf handleUserSwitch:fromUser toUser:toUser];
        };
    return self;
}

- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [[SalesforceSDKManager sharedManager] launch];
}
```

In the `init` method, the `SalesforceSDKManager` object:

- Initializes configuration items, such as Connected App identifiers amd OAuth scopes, using the `SalesforceSDKManager` shared instance. For example:

  ```
  [SalesforceSDKManager sharedManager].connectedAppId = RemoteAccessConsumerKey;
  [SalesforceSDKManager sharedManager].connectedAppCallbackUri = OAuthRedirectURI;
  [SalesforceSDKManager sharedManager].authScopes = @[ @"web", @"api" ];
  ```

- Assigns code blocks to properties that handle the `postLaunchAction`, `launchErrorAction`, `postLogoutAction`, and `switchUserAction` events. Notice the use of weak self in the block implementations. Besides protecting the code against cycles, this usage demonstrates an important point: `SalesforceSDKManager` is just a manager—any real work requiring a persistent self occurs within the delegate methods that actually perform the task. The following table summarizes how the `AppDelegate` template handles each event.

Event	Delegate Method	Default Behavior
`postLaunch`	`setupRootViewController`	Instantiates the controller for the app's root view and assigns it to the `window.rootViewController` property of `AppDelegate`.
`launchError`	`initializeAppViewState`	Resets the root view controller to the initial view controller.
`postLogout`	`handleSdkManagerLogout`	If there are multiple active user accounts, changes the root view controller to the multi-user view controller to allow the user to choose a previously authenticated account. If there is only one active account, automatically switches to that account. If there are no active accounts, presents the login screen.
`switchUser`	`handleUserSwitch:toUser:`	Resets the root view controller to the initial view controller, and then re-initiates the launch flow.

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). The template app uses this method to:

- Initialize the `window` property
- Set the root view controller to the initial view controller (see `initializeAppViewState`)
- Display the initial window
- Initiate authentication by sending the launch message to the shared `SalesforceSDKManager` instance.

`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.

`application:didRegisterForRemoteNotificationsWithDeviceToken:`,
`application:didFailToRegisterForRemoteNotificationsWithError:`

Used for handling incoming push notifications from Salesforce.

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

About Deferred Login

You can defer user login authentication to any logical point after the `postLaunch` event occurs. To defer authentication:

1. In the `init` method of your `AppDelegate` class, set the `authenticateAtLaunch` property of `SalesforceSDKManager` to `NO`.
2. Send the `launch` method to `SalesforceSDKManager`.
3. Call the `loginWithCompletion:failure:` method of `SFAuthenticationManager` at the point of deferred login.

If you defer authentication, the logic that handles login completions and failures is left to your app's discretion.

Upgrading Existing Apps

If you're upgrading an app from Mobile SDK 2.3 or earlier, you can reuse any custom code that handles launch events, but you'll have to move it to slightly different contexts. For example, code that formerly implemented the `authManagerDidLogout:` method of `SFAuthenticationManagerDelegate` now goes into the `postLogoutAction` block of `SalesforceSDKManager`. Likewise, code that implemented the `useraccountManager:didSwitchFromUser:toUser:` method of `SFUserAccountManagerDelegate` now belongs in the `switchUserAction` block of `SalesforceSDKManager`.

Finally, in your `AppDelegate` implementation, replace all calls to the `loginWithCompletion:failure:` method of `SFAuthenticationManager` with the `launch` method of `SalesforceSDKManager`. Move the code in your completion block to the `postLaunchAction` property, and move the failure block code to the `launchErrorAction` property.

SEE ALSO:

Using Push Notifications in iOS

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. File requests are defined in the `SFRestAPI (Files)` category and are documented in `SFRestAPI (Files)` Category.

Supported operations are:

Operation	Delegate method	Block method

Manual REST request
Executes a request that you've built
	`send:delegate:`	`sendRESTRequest:`
		`failBlock:`
		`completeBlock:`

SOQL query
Executes the given SOQL string and returns the resulting data set
	`requestForQuery:`	`performSOQLQuery:`
		`failBlock:`
		`completeBlock:`

SOSL search
Executes the given SOSL string and returns the resulting data set
	`requestForSearch:`	`performSOSLSearch:`
		`failBlock:`
		`completeBlock:`

Search Scope and Order
Executes a request to get a search result layout
	`requestForSearchResultLayout:`	`performRequestForSearchResultLayout:`
		`failBlock:`
		`completeBlock:`

Search Scope and Order
Executes a request to get search scope and order
	`requestForSearchScopeAndOrder`	`performRequestForSearchScopeAndOrderWithFailBlock:`
		`failBlock:`
		`completeBlock:`

Metadata
Returns the object's metadata
	`requestForMetadataWithObjectType:`	`performMetadataWithObjectType:`
		`failBlock:`
		`completeBlock:`

Describe global Returns a list of all available objects in your org and their metadata	`requestForDescribeGlobal`	`performDescribeGlobalWithFailBlock:` `completeBlock:`

Describe with object type
Returns a description of a single object type
	`requestForDescribeWithObjectType:`	`performDescribeWithObjectType:`
		`failBlock:`
		`completeBlock:`

Retrieve	`requestForRetrieveWithObjectType:`	`performRetrieveWithObjectType:` `objectId:`

Operation	Delegate method	Block method
Retrieves a single record by object ID	`objectId:` `fieldList:`	`fieldList:` `failBlock:completeBlock:`

Update
Updates an object with the given map
	`requestForUpdateWithObjectType:`
	`objectId:`
	`fields:`
		`performUpdateWithObjectType:`
		`objectId:`
		`fields:`
		`failBlock:`
		`completeBlock:`

Upsert
Updates or inserts an object from external data, based on whether
	`requestForUpsertWithObjectType:`
	`externalIdField:`
	`externalId:`
		`performUpsertWithObjectType:`
		`externalIdField:`
		`externalId:`
		`fields:`
		`failBlock:`
		`completeBlock:`

Operation	Delegate method	Block method
the external ID currently exists in the external ID field	`fields:`	

Create
Creates a new record in the specified object
	`requestForCreateWithObjectType:`
	`fields:`
		`performCreateWithObjectType:`
		`fields:`
		`failBlock:`
		`completeBlock:`

Delete
Deletes the object of the given type with the given ID
	`requestForDeleteWithObjectType:`
	`objectId:`
		`performDeleteWithObjectType:`
		`objectId:`
		`failBlock:`
		`completeBlock:`

Versions Returns Salesforce version metadata	`requestForVersions`	`performRequestForVersionsWithFailBlock:` `completeBlock:`
Resources Returns available resources for the specified API version, including resource name and URI	`requestForResources`	`performRequestForResourcesWithFailBlock:` `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

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];
```

6. To set an HTTP header for your request, use the `setHeaderValue:forHeaderName` method. This method can help you when you're displaying Chatter feeds, which come pre-encoded for HTML display. If you find that your native app displays unwanted escape sequences in Chatter comments, set the `X-Chatter-Entity-Encoding` header to "false" before sending your request, as follows:

```
...
[request setHeaderValue:@"false" forHeaderName:@"X-Chatter-Entity-Encoding"];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

Unauthenticated REST Requests

In certain cases, some applications must make REST calls before the user becomes authenticated. In other cases, the application must access services outside of Salesforce that don't require Salesforce authentication. To configure your `SFRestRequest` instance so that it doesn't require an authentication token, set its `requiresAuthentication` property to `NO`.

Note: Unauthenticated REST requests require a full path URL. Mobile SDK doesn't prepend an instance URL to unauthenticated endpoints.

Example:

```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForVersions];
request.requiresAuthentication = NO;
```

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.

Parameter name	Description
`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:

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

`SFRestAPI (Files)` Category

The `SFRestAPI (Files)` category provides methods that create file operation requests. Each method returns a new `SFRestRequest` object. Applications send this object to the Salesforce service to process the request. For example, the following code snippet calls the `requestForOwnedFilesList:page:` method to retrieve a `SFRestRequest` object. It then sends the request object to the server, specifying its owning object as the delegate that receives the response.

```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForOwnedFilesList:nil page:0];
[[SFRestAPI sharedInstance] send:request delegate:self];
```

Note: This example passes nil to the first parameter (`userId`). This value tells the `requestForOwnedFilesList:page:` method to use the ID of the context, or logged in, user. Passing 0 to the `pageNum` parameter tells the method to fetch the first page.

See Files and Networking for a full description of the Files feature and networking functionality.

Methods

`SFRestAPI (Files)` category supports the following operations. For a full reference of this category, see SFRestAPI (Files) Category—Request Methods (iOS). For a full description of the REST request and response bodies, go to **Chatter REST API Resources** > **FilesResources** at http://www.salesforce.com/us/developer/docs/chatterapi.

Method	Description
`- (SFRestRequest *) requestForOwnedFilesList:(NSString *) userId page:(NSUInteger)page`	Builds a request that fetches a page from the list of files owned by the specified user.

Method	Description
`- (SFRestRequest *) requestForFilesInUsersGroups:(NSString *)userId page:(NSUInteger)page`	Builds a request that fetches a page from the list of files owned by the user's groups.
`- (SFRestRequest *) requestForFilesSharedWithUser:(NSString *)userId page:(NSUInteger)page`	Builds a request that fetches a page from the list of files that have been shared with the user.
`- (SFRestRequest *) requestForFileDetails:(NSString *)sfdcId forVersion:(NSString *)version`	Builds a request that fetches the file details of a particular version of a file.
`- (SFRestRequest *) requestForBatchFileDetails:(NSArray *)sfdcIds`	Builds a request that fetches the latest file details of one or more files in a single request.
`- (SFRestRequest *) requestForFileRendition:(NSString *)sfdcId version:(NSString *)version renditionType:(NSString *)renditionType page:(NSUInteger)page`	Builds a request that fetches the a preview/rendition of a particular page of the file (and version).
`- (SFRestRequest *) requestForFileContents:(NSString *) sfdcId version:(NSString*) version`	Builds a request that fetches the actual binary file contents of this particular file.
`- (SFRestRequest *) requestForFileShares:(NSString *)sfdcId page:(NSUInteger)page`	Builds a request that fetches a page from the list of entities that share this file.
`- (SFRestRequest *) requestForAddFileShare:(NSString *)fileId entityId:(NSString *)entityId shareType:(NSString*)shareType`	Builds a request that add a file share for the specified file ID to the specified entity ID.
`- (SFRestRequest *) requestForDeleteFileShare:(NSString *)shareId;`	Builds a request that deletes the specified file share.
`- (SFRestRequest *) requestForUploadFile:(NSData *)data`	Builds a request that uploads a new file to the server. Creates a new file with version set to 1.

Method	Description

`name:(NSString *)name`
`description:(NSString *)description`
`mimeType:(NSString *)mimeType`	

Handling Authentication Errors

Mobile SDK provides default error handlers that display messages and divert the app flow when authentication errors occur. These error handlers are instances of the `SFAuthErrorHandler` class. They're managed by the `SFAuthErrorHandlerList` class, which stores references to all authentication error handlers. Error handlers define their implementation in anonymous blocks that use the following prototype:

```
typedef BOOL (^SFAuthErrorHandlerEvalBlock)(NSError *, SFOAuthInfo *);
```

A return value of `YES` indicates that the handler was used for the current error condition, and none of the other error handlers apply. If the handler returns `NO`, the block was not used, and the error handling process continues to the next handler in the list. Implementation details for error handlers are left to the developer's discretion. To see how the Mobile SDK defines these blocks, look at the `SFAuthenticationManager.m` file in the `SalesforceSDKCore` project.

To substitute your own error handling mechanism, you can:

- Override the Mobile SDK default error handler by adding your own handler to the top of the error handler stack (at index 0):

```
SFAuthErrorHandler *authErrorHandler =
    [[SFAuthErrorHandler alloc] initWithName:@"myAuthErrorHandler"
    evalBlock:^BOOL(NSError *error, SFOAuthInfo *authInfo) {
        // Add your error-handling code here
    }];
[[SFAuthenticationManager sharedManager].authErrorHandlerList
addAuthErrorHandler:authErrorHandler atIndex:0];
```

- Remove the Mobile SDK generic "catch-all" error handler from the list. This causes authentication errors to fall through to the `launchErrorAction` block of your `SalesforceSDKManager` implementation during the launch process, or to the `failure:` block of your `loginWithCompletion:failure:` definition if you've implemented deferred login. Here's how you disable the generic error handler:

```
SFAuthErrorHandler *genericHandler =
    [SFAuthenticationManager sharedManager].genericAuthErrorHandler;
[[SFAuthenticationManager sharedManager].authErrorHandlerList
    removeAuthErrorHandler:genericHandler];
```

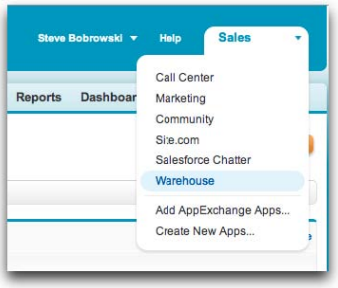
Tutorial: Creating a Native iOS Warehouse App

Prerequisites

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

  1. Click the installation URL link: http://bit.ly/package100

2. If you aren't logged in already, enter the username and password of your DE org.
3. On the Package Installation Details page, click **Continue**.
4. Click **Next**, and on the Security Level page click **Next**.
5. Click **Install**.
6. Click **Deploy Now** and then **Deploy**.
7. Once the installation completes, you can select the **Warehouse** app from the app picker in the upper right corner.



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

- Install the latest versions of Xcode and the iOS SDK.
- Install the Salesforce Mobile SDK using npm:
  1. If you've already successfully installed Node.js and npm, skip to step 4.
  2. Install Node.js on your system. The Node.js installer automatically installs npm.
     i. Download Node.js from www.nodejs.org/download.
     ii. Run the downloaded installer to install Node.js and npm. Accept all prompts asking for permission to install.
  3. At the Terminal window, type `npm` and press `Return` to make sure your installation was successful. If you don't see a page of usage information, revisit Step 2 to find out what's missing.
  4. At the Terminal window, type `sudo npm install forceios -g`

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

Create a Native iOS App

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

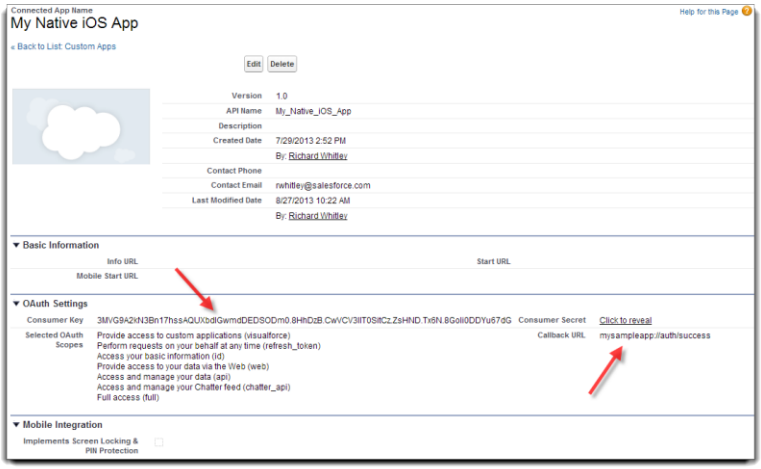
Step 1: Create a Connected App

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

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

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

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



Step 2: Create a Native iOS Project

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

1. Change to the directory in which you want to create your project.
2. To create an iOS project, type *forceios create*.

   The forceios utility prompts you for each configuration value.
3. For application type, enter *native*.
4. For application name, enter *MyNativeiOSApp*.
5. For company identifier, enter *com.acme.goodapps*.
6. For organization name, enter *GoodApps, Inc.*.
7. For output directory, enter *tutorial/iOSNative*.
8. For the Connected App ID, copy and paste the Consumer Key from your Connected App definition.

9. For the Connected App Callback URI, copy and paste the Callback URL from your Connected App definition.

The input screen should look similar to this:

```
rwhitley-ltm1:~ 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): tutorial/iOSNative
Enter your Connected App ID (defaults to the sample app's ID): 3MVG9A2kN3Bn17hssAQUXbdlGwmdDEDSODm0.8HhDzB.
CwVCV3llT0SitCz.ZsHND.Tx6N.8Goli0DDYu67dG
Enter your Connected App Callback URI (defaults to the sample app's URI): https://login.salesforce.com/serv
ices/oauth2/success
Creating output folder tutorial/iOSNative
Creating app in /Users/rwhitley/tutorial/iOSNative/MyNativeiOSApp
Successfully created native app 'MyNativeiOSApp'.
```

Step 3: Run the New iOS App

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 the iOS simulator. Make sure that you've selected **Product** > **Destination** > **iPhone 6.0 Simulator** in the Xcode menu.
5. When you start the app, after showing an initial splash screen, you should see the Salesforce login screen. Login with your DE username and password.

6. When prompted, click **Allow** to let the app access your data in Salesforce. You should see a table listing the names of users defined in your DE org.

Step 4: Explore How the iOS App Works

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

- The model is the Force.com database schema
- The views come from the nib and implementation files in your project
- The controller functionality represents a joint effort between the iOS SDK classes, the Salesforce Mobile SDK, and your app

AppDelegate Class and the Root View Controller

When the app is launched, the `AppDelegate` class initially controls the execution flow. After the login process completes, the `AppDelegate` instance passes control to the root view. In the template app, the root view controller class is named `RootViewController`. This class becomes the root view for the app in the `AppDelegate.m` file, where it's subsumed by a `UINavigationController` instance that controls navigation between views:

```
- (void)setupRootViewController
{
    RootViewController *rootVC = [[RootViewController alloc] initWithNibName:nil bundle:nil];

    UINavigationController *navVC = [[UINavigationController alloc]
initWithRootViewController:rootVC];
```

```
    self.window.rootViewController = navVC;
}
```

Before it's customized, though, the app doesn't include other views or touch event handlers. It simply logs into Salesforce, issues a request using Salesforce Mobile SDK REST APIs, and displays the response in the root view.

UITableViewController Class

`RootViewController` inherits the `UITableViewController` class. Because it doesn't customize the table in its inherited view, there's no need for a nib or xib file. The controller class simply loads data into the `tableView` property and lets the super class handle most of the display tasks. However, `RootViewController` does add some basic cell formatting by calling the `tableView:cellForRowAtIndexPath:` method. It creates a new cell, assigns it a generic ID (`@"CellIdentifier"`), puts an icon at the left side of the cell, and adds an arrow to the right side. Most importantly, it sets the cell's label to assume the Name value of the current row from the REST response object. Here's the code:

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView_ cellForRowAtIndexPath:(NSIndexPath
 *)indexPath {
   static NSString *CellIdentifier = @"CellIdentifier";

   // Dequeue or create a cell of the appropriate type.
   UITableViewCell *cell = [tableView_ dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleValue1
reuseIdentifier:CellIdentifier] autorelease];

    }
 //if you want to add an image to your cell, here's how
 UIImage *image = [UIImage imageNamed:@"icon.png"];
 cell.imageView.image = image;

 // Configure the cell to show the data.
 NSDictionary *obj = [self.dataRows objectAtIndex:indexPath.row];
 cell.textLabel.text =  [obj objectForKey:@"Name"];

 //this adds the arrow to the right hand side.
 cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

 return cell;

}
```

SFRestAPI Shared Object and SFRestRequest Class

You can learn how the app creates and sends the REST request by browsing the `RootViewController.viewDidLoad` method. The app defines a literal SOQL query string and passes it to the `SFRestAPI:requestForQuery:` instance method. To call this method, the app sends a message to the shared singleton `SFRestAPI` instance. The method creates and returns an appropriate, pre-formatted `SFRestRequest` object that wraps the SOQL query. The app then forwards this object to the server by sending the `send:delegate:` message to the shared `SFRestAPI` object:

```
    //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 `SFRestAPI` class serves as a factory for `SFRestRequest` instances. It defines a series of request methods that you can call to easily create request objects. If you want, you can also build `SFRestRequest` instances directly, but, for most cases, manual construction isn't necessary.

Notice that the app specifies `self` for the `delegate` argument. This tells the server to send the response to a delegate method implemented in the `RootViewController` class.

SFRestDelegate Interface

To be able to accept REST responses, `RootViewController` implements the `SFRestDelegate` interface. This interface declares four methods—one for each possible response type. The `request:didLoadResponse:` delegate method executes when the request succeeds. When `RootViewController` receives a `request:didLoadResponse:` callback, it copies the returned records into its data rows and reloads the data displayed in the Warehouse view. Here's the code that implements the `SFRestDelegate` interface in the `RootViewController` class:

```
#pragma mark - SFRestDelegate

- (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];
}


- (void)request:(SFRestRequest*)request didFailLoadWithError:(NSError*)error {
    NSLog(@"request:didFailLoadWithError: %@", error);
    //add your failed error handling here
}

- (void)requestDidCancelLoad:(SFRestRequest *)request {
    NSLog(@"requestDidCancelLoad: %@", request);
    //add your failed error handling here
}

- (void)requestDidTimeout:(SFRestRequest *)request {
    NSLog(@"requestDidTimeout: %@", request);
    //add your failed error handling here
}
```

As the comments indicate, this code fully implements only the `request:didLoadResponse:` success delegate method. For responses other than success, this template app simply logs a message.

Customize the List Screen

In this tutorial, you modify the root view controller to make the app specific to the Warehouse schema. You also adapt the existing SOQL query to obtain all the information we need from the Merchandise custom object.

Step 1: Modify the Root View Controller

To adapt the template project to our Warehouse design, let's rename the `RootViewController` class.

1. In the Project Navigator, choose the `RootViewController.h` file.
2. In the Editor, click the name "RootViewController" on this line:

```
@interface RootViewController : UITableViewController <SFRestDelegate>{
```

3. Using the Control-Click menu, choose **Refactor** > **Rename**. Be sure that **Rename Related Files** is checked.
4. Change "RootViewController" to "WarehouseViewController". Click **Preview**.

   Xcode presents a new window that lists all project files that contain the name "RootViewController" on the left. The central pane shows a diff between the existing version and the proposed new version of each changed file.

5. Click **Save**.
6. Click **Enable** when Xcode asks you if you'd like it to take automatic snapshots before refactoring.

After the snapshot is complete, the Refactoring window goes away, and you're back in your refactored project. Notice that the file names `RootViewController.h` and `RootViewController.m` are now `WarehouseViewController.h` and `WarehouseViewController.m`. Every instance of `RootViewController` in your project code has also been changed to `WarehouseViewController`.

Step 2: Create the App's Root View

The native iOS template app creates a SOQL query that extracts Name fields from the standard User object. For this tutorial, though, you use records from a custom object. Later, you create a detail screen that displays Name, Quantity, and Price fields. You also need the record ID.

Let's update the SOQL query to operate on the custom Merchandise__c object and to retrieve the fields needed by the detail screen.

1. In the Project Navigator, select `WarehouseViewController.m`.
2. Scroll to the `viewDidLoad` method.
3. Update the view's display name to "Warehouse App". Change:

```
self.title = @"Mobile SDK Sample App"
```

   to

```
self.title = @"Warehouse App"
```

4. Change the SOQL query in the following line:

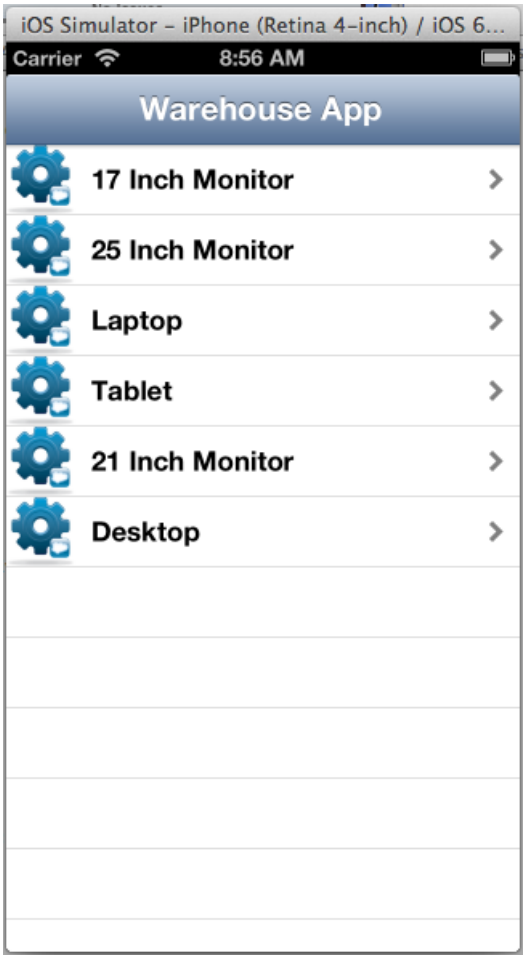
```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name
FROM User LIMIT 10"];
```

   to:

```
SELECT Name, Id, Quantity__c, Price__c FROM Merchandise__c LIMIT 10
```

Step 3:Try Out the App

Build and run the app. When prompted, log into your DE org. The initial page should look similar to the following screen.



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

Create the Detail Screen

In the previous tutorial, you modified the template app so that, after it starts, it lists up to ten Merchandise records. In this tutorial, you finish the job by creating a detail view and controller. You also establish communication between list view and detail view controllers.

Step 1: Create the App's Detail View Controller

When a user taps a Merchandise record in the Warehouse view, an `IBAction` generates record-specific information and then loads a view from `DetailViewController` that displays this information. However, this view doesn't yet exist, so let's create it.

1. Click **File** > **New** > **File...** > **Cocoa Touch** > **Objective-C class**.
2. Create a new Objective-C class named `DetailViewController` that subclasses `UIViewController`. Make sure that **With XIB for user interface** is checked.
3. Click **Next**.
4. Place the new class in the Classes group under Mobile Warehouse App in the **Groups** drop-down menu.

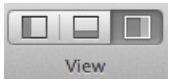
   Xcode creates three new files in the `Classes` folder: `DetailViewController.h`, `DetailViewController.m`, and `DetailViewController.xib`.

5. Select `DetailViewController.m` in the Project Navigator, and delete the following method declaration:

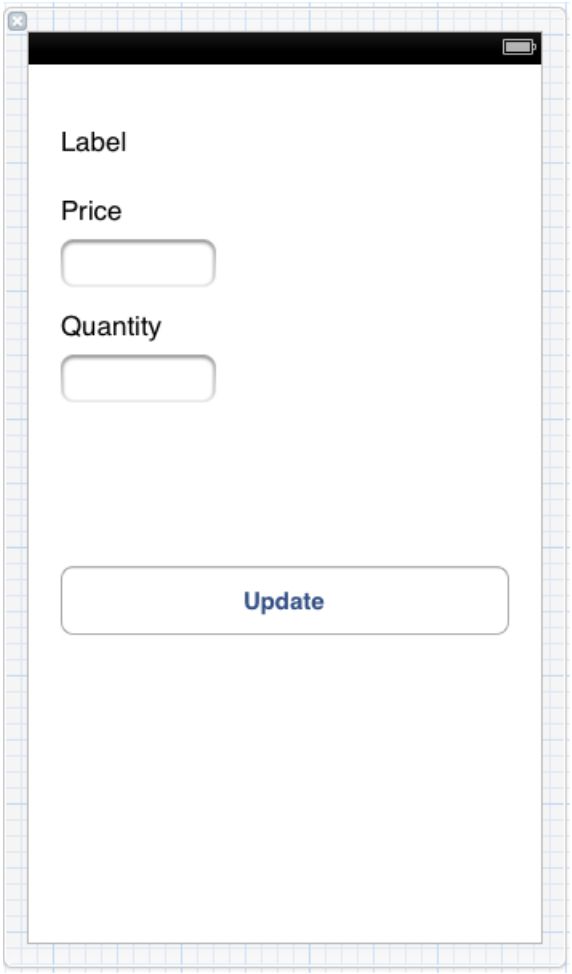
```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        // Custom initialization
    }
    return self;
}
```

   In this app, you don't need this initialization method because you're not specifying a NIB file or bundle.

6. Select `DetailViewController.xib` in the Project Navigator to open the Interface Builder.

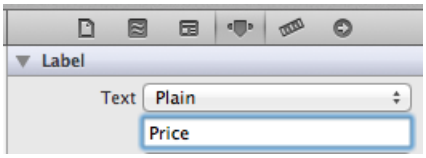
7. From the Utilities view 

, drag three labels, two text fields, and one button onto the view layout. Arrange and resize the controls so that the screen looks like this:



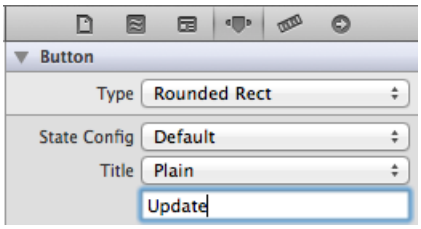
   We'll refer to topmost label as the Name label. This label is dynamic. In the next tutorial, you'll add controller code that resets it at runtime to a meaningful value.

8. In the Attributes inspector, set the display text for the static Price and Quantity labels to the values shown. Select each label individually in the Interface Builder and specify display text in the unnamed entry field below the Text drop-down menu.



Note: Adjust the width of the labels as necessary to see the full display text.

9. In the Attributes inspector, set the display text for the Update button to the value shown. Select the button in the Interface Builder and specify its display text in the unnamed entry field below the Title drop-down menu.



10. Build and run to check for errors. You won't yet see your changes.

The detail view design shows Price and Quantity fields, and provides a button for updating the record's Quantity. However, nothing currently works. In the next step, you learn how to connect this design to Warehouse records.

Step 2: Set Up DetailViewController

To establish connections between view elements and their view controller, you can use the Xcode Interface Builder to connect UI elements with code elements.

Add Instance Properties

1. Create properties in `DetailViewController.h` to contain the values passed in by the `WarehouseViewController`: Name, Quantity, Price, and Id. Place these properties within the `@interface` block. Declare each `nonatomic` and `strong`, using these names:

```
@interface DetailViewController : UIViewController

@property (nonatomic, strong) NSNumber *quantityData;
@property (nonatomic, strong) NSNumber *priceData;
@property (nonatomic, strong) NSString *nameData;
@property (nonatomic, strong) NSString *idData;


@end
```

2. In `DetailViewController.m`, just after the `@implementation` tag, synthesize each of the properties.

```
@implementation DetailViewController

@synthesize nameData;
@synthesize quantityData;
```

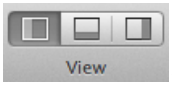
```
@synthesize priceData;
@synthesize idData;
```

Add IBOutlet Variables

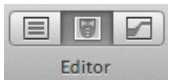
`IBOutlet` member variables let the controller manage each non-static control. Instead of coding these manually, you can use the Interface Builder to create them. Interface Builder provides an Assistant Editor that gives you the convenience of side-by-side editing windows. To make room for the Assistant Editor, you'll usually want to reclaim screen area by hiding unused controls.

1. In the Project Navigator, click the `DetailViewController.xib` file.

   The `DetailViewController.xib` file opens in the Standard Editor.

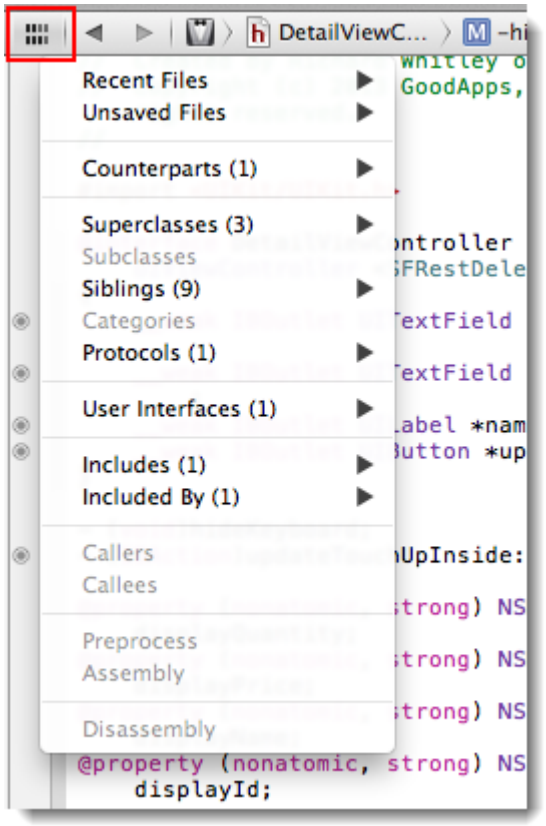
2. Hide the Navigator by clicking Hide or Show Navigator on the View toolbar 

. Alternatively, you can choose **View** > **Navigators** > **Hide Navigators** in the Xcode menu.

3. Open the Assistant Editor by clicking Show the Assistant editor in the Editor toolbar 

. Alternatively, you can choose **View** > **Assistant Editor** > **Show Assistant Editor** in the Xcode menu.

   Because you opened `DetailViewController.xib` in the Standard Editor, the Assistant Editor shows the `DetailViewController.h` file. The Assistant Editor guesses which files are most likely to be used together. If you need to open a different file, click the Related Files control in the upper left hand corner of the Assistant Editor



4. At the top of the interface block in `DetailViewController.h`, add a pair of empty curly braces:

```
@interface DetailViewController : UiViewController <SFRestDelegate>
{

}
```

5. In the Standard Editor, control-click the Price text field control and drag it into the new curly brace block in the `DetailViewController.h` file.
6. In the popup dialog box, name the new outlet `_priceField`, and click **Connect**.
7. Repeat steps 2 and 3 for the Quantity text field, naming its outlet `_quantityField`.
8. Repeat steps 2 and 3 for the Name label, naming its outlet `_nameLabel`.

Your interface code now includes this block:

```
@interface DetailViewController : UIViewController
{
    __weak IBOutlet UITextField *_priceField;
    __weak IBOutlet UITextField *_quantityField;
    __weak IBOutlet UILabel *_nameLabel;
}
```

Add an Update Button Event

1. In the Interface Builder, select the **Update** button and open the Connections Inspector

2. In the Connections Inspector, select the circle next to **Touch Up Inside** and drag it into the `DetailViewController.h` file. Be sure to drop it below the closing curly brace. Name it `updateTouchUpInside`, and click **Connect**.

   The Touch Up Inside event tells you that the user raised the finger touching the Update button without first leaving the button. You'll perform a record update every time this notification arrives.

Step 3: Create the Designated Initializer

Now, let's get down to some coding. Start by adding a new initializer method to `DetailViewController` that takes the name, ID, quantity, and price. The method name, by convention, must begin with "init".

1. Click **Show the Standard Editor** and open the Navigator.
2. Add this declaration to the `DetailViewController.h` file just above the `@end` marker:

```
- (id) initWithName:(NSString *)recordName
                        sobjectId:(NSString *)salesforceId
                        quantity:(NSNumber *)recordQuantity
                              price:(NSNumber *)recordPrice;
```

   Later, we'll code `WarehouseViewController` to use this method for passing data to the `DetailViewController`.

3. Open the `DetailViewController.m` file, and copy the signature you created in the previous step to the end of the file, just above the `@end` marker.

4. Replace the terminating semi-colon with a pair of curly braces for your implementation block.

```
- (id) initWithName:(NSString *)recordName
                          sobjectId:(NSString *)salesforceId
                          quantity:(NSNumber *)recordQuantity
                                price:(NSNumber *)recordPrice {
}
```

5. In the method body, send an `init` message to the super class. Assign the return value to `self`:

```
self = [super init];
```

This `init` message gives you a functional object with base implementation which will serve as your return value.

6. Add code to verify that the super class initialization succeeded, and, if so, assign the method arguments to the corresponding instance variables. Finally, return `self`.

```
if (self) {
    self.nameData = recordName;
    self.idData = salesforceId;
    self.quantityData = recordQuantity;
    self.priceData = recordPrice;
}
return self;
```

Here's the completed method:

```
- (id) initWithName:(NSString *)recordName
                          sobjectId:(NSString *)salesforceId
                          quantity:(NSNumber *)recordQuantity
                                price:(NSNumber *)recordPrice {
    self = [super init];
    if (self) {
        self.nameData = recordName;
        self.idData = salesforceId;
        self.quantityData = recordQuantity;
        self.priceData = recordPrice;
    }
    return self;
}
```

7. To make sure the controls are updated each time the view appears, add a new `viewWillAppear:` event handler after the `viewDidLoad` method implementation. Begin by calling the super class method.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

}
```

8. Copy the values of the property variables to the corresponding dynamic controls.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [_nameLabel setText:self.nameData];
    [_quantityField setText:[self.quantityData stringValue]];
```

```
    [_priceField setText:[self.priceData stringValue]];
}
```

9. Build and run your project to make sure you've coded everything without compilation errors. The app will look the same as it did at first, because you haven't yet added the code to launch the Detail view.

Note: The `[super init]` message used in the `initWithName:` method calls `[super initWithNibName:bundle:]` internally. We use `[super init]` here because we're not passing a NIB name or a bundle. If you are specifying these resources in your own projects, you'll need to call `[super initWithNibName:bundle:]` explicitly.

Step 4: Establish Communication Between the View Controllers

Any view that consumes Salesforce content relies on a `SFRestAPI` delegate to obtain that content. You can designate a single view to be the central delegate for all views in the app, which requires precise communication between the view controllers. For this exercise, let's take a slightly simpler route: Make `WarehouseViewController` and `DetailViewController` each serve as its own `SFRestAPI` delegate.

Update WarehouseViewController

First, let's equip `WarehouseViewController` to pass the quantity and price values for the selected record to the detail view, and then display that view.

1. In `WarehouseViewController.m`, above the `@implementation` block, add the following line:

```
#import "DetailViewController.h"
```

2. On a new line after the `#pragma mark - Table view data source` marker, type the following starter text to bring up a list of `UITableView` delegate methods:

```
- (void)tableView
```

3. From the list, select the `tableView:didSelectRowAtIndexPath:` method.
4. Change the `tableView` parameter name to `itemTableView`.

```
- (void)tableView:(UITableView *)itemTableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath
```

5. At the end of the signature, type an opening curly brace ({) and press return to stub in the method implementation block.
6. At the top of the method body, per standard iOS coding practices, add the following call to deselect the row.

```
    [itemTableView deselectRowAtIndexPath:indexPath animated:NO];
```

7. Next, retrieve a pointer to the `NSDictionary` object associated with the selected data row.

```
NSDictionary *obj = [self.dataRows objectAtIndex:indexPath.row];
```

8. At the end of the method body, create a local instance of `DetailViewController` by calling the `DetailViewController.initWithName:salesforceId:quantity:price:` method. Use the data stored in the `NSDictionary` object to set the name, Salesforce ID, quantity, and price arguments. The finished call looks like this:

```
DetailViewController *detailController =
    [[DetailViewController alloc] initWithName:[obj objectForKey:@"Name"]
                                    salesforceId:[obj objectForKey:@"Id"]
                                        quantity:[obj objectForKey:@"Quantity__c"]
                                           price:[obj objectForKey:@"Price__c"]];
```

9. To display the Detail view, add code that pushes the initialized `DetailViewController` onto the `UINavigationController` stack:

```
    [[self navigationController] pushViewController:detailController animated:YES];
```

   Great! Now you're using a `UINavigationController` stack to handle a set of two views. The root view controller is always at the bottom of the stack. To activate any other view, you just push its controller onto the stack. When the view is dismissed, you pop its controller, which brings the view below it back into the display.

10. Build and run your app. Click on any Warehouse item to display its details.

Add Update Functionality

Now that the `WarehouseViewController` is set up, we need to modify the `DetailViewController` class to send the user's updates to Salesforce via a REST request.

1. In the `DetailViewController.h` file, add an instance method to `DetailViewController` that lets a user update the price and quantity fields. This method needs to send a record ID, the names of the fields to be updated, the new quantity and price values, and the name of the object to be updated. Add this declaration after the interface block and just above the `@end` marker.

```
- (void)updateWithObjectType:(NSString *)objectType
              objectId:(NSString *)objectId
               quantity:(NSString *)quantity
               price:(NSString *)price;
```

   To implement the method, you create an `SFRestRequest` object using the input values, then send the request object to the shared instance of the `SFRestAPI`.

2. In the `DetailViewController.m` file, add the following line above the `@implementation` block.

```
#import "SFRestAPI.h"
```

3. At the end of the file, just above the `@end` marker, copy the `updateWithObjectType:objectId:quantity:price:` signature, followed by a pair of curly braces:

```
- (void)updateWithObjectType:(NSString *)objectType
              objectId:(NSString *)objectId
               quantity:(NSString *)quantity
               price:(NSString *)price {

}
```

4. In the implementation block, create a new `NSDictionary` object to contain the Quantity and Price fields. To allocate this object, use the `dictionaryWithObjectsAndKeys: ... NSDictionary` class method with the desired list of fields.

```
- (void)updateWithObjectType:(NSString *)objectType
              objectId:(NSString *)objectId
               quantity:(NSString *)quantity
               price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                                          quantity, @"Quantity__c",
                                          price, @"Price__c",
                                          nil];

}
```

5. Create a `SFRestRequest` object. To allocate this object, use the `requestForUpdateWithObjectType:objectId:fields:` instance method on the `SFRestAPI` shared instance.

```
- (void)updateWithObjectType:(NSString *)objectType
              objectId:(NSString *)objectId
               quantity:(NSString *)quantity
               price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                                          quantity, @"Quantity__c",
                                          price, @"Price__c",
                                          nil];
    SFRestRequest *request =
        [[SFRestAPI sharedInstance]
            requestForUpdateWithObjectType:objectType
                                  objectId:objectId
                                    fields:fields];

}
```

6. Finally, send the new `SFRestRequest` object to the service by calling `send:delegate:` on the `SFRestAPI` shared instance. For the `delegate` argument, be sure to specify `self`, since `DetailViewController` is the `SFRestDelegate` in this case.

```
- (void)updateWithObjectType:(NSString *)objectType
              objectId:(NSString *)objectId
               quantity:(NSString *)quantity
               price:(NSString *)price {

    NSDictionary *fields = [NSDictionary dictionaryWithObjectsAndKeys:
                                          quantity, @"Quantity__c",
                                          price, @"Price__c",
                                          nil];
    SFRestRequest *request =
        [[SFRestAPI sharedInstance]
            requestForUpdateWithObjectType:objectType
                                  objectId:objectId
                                    fields:fields];
```

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

7. Edit the `updateTouchUpInside:` action method to call the `updateWithObjectType:objectId:quantity:price:` method when the user taps the **Update** button.

```
- (IBAction)updateTouchUpInside:(id)sender {
    // For Update button
    [self updateWithObjectType:@"Merchandise__c"
                      objectId:self.idData
                      quantity:[_quantityField text]
                         price:[_priceField text]];}
```

Note:

- **Extra credit:** Improve your app's efficiency by performing updates only when the user has actually changed the quantity value.

Add SFRestDelegate to DetailViewController

We're almost there! We've issued the REST request, but still need to provide code to handle the response.

1. Open the `DetailViewController.h` file and change the `DetailViewController` interface declaration to include `<SFRestDelegate>`

```
@interface DetailViewController : UIViewController <SFRestDelegate>
```

2. Open the `WarehouseViewController.m` file.
3. Find the `pragma` that marks the `SFRestAPIDelegate` section.

```
#pragma mark - SFRestAPIDelegate
```

4. Copy the four methods under this pragma into the `DetailViewController.m` file.

```
- (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];
}

- (void)request:(SFRestRequest*)request didFailLoadWithError:(NSError*)error {
    NSLog(@"request:didFailLoadWithError: %@", error);
    //add your failed error handling here
}

- (void)requestDidCancelLoad:(SFRestRequest *)request {
    NSLog(@"requestDidCancelLoad: %@", request);
    //add your failed error handling here
}

- (void)requestDidTimeout:(SFRestRequest *)request {
    NSLog(@"requestDidTimeout: %@", request);
```

```
    //add your failed error handling here
}
```

These methods are all we need to implement the `SFRestAPI` interface. For this tutorial, we can retain the simplistic handling of error, cancel, and timeout conditions. However, we need to change the `request:didLoadResponse:` method to suit the detail view purposes. Let's use the `UINavigationController` stack to return to the list view after an update occurs.

5. In the `DetailViewController.m` file, delete the existing code in the `request:didLoadResponse:` delegate method. In its place, add code that logs a success message and then pops back to the root view controller. The revised method looks like this.

```
- (void)request:(SFRestRequest *)request didLoadResponse:(id)jsonResponse {
    NSLog(@"1 record updated");
    [self.navigationController popViewControllerAnimated:YES];
}
```

6. Build and run your app. In the Warehouse view, click one of the items. You're now able to access the Detail view and edit its quantity, but there's a problem: the keyboard won't go away when you want it to. You need to add a little finesse to make the app truly functional.

Hide the Keyboard

The iOS keyboard remains visible as long as any text input control on the screen is responding to touch events. This is where the "First Responder" setting, which you might have noticed in the Interface Builder, comes into play. We didn't set a first responder because our simple app just uses the default UIKit behavior. As a result, iOS can consider any input control in the view to be the first responder. If none of the controls explicity tell iOS to hide the keyboard, it remains active.

You can resolve this issue by making every touch-enabled edit control resign as first responder.

1. In `DetailViewController.h`, below the curly brace block, add a new instance method named `hideKeyboard` that takes no arguments and returns `void`.

```
- (void)hideKeyboard;
```

2. In the implementation file, implement this method to send a `resignFirstResponder` message to each touch-enabled edit control in the view.

```
- (void)hideKeyboard {
    [_quantityField resignFirstResponder];
    [_priceField resignFirstResponder];
}
```

The only remaining question is where to call the `hideKeyboard` method. We want the keyboard to go away when the user taps outside of the text input controls. There are many likely events that we could try, but the only one that is sure to catch the background touch under all circumstances is `[UIResponder touchesEnded:withEvent:]`.

3. Since the event is already declared in a class that `DetailViewController` inherits, there's no need to re-declare it in the `DetailViewController.h` file. Rather, in the `DetailViewController.m` file, type the following incomplete code on a new line outside of any method body:

```
- (void)t
```

A popup menu displays with a list of matching instance methods from the `DetailViewController` class hierarchy.

Note: If the popup menu doesn't appear, just type the code described next.

4. In the popup menu, highlight the `touchesEnded:withEvent:` method and press *Return*. The editor types the full method signature into your file for you. Just type an opening brace, press *Return*, and your stub method is completed by the editor. Within this stub, send a `hideKeyboard` message to `self`.

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    [self hideKeyboard];
}
```

Normally, in an event handler, you'd be expected to call the super class version before adding your own code. As documented in the iOS Developer Library, however, leaving out the super call in this case is a common usage pattern. The only "gotcha" is that you also have to implement the other touches event handlers, which include:

```
- touchesBegan:withEvent:
- touchesMoved:withEvent:
- touchesCancelled:withEvent:
```

The good news is that you only need to provide empty stub implementations.

5. Use the Xcode editor to add these stubs the same way you added the `touchesEnded:` stub. Make sure your final code looks like this:

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event{
    [self hideKeyboard];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event{

}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event{

}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event{

}
```

Refreshing the Query with viewWillAppear

The `viewDidLoad` method lets you configure the view when it first loads. In the `WarehouseViewController` implementation, this method contains the REST API query that populates both the list view and the detail view. However, since `WarehouseViewController` represents the root view, the `viewDidLoad` notification is called only once—when the view is initialized. What does this mean? When a user updates a quantity in the detail view and returns to the list view, the query is not refreshed. Thus, if the user returns to the same record in the detail view, the updated value does not display, and the user is not happy.

You need a different method to handle the query. The `viewWillAppear` method is called each time its view is displayed. Let's add this method to `WarehouseViewController` and move the SOQL query into it.

1. In the `WarehouseViewController.m` file, add the following code after the `viewDidLoad` implementation.

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

}
```

2. Cut the following lines from the `viewDidLoad` method and paste them into the `viewWillAppear:` method, after the call to `super`:

```
    SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name,
 ID,
        Price__c, Quantity__c FROM Merchandise__c LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
```

   The final `viewDidLoad` and `viewWillAppear:` methods look like this.

```
- (void)viewDidLoad{
    [super viewDidLoad];
    self.title = @"Warehouse App";
}

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    //Here we use a query that should work on either Force.com or Database.com
    SFRestRequest *request = [[SFRestAPI sharedInstance] requestForQuery:@"SELECT Name,
 ID,
        Price__c, Quantity__c FROM Merchandise__c LIMIT 10"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
}
```

The `viewWillAppear:` method refreshes the query each time the user navigates back to the list view. Later, when the user revisits the detail view, the list view controller updates the detail view with the refreshed data.

Step 5: Try Out the App

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

iOS Native Sample Applications

The app you created in Run 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 demonstrate more functionality you can examine and work into your own apps.

- **RestAPIExplorer** exercises all native REST API wrappers. It resides in the Mobile SDK for iOS under `native/SampleApps/RestAPIExplorer`.
- **NativeSqlAggregator** shows SQL aggregation examples plus a native SmartStore implementation. It resides in the Mobile SDK for iOS under `native/SampleApps/NativeSqlAggregator`.
- **FileExplorer** demonstrates the Files API and the underlying network library. This sample resides in the Mobile SDK for iOS under `native/SampleApps/FileExplorer`.
- **SmartSyncExplorer** demonstrates the power of the native SmartSync library on iOS. It resides in the Mobile SDK for iOS under `native/SampleApps/SmartSyncExplorer`.

CHAPTER 4 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 the forcedroid application 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

Mobile SDK 3.2 Android development requires the following software.

- Java JDK 7 or higher—`http://www.oracle.com/downloads`.
- Apache Ant 1.8 or later—`http://ant.apache.org`.
- Minimum Android SDK is 4.2.2 (API level 17). Target Android SDK is 5.0.1 (API 21). The default and target Android SDK version for Mobile SDK hybrid apps is 5.0.1 (API 21). The minimum Android SDK version is 4.2.2 (API level 17) or above.
- Android SDK Tools, version 24 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—`https://www.eclipse.org`. Check the Android Development Tools website for the minimum supported Eclipse version.
- In order to run the application in the Emulator, you need to set up at least one Android Virtual Device (AVD) that targets Platform 4.2.2 (API level 17) and 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`.

On the Salesforce side, you'll also need:

- Salesforce Mobile SDK 3.0 for Android. See Install the Mobile SDK.
- A Salesforce Developer Edition organization with a connected app.

The `SalesforceSDK` project is built with the Android 4.2.2 (API level 17) library.

Creating an 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.

If you prefer video tutorials, see:

- "Installing Salesforce Mobile SDK For Android On Windows" at http://www.salesforce.com/_app/video/developer/help/MobileSDK_part1.jsp
- "Creating Native Android Apps With Salesforce Mobile SDK" at http://www.salesforce.com/_app/video/developer/help/MobileSDK_part2.jsp

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.

Specifying Application Options Directly

If you prefer, you can specify forceios parameters directly at the command line. 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)
   --targetandroidapi=<Target API> (e.g. 21 for Lollipop = Only required/used for 'native')

    --startpage=<Path to the remote start page> (/apex/MyPage — Only required/used for
'hybrid_remote')
    [--usesmartstore=<Whether or not to use SmartStore/SmartSync> ('yes' or 'no', 'no' by
 default -- Only required/used for 'native')]
```

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

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

Import and Build Your App in Eclipse

Use the following instructions to build and run your new app in the Eclipse editor.

1. Launch Eclipse and select your target directory as the workspace directory.
2. Select **Eclipse** > **Preferences**, choose the **Android** section, and enter your Android SDK location if it is not already set.
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 select the SalesforceSDK and Cordova projects.
7. If you set `–use_smartstore=yes`, select 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.

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. Before building the new application, build the SalesforceSDK project by running the following commands at the command prompt:

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

   where `SALESFORCE_SDK_DIR` points to your Salesforce SDK installation directory, and `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 for supported API levels.

2. Build the SmartStore project by running the following commands at the command prompt:

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

   where `SALESFORCE_SDK_DIR` points to your Salesforce SDK installation directory, and `ANDROID_SDK_DIR` points to your Android SDK directory.

3. To build the new application, run 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.

4. If your emulator is not running, use the Android AVD Manager to start it. If you're using a device, connect it.
5. 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.

SEE ALSO:

Forcedroid Parameters

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 your target directory as the workspace directory.
2. Select **Window** > **Preferences** or **Eclipse** > **Preferences**, choose the **Android** section, and enter the Android SDK location.
3. Click OK.
4. Select **File** > **Import**.
5. Select **General** > **Existing Android Code into Workspace**.
6. Click Next.
7. Specify your target directory as the root directory and import the projects listed in Android Project Files.

Android Project Files

Under the forcedroid directory are a couple of important library projects:

- `libs/SalesforceSDK`—Salesforce Mobile SDK project. Provides support for OAuth2 and REST API calls
- `external/cordova`—The Cordova library. This library is needed to build native as well as hybrid apps.

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.

Android Application Structure

Native Android apps that use the Mobile SDK typically require:

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

With Mobile SDK, you:

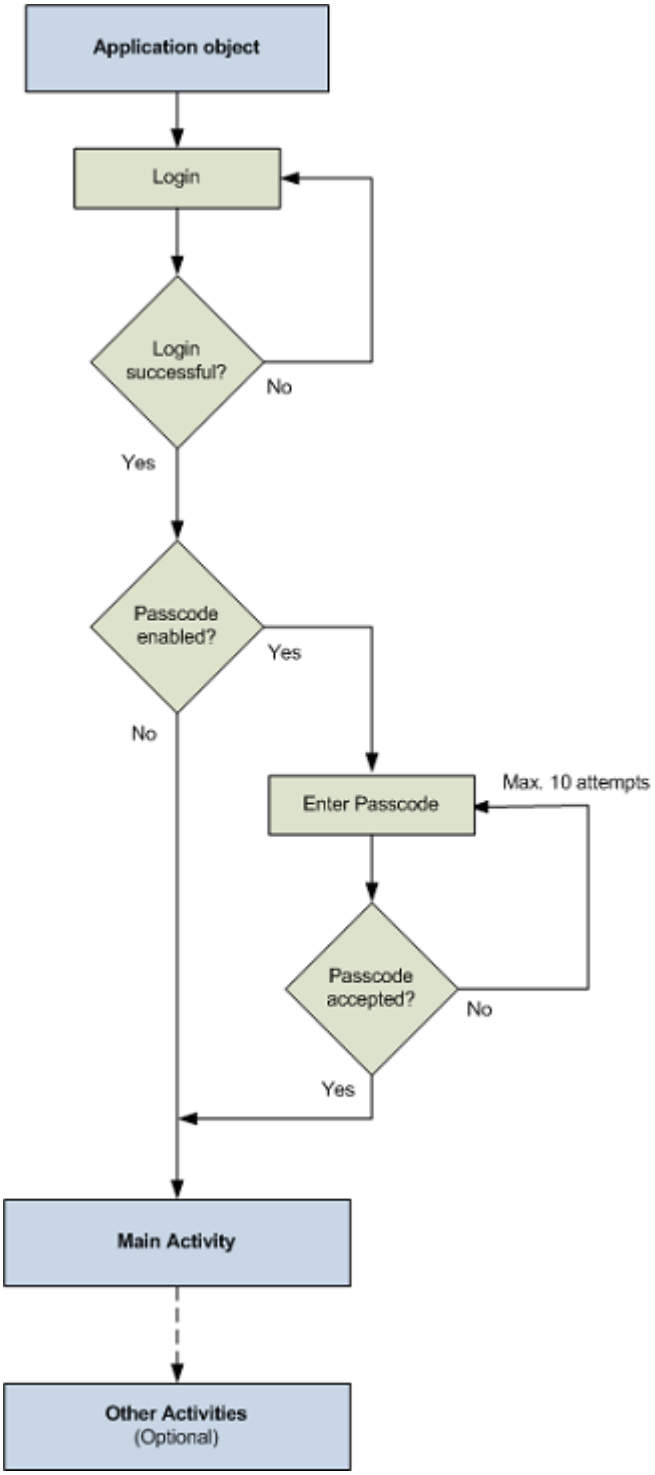
- Create a stub class that extends `android.app.Application`.
- Implement `onCreate()` in your `Application` stub class to call `SalesforceSDKManager.initNative()`.
- Extend `SalesforceActivity`, `SalesforceListActivity`, or `SalesforceExpandableListActivity`. This extension is optional but recommended.

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 Android APIs into Java packages. For a quick overview of these packages and points of interest within them, see Android Packages and Classes on page 336.

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.

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

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 Class

`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.

RestClient Class

As its name implies, the `RestClient` class is an Android app's liaison to the Salesforce REST API.

You don't explicitly create new instances of the `RestClient` class. Instead, you use the `ClientManager` factory class to obtain a `RestClient` instance. Once you get the `RestClient` instance, you can use it to send REST API calls to Salesforce. The method you call depends on whether you're calling from a UI context. See ClientManager Class.

Use the following `RestClient` methods to send REST requests:

- `sendAsync()`—Call this method if you obtained your `RestClient` instance by calling `ClientManager.getRestClient()`.
- `sendSync()`—Call this method if you obtained your `RestClient` instance by calling `ClientManager.peekRestClient()`.

sendSync() Method

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

sendAsync() Method

The `RestClient.sendAsync()` method wraps your `RestRequest` object in a new instance of `WrappedRestRequest`. It then adds the `WrappedRestRequest` object to the request queue and returns that object. If you wish to cancel the request while it's pending, call `cancel()` on the `WrappedRestRequest` object.

getRequestQueue() Method

You can access the underlying `RequestQueue` object by calling `restClient.getRequestQueue()` on your `RestClient` instance. With the `RequestQueue` object you can directly cancel and otherwise manipulate pending requests. For example, you can cancel an entire pending request queue by calling `restClient.getRequestQueue().cancelAll()`. See a code example at Managing the Request Queue.

RestRequest Class

The `RestRequest` class creates and formats REST API requests from the data your app provides. It is implemented by Mobile SDK and serves as a factory for instances of itself.

Don't directly create instances of `RestRequest`. Instead, call an appropriate `RestRequest` static factory method such as `RestRequest.getRequestForCreate()`. To send the request, pass the returned `RestRequest` object to `RestClient.sendAsync()` or `RestClient.sendSync()`. See Using REST APIs.

The `RestRequest` class natively handles the standard Salesforce data operations offered by the Salesforce REST API and SOAP API. 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	Returns the object's complete metadata collection

Operation	Parameters	Description
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
Search	API version, SOQL query string	Executes the specified SOQL search
SearchResultLayout	API version, list of objects	Returns search result layout information for the specified objects
SearchScopeAndOrder	API version	Returns an ordered list of objects in the default global search scope of a logged-in user
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()`
- `getRequestForSearchResultLayout()`
- `getRequestForSearchScopeAndOrder()`
- `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.

FileRequests Class

The `FileRequests` class provides methods that create file operation requests. Each method returns a new `RestRequest` object. Applications send this object to the Salesforce service to process the request. For example, the following code snippet calls the `ownedFilesList()` method to retrieve a `RestRequest` object. It then sends the `RestRequest` object to the server using `RestClient.sendAsync()`:

```
RestRequest ownedFilesRequest = FileRequests.ownedFilesList(null, null);
RestClient client = this.client;
client.sendAsync(ownedFilesRequest, new AsyncRequestCallback() {
    // Do something with the response
});
```

Note: This example passes null to the first parameter (`userId`). This value tells the `ownedFilesList()` method to use the ID of the context, or logged in, user. The second null, for the `pageNum` parameter, tells the method to fetch the first page of results.

See Files and Networking for a full description of `FileRequests` methods.

Methods

For a full reference of `FileRequests` methods, see FileRequests Methods (Android). For a full description of the REST request and response bodies, go to **Chatter REST API Resources** > **Files Resources** at http://www.salesforce.com/us/developer/docs/chatterapi.

Method Name	Description
`ownedFilesList`	Builds a request that fetches a page from the list of files owned by the specified user.
`filesInUsersGroups`	Builds a request that fetches a page from the list of files owned by the user's groups.
`filesSharedWithUser`	Builds a request that fetches a page from the list of files that have been shared with the user.
`fileDetails`	Builds a request that fetches the file details of a particular version of a file.
`batchFileDetails`	Builds a request that fetches the latest file details of one or more files in a single request.
`fileRendition`	Builds a request that fetches the a preview/rendition of a particular page of the file (and version).
`fileContents`	Builds a request that fetches the actual binary file contents of this particular file.
`fileShares`	Builds a request that fetches a page from the list of entities that this file is shared to.

Method Name	Description
`addFileShare`	Builds a request that add a file share for the specified file ID to the specified entity ID.
`deleteFileShare`	Builds a request that deletes the specified file share.
`uploadFile`	Builds a request that uploads a new file to the server. Creates a new file.

`WrappedRestRequest` Class

The `WrappedRestRequest` class subclasses the Volley `Request` class. You don't create `WrappedRestRequest` objects. The `RestClient.sendAsync()` method uses this class to wrap the `RestRequest` object that you passed in and returns it to the caller. You can use this returned object to cancel the request "in flight" by calling the `cancel()` method.

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.
`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.

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 `libs/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-xhdpi`	Resources for extra high-density screens (~320dpi
`drawable-xlarge`	For tablet screens in landscape orientation
`drawable-xlarge-port`	For tablet screens in portrait orientation
`drawable-xxhdpi-port`	Resources for extra-extra high density screens (~480 dpi)
`layout`	Generic versions of layouts
`menus`	Add Connection dialog and login menu for phones

Folder name	Usage
values	Generic styles and values
xml	General app configuration

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

File name	Contains
`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.

SEE ALSO:

Android Resources

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 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.

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.

Unauthenticated REST Requests

In certain cases, some applications must make REST calls before the user becomes authenticated. In other cases, the application must access services outside of Salesforce that don't require Salesforce authentication. To implement such requirements, use a special `RestClient` instance that doesn't require an authentication token.

To obtain an unauthenticated `RestClient` on Android, use one of the following `ClientManager` factory methods:

```
/**
* Method to created an unauthenticated RestClient asynchronously
* @param activityContext
* @param restClientCallback
*/
public void getUnauthenticatedRestClient(Activity activityContext, RestClientCallback
restClientCallback);
/**
* Method to create an unauthenticated RestClient.
* @return
*/
public RestClient peekUnauthenticatedRestClient();
```

Note: A REST request sent through either of these `RestClient` objects requires a full path URL. Mobile SDK doesn't prepend an instance URL to unauthenticated endpoints.

Example:

```
RestClient unauthenticatedRestClient = clientManager.peekUnauthenticatedRestClient();
RestRequest request = new RestRequest(RestMethod.GET,
"https://api.spotify.com/v1/search?q=James%20Brown&type=artist", null);
RestResponse response = unauthenticatedRestClient.sendSync(request);
```

Deferring Login in Native Android Apps

When you create Mobile SDK apps using forcedroid, forcedroid bases your project on a template app that gives you lots of free standard functionality. For example, you don't have to implement authentication—login and passcode handling are built into your launcher activity. This design works well for most apps, and the free code is a big time-saver. However, after you've created your forcedroid app you might find reasons for deferring Salesforce authentication until some point after the launcher activity runs.

You can implement deferred authentication easily while keeping the template app's built-in functionality. Here are the guidelines and caveats:

- Replace the launcher activity (named `MainActivity` in the template app) with an activity that does *not* extend any of the following Mobile SDK activities:
  - `SalesforceActivity`
  - `SalesforceListActivity`
  - `SalesforceExpandableListActivity`

  This rule likewise applies to any other activities that run before you authenticate with Salesforce.
- Do not call the `peekRestClient()` or the `getRestClient()` `ClientManager` method from your launcher activity or from any other pre-authentication activities.
- Do not change the `initNative()` call in the `TemplateApp` class. It must point to the activity class that launches after authentication (`MainActivity` in the template app).
- When you're ready to authenticate with Salesforce, launch the `MainActivity` class.

The following example shows how to place a non-Salesforce activity ahead of Salesforce authentication. You can of course expand and embellish this example with additional pre-authentication activities, observing the preceding guidelines and caveats.

1. Create an XML layout for the pre-authentication landing page of your application. For example, the following layout file, `launcher.xml`, contains only a button that triggers the login flow.

   Note: The following example uses a string resource, `@string/login`, that is defined in the `res/strings.xml` file as follows:

   ```
   <string name="login">Login</string>
   ```

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

   <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:orientation="vertical"
       android:background="@android:color/white"
       android:id="@+id/root">

       <Button android:id="@+id/login_button"
           android:layout_width="80dp"
           android:layout_height="60dp"
           android:text="@string/login"
           android:textColor="@android:color/black"
           android:textStyle="bold"
           android:gravity="center"
           android:layout_gravity="center"
           android:textSize="18sp"
   ```

```
        android:onClick="onLoginClicked" />
</LinearLayout>
```

2. Create a landing screen activity. For example, here's a landing screen activity named `LauncherActivity`. This screen simply inflates the XML layout defined in `launcher.xml`. This class must not extend any of the Salesforce activities or call `peekRestClient()` or `getRestClient()`, since these calls trigger the authentication flow.

```
package com.salesforce.samples.smartsyncexplorer.ui;

import com.salesforce.samples.smartsyncexplorer.R;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class LauncherActivity extends Activity {

 @Override
 public void onCreate(Bundle savedInstance) {
  super.onCreate(savedInstance);
  setContentView(R.layout.launcher);
 }

 /**
  * Callback received when the 'Delete' button is clicked.
  *
  * @param v View that was clicked.
  */
 public void onLoginClicked(View v) {

  /*
   * TODO: Add logic here to determine if we are already logged in,
   * and skip this screen by calling 'finish()', if that is the case.
   */
  final Intent mainIntent = new Intent(this, MainActivity.class);
  mainIntent.addCategory(Intent.CATEGORY_DEFAULT);
  startActivity(mainIntent);
  finish();
 }
}
```

3. Modify the `AndroidManifest.xml` to specify `LauncherActivity` as the activity to be launched when the app first starts.

```
<!-- Launcher screen -->
<activity android:name="com.salesforce.samples.smartsyncexplorer.ui.LauncherActivity"
    android:label="@string/app_name"
    android:theme="@style/SalesforceSDK.ActionBarTheme">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

```
<!-- Main screen -->
<activity android:name="com.salesforce.samples.smartsyncexplorer.ui.MainActivity"
    android:label="@string/app_name"
    android:theme="@style/SalesforceSDK.ActionBarTheme">
    <intent-filter>
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

When you start the application. the `LauncherActivity` screen appears. Click the login button to initiate the Salesforce authentication flow. After authentication completes, the app launches `MainActivity`.

Android Template App: Deep Dive

The TemplateApp sample project implements everything you need to create a basic native 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. By studying this app, you can gain a quick understanding of native apps built with Mobile SDK for Android.

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`. The TemplateApp class accomplishes two main tasks:

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

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);
 }
}
```

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.

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 template app's `MainActivity` class extends the abstract Mobile SDK activity class, `com.salesforce.androidsdk.ui.sfnative.SalesforceActivity`. This superclass 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 `MainActivity` 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.

Using an Anonymous Class in Java

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. The only required component is:

Name	Type	Description
`MainActivity`	Activity	The first activity to be called after login. The name and the class are defined in the project.

Because any app created by the `create_native` script is based on the TemplateApp project, the `MainActivity` component is already included in its manifest. As with any Android app, you can add other components, such as custom activities or services, using the Android Manifest editor in Eclipse.

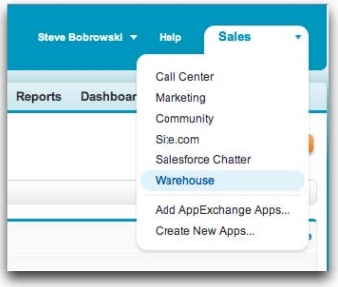
Tutorial: Creating a Native Android Warehouse Application

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

Prerequisites

This tutorial requires the following tools and packages.

- This tutorial uses a Warehouse app that contains a basic inventory database. You'll need to install this app in a DE org. If you install it in an existing DE org, be sure to delete any existing Warehouse components you've made before you install.
  1. Click the installation URL link: http://bit.ly/package100
  2. If you aren't logged in already, enter the username and password of your DE org.
  3. On the Package Installation Details page, click **Continue**.
  4. Click **Next**, and on the Security Level page click **Next**.
  5. Click **Install**.
  6. Click **Deploy Now** and then **Deploy**.
  7. Once the installation completes, you can select the **Warehouse** app from the app picker in the upper right corner.



  8. To create data, click the **Data** tab.
  9. Click the **Create Data** button.
- Install the latest versions of:
  - Java JDK 7 or higher—`http://www.oracle.com/downloads`.
  - Apache Ant 1.8 or later—`http://ant.apache.org`.
  - Minimum Android SDK is 4.2.2 (API level 17). Target Android SDK is 5.0.1 (API 21). The default and target Android SDK version for Mobile SDK hybrid apps is 5.0.1 (API 21). The minimum Android SDK version is 4.2.2 (API level 17) or above.
  - Android SDK Tools, version 24 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—`https://www.eclipse.org`. Check the Android Development Tools website for the minimum supported Eclipse version.

  - In order to run the application in the Emulator, you need to set up at least one Android Virtual Device (AVD) that targets Platform 4.2.2 (API level 17) and 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`.

- Install the Salesforce Mobile SDK using npm:
  1. If you've already successfully installed Node.js and npm, skip to step 4.
  2. Install Node.js on your system. The Node.js installer automatically installs npm.
     i. Download Node.js from www.nodejs.org/download.
     ii. Run the downloaded installer to install Node.js and npm. Accept all prompts asking for permission to install.
  3. At the Terminal window, type *`npm`* and press *`Return`* to make sure your installation was successful. If you don't see a page of usage information, revisit Step 2 to find out what's missing.
  4. At the Terminal window, type *`sudo npm install forcedroid -g`*

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

Create a Native Android App

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

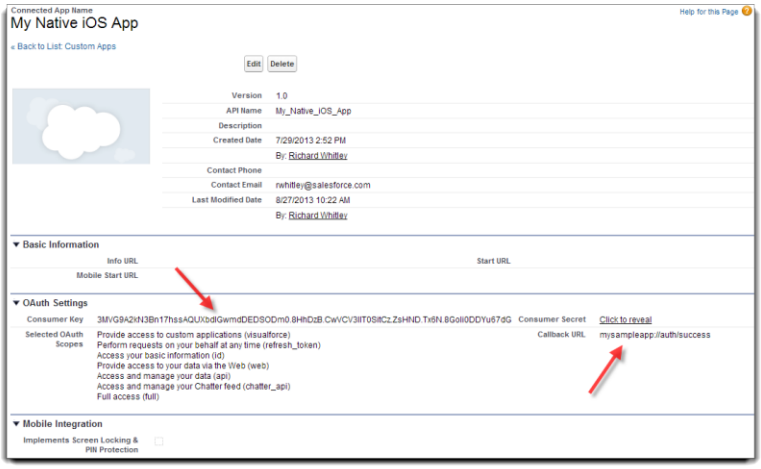
Step 1: Create a Connected App

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

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

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

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



Step 2: Create a Native Android Project

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

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

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

Step 3: Run the New Android App

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

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

Importing and Building Your App in Eclipse

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

1. Launch Eclipse and select `tutorial/AndroidNative` as your workspace directory.
2. Select **Eclipse** > **Preferences**, choose the **Android** section, and enter the Android SDK location.
3. Click OK.
4. Select **File** > **Import** and select **General** > **Existing Projects into Workspace**.
5. Click Next.

6. Specify the `forcedroid/native` directory as your root directory. Next to the list that displays, click **Deselect All**, then browse the list and check the SalesforceSDK project.
7. Click **Finish**.
8. Repeat Steps 4–8. In Step 6, choose `tutorial/AndroidNative` as the root, then select only your new Warehouse 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**.

   Note: If the **Run As** menu doesn't include **Android Application**, you need to configure an Android emulator or device.

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

Step 4: Explore How the Android App Works

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

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

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

MainActivity Class

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

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

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

DetailActivity Class

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

RestClient and RestRequest Classes

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

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

Customize the List Screen

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

Step 1: Remove Existing Controls

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

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

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

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

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

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

Step 2: Update the SOQL Query

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

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

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

with

```
private void fetchDataForList()
```

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

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

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

3. Wrap a `try...catch` block around the call to `RestRequest.getRequestForQuery()`. Replace this:

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

with this:

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

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

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

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

```
        }

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

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

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

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

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

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

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

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

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

7. Save the `MainActivity.java` file.

Step 3:Try Out the App

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

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

Create the Detail Screen

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

Step 1: Create the Detail Screen

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

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

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

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

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

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

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

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

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

    <LinearLayout
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:orientation="horizontal" >

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

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

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

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

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

</LinearLayout>
```

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

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

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

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

```
<!-- Merchandise detail screen -->
<activity android:name="com.samples.warehouse.DetailActivity"
```

```
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen">
</activity>
```

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

Step 2: Create the DetailActivity Class

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

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

   The compiler provides a stub implementation of the required `onResume()` method. Mobile SDK passes an instance of `RestClient` to this method. Since you need this instance to create REST API requests, it's a good idea to cache a reference to it.

6. Add the following declaration to the list of member variables at the top of the new class:

```
private RestClient client;
```

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

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

Step 3: Customize the DetailActivity Class

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

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

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

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

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

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

```
private String merchandiseId;
```

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

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

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

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

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

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

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

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

```
/**
 * Simple class to represent a Merchandise record
 */
static class Merchandise {
 public final String name;
 public final String id;
 public final int quantity;
 public final double price;

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

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

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

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

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

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

    new ArrayList<Merchandise>());
```

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

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

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

6. Change the `onSuccess()` method in `fetchDataForList()` to use the new `Merchandise` object:

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

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

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

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

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

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

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

4. Add a private member for the list view:

```
private ListView listView;
```

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

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

    // Create list adapter
```

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

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

    super.onResume();
}
```

Now that you've designated a listener for list item clicks, you're ready to add the list item click handler.

6. Add the following imports:

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

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

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

}
```

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

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

}
```

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

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

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

10. In the Package Explorer, open **src** > **com.samples.warehouse** > **DetailActivity.java**.
11. In the `onCreate()` method, assign values from the list screen selection to their corresponding data members in the detail activity:

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

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

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

Step 6: Implement the Update Button

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

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

1. Reopen `detail.xml` and add the following `<Button>` node as the last node in the outermost layout.

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

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

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

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

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

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

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

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

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

```
  fields.put("Price__c", priceField.getText().toString());
  saveData(merchandiseId, fields);
 }
```

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

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

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

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

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

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

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

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

Step 7: Try Out the App

1. Build your app and run it in the Android emulator. If you did everything correctly, a detail page appears when you click a Merchandise record in the Warehouse screen.

2. Update a record's quantity and price. Be sure to click the **Update** button in the detail view after you edit the values. When you navigate back to the detail view, the updated values display.

3. Log into your DE org and view the record using the browser UI to see the updated values.

Android Native Sample Applications

Salesforce Mobile SDK includes the following native Android sample applications.

- **RestExplorer** demonstrates the OAuth and REST API functions of the SalesforceSDK. It's also useful for investigating REST API actions from a tablet.
- **NativeSqlAggregator** demonstrates SQL aggregation with SmartSQL. As such, it also demonstrates a native implementation of SmartStore.
- **FileExplorer** demonstrates the Files API as well as the underlying Google Volley networking enhancements.
- **SmartSyncExplorer** demonstrates the power of the native SmartSync library on Android. It resides in the Mobile SDK for Android under `native/SampleApps/SmartSyncExplorer`.

CHAPTER 5 HTML5 and Hybrid 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. HTML5 now supports advanced mobile functionality such as camera and GPS, making it simple to use these popular device features in your Salesforce mobile app.

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

In addition, you can repurpose HTML5 code in a standalone Mobile SDK hybrid app, and then distribute it through an app store. To convert to hybrid, you use the third-party Cordova command line to create a Mobile SDK container project, and then import your HTML5, JavaScript, and CSS files into that project.

Getting Started

If you're already a web developer, you're set up to write HTML5 apps that access Salesforce. HTML5 apps can run in a browser and don't require the Salesforce Mobile SDK. You simply call Salesforce APIs, capture the return values, and plug them into your logic and UI. The same advantages and challenges of running any app in a mobile browser apply. However, Salesforce and its partners provide tools that help streamline mobile web design and coding.

If you want to build your HTML5 app as standalone in a hybrid container and distribute it in the Apple® AppStore® or an Android marketplace, you'll need to create a hybrid app using the Mobile SDK.

Using HTML5 and JavaScript

You don't need a professional development environment such as Xcode or Microsoft® Visual Studio® to write HTML5 and JavaScript code. Most modern browsers include sophisticated developer features including HTML and JavaScript debuggers. You can literally write your application in a text editor and test it in a browser. However, you do need a good knowledge of popular industry libraries that can help to minimize your coding effort.

The recent growth in mobile development has led to an explosion of new web technology toolkits. Often, these JavaScript libraries are open-source and don't require licensing. Most of the tools provided by Salesforce for HTML5 development are built on these third-party technologies.

HTML5 Development Requirements

If you're planning to write a browser-based HTML5 Salesforce application, you don't need Salesforce Mobile SDK.

- 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.

Multi-Device Strategy

With the worldwide proliferation of mobile devices, HTML5 mobile applications must support a variety of platforms, form factors, and device capabilities. Developers who write device-independent mobile apps in Visualforce face these key design questions:

- Which devices and form factors should my app support?
- How does my app detect various types of devices?
- How should I design a Force.com application to best support multiple device types?

Which Devices and Form Factors Should Your App Support?

The answer to this question is dependent on your specific use case and end-user requirements. It is, however, important to spend some time thinking about exactly which devices, platforms, and form factors you do need to support. Where you end up in the spectrum of 'Support all platforms/devices/form factors' to 'Support only desktop and iPhone' (as an example) plays a major role in how you answer the subsequent two questions.

As can be expected, important trade-offs have to be made when making this decision. Supporting multiple form factors obviously increases the reach for your application. But, it comes at the cost of additional complexity both in terms of initially developing the application, and maintaining it over the long-term.

Developing true cross-device applications is not simply a question of making your web page look (and perform) optimally across different form factors and devices (desktop vs phone vs tablet). You really need to rethink and customize the user experience for each specific device/form factor. The phone or tablet version of your application very often does not need all the bells and whistles supported by your existing desktop-optimized Web page (e.g., uploading files or supporting a use case that requires many distinct clicks).

Conversely, the phone/tablet version of your application can support features like geolocation and taking pictures that are not possible in a desktop environment. There are even significant differences between the phone and tablet versions of the better designed applications like LinkedIn and Flipboard (e.g,. horizontal navigation in a tablet version vs single hand vertical scrolling for a phone version). Think of all these consideration and the associated time and cost it will take you to support them when deciding which devices and form factors to support for your application.

Once you've decided which devices to support, you then have to detect which device a particular user is accessing your Web application from.

Client-Side Detection

The client-side detection approach uses JavaScript (or CSS media queries) running on the client browser to determine the device type. Specifically, you can detect the device type in two different ways.

- **Client-Side Device Detection with the User-Agent Header** — This approach uses JavaScript to parse out the User-Agent HTTP header and determine the device type based on this information. You could of course write your own JavaScript to do this. A better option is to reuse an existing JavaScript. A cursory search of the Internet will result in many reusable JavaScript snippets that can detect the device type based on the User-Agent header. The same cursory search, however, will also expose you to some of the perils of using this approach. The list of all possible User-Agents is huge and ever growing and this is generally considered to be a relatively unreliable method of device detection.
- **Client-Side Device Detection with Screen Size and/or Device Features** — A better alternative to sniffing User-Agent strings in JavaScript is to determine the device type based on the device screen size and or features (e.g., touch enabled). One example of this approach can be found in the open-source Contact Viewer HTML5 mobile app that is built entirely in Visualforce. Specifically, the MobileAppTemplate.page includes a simple JavaScript snippet at the top of the page to distinguish between phone and tablet clients based on the screen size of the device. Another option is to use a library like Device.js or Modernizr to detect the device type. These libraries use some combination of CSS media queries and feature detection (e.g., touch enabled) and are therefore a more reliable option for detecting device type. A simple example that uses the Modernizr library to accomplish this can be found at `http://www.html5rocks.com/static/demos/cross-device/feature/index.html`. A more complete example that uses the `Device.js` library and integrates with Visualforce can be found in this GitHub repo: `https://github.com/sbhanot-sfdc/Visualforce-Device.js`. Here is a snippet from the DesktopVersion.page in that repo.

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

<head>
  <!-- Every version of your webapp should include a list of all
  versions. -->
  <link rel="alternate" href="/apex/DesktopVersion" id="desktop" media="only screen and
(touch-enabled: 0)"/>
  <link rel="alternate" href="/apex/PhoneVersion" id="phone" media="only screen and
(max-device-width: 640px)"/>
  <link rel="alternate" href="/apex/TabletVersion" id="tablet" media="only screen and
(min-device-width: 641px)"/>

  <meta name="viewport" content="width=device-width, user-scalable=no"/>
  <script src="{!URLFOR($Resource.Device_js)}"/>
</head>
```

```
<body>
  <ul>
    <li><a href="?device=phone">Phone Version</a></li>
    <li><a href="?device=tablet">Tablet Version</a></li>
  </ul>
  <h1> This is the Desktop Version</h1>
</body>
</apex:page>
```

The snippet above shows how you can simply include a `<link>` tag for each device type that your application supports. The `Device.js` library then automatically redirects users to the appropriate Visualforce page based on device type detected. There is also a way to override the default Device.js redirect by using the '?device=xxx' format shown above.

Server-Side Device Detection

Another option is to detect the device type on the server (i.e., in your Apex controller/extension class). Server-side device detection is based on parsing the User-Agent HTTP header and here is a small code snippet of how you can detect if a Visualforce page is being viewed from an iPhone client.

```
<apex:page docType="html-5.0" sidebar="false" showHeader="false" cache="false"
            standardStylesheets="false" controller="ServerSideDeviceDetection"
            action="{!detectDevice}">
  <h1> This is the Desktop Version</h1>
</apex:page>
```

```
public with sharing class ServerSideDeviceDetection {
    public boolean isIPhone {get;set;}

    public ServerSideDeviceDetection() {
         String userAgent =
                System.currentPageReference().getHeaders().get('User-Agent');
         isIPhone = userAgent.contains('iPhone');
    }

    public PageReference detectDevice(){
        if (isIPhone)
            return Page.PhoneVersion.setRedirect(true);
        else
            return null;
    }
}
```

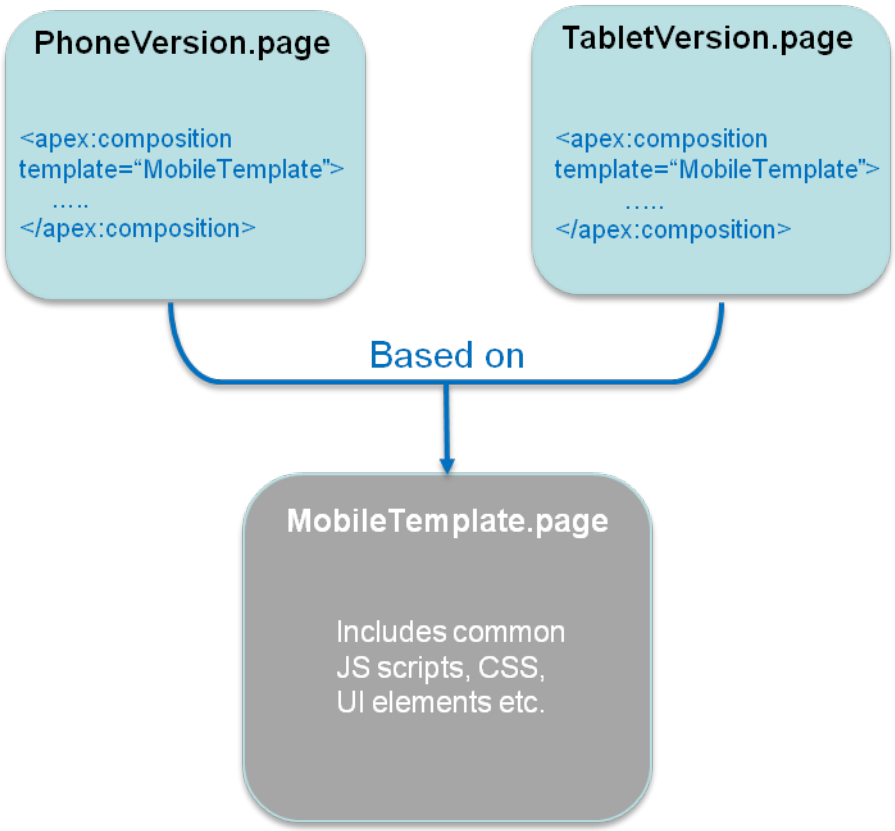
Note that User-Agent parsing in the code snippet above is far from comprehensive and you should implement something more robust that detects all the devices that you need to support based on regular expression matching. A good place to start is to look at the RegEx included in the detectmobilebrowsers.com code snippets.

How Should You Design a Force.com Application to Best Support Multiple Device Types?

Finally, once you know which devices you need to support and how to distinguish between them, what is the optimal application design for delivering a customized user experiences for each device/form factor? Again, a couple of options to consider.

For simple applications where all you need is for the same Visualforce page to display well across different form factors, a responsive design approach is an attractive option. In a nutshell, Responsive design uses CCS3 media queries to dynamically reformat a page to fit the form factor of the client browser. You could even use a responsive design framework like Twitter Bootstrap to achieve this.

Another option is to design multiple Visualforce pages, each optimized for a specific form factor and then redirect users to the appropriate page using one of the strategies described in the previous section. Note that having separate Visualforce pages does not, and should not, imply code/functionality duplication. A well architected solution can maximize code reuse both on the client-side (by using Visualforce strategies like Components, Templates etc.) as well as the server-side (e.g., encapsulating common business logic in an Apex class that gets called by multiple page controllers). An excellent example of such a design can be found in the same open-source Contact Viewer application referenced before. Though the application has separate pages for its phone and tablet version (`ContactsAppMobile.page` and `ContactsApp.page` respectively), they both share a common template (`MobileAppTemplate.page`), thus maximizing code and artifact reuse. The figure below is a conceptual representation of the design for the Contact Viewer application.



Lastly, it is also possible to service multiple form factors from a single Visualforce page by doing server-side device detection and making use of the 'rendered' attribute available in most Visualforce components (or more directly, the CSS 'display:none/block' property on a tag) to selectively show/hide page elements. This approach however can result in bloated and hard-to-maintain code and should be used sparingly.

Supported Browsers

Learn about the browsers we support for the full Salesforce site.

Important: Beginning Summer '15, we'll discontinue support for Microsoft® Internet Explorer® versions 7 and 8. For these versions, this means that some functions may no longer work after this date. Salesforce Customer Support will not investigate issues related to Internet Explorer 7 and 8 after this date.

Accessing the full site in any mobile browser isn't supported. Instead, we recommend using the Salesforce1 app when you're working on a mobile device. To see the mobile browsers that are supported for Salesforce1, check out "Requirements for Using the Salesforce1 App" in the Salesforce Help.

Browser	Comments
Microsoft® Internet Explorer® versions 7, 8, 9, 10, and 11	If you use Internet Explorer, we recommend using the latest version that Salesforce supports. Apply all Microsoft software updates. Note these restrictions. • The full Salesforce site is not supported in Internet Explorer on touch-enabled devices for Windows. Use the Salesforce1 mobile browser app instead. • The Salesforce1 Setup page and the Salesforce1 Wizard require Internet Explorer 9 or later. • The HTML solution editor in Internet Explorer 11 is not supported in Salesforce Knowledge. • The Compatibility View feature in Internet Explorer isn't supported. • The Metro version of Internet Explorer 10 isn't supported. • Internet Explorer 6 and 7 aren't supported for login hints for multiple accounts. • Internet Explorer 7 and 8 aren't supported for the Data Import Wizard. • Internet Explorer 7, 8, and 11 aren't supported for the Developer Console. • Internet Explorer 7 isn't supported for Open CTI. • Internet Explorer 7 and 11 aren't supported for Salesforce CRM Call Center built with CTI Toolkit version 4.0 or higher. • Internet Explorer 7 isn't supported for Force.com Canvas. • Internet Explorer 7 isn't supported for Salesforce console features that require more advanced browser performance and recent Web technologies. The console features not available in Internet Explorer 7 include: – The Most Recent Tabs component – Multiple custom console components on sidebars – Vertical auto-sizing for stacked console components in sidebars – Font and font color for console components' `Button CSS` – Multi-monitor components – The resizable highlights panel – The full-width feed option on feed-based page layouts • Internet Explorer 7 and 8 aren't supported for Community Templates for Self-Service. • Internet Explorer 7 and 8 have performance issues when Multi-Line Edit in Opportunity Splits is used. • Community Templates for Self-Service supports Internet Explorer 9 and above for desktop users and Internet Explorer 11 and above for mobile users. • Internet Explorer 7, 8, and 9 aren't supported for Salesforce Analytics Cloud.

Browser	Comments
	For configuration recommendations, see "Configuring Internet Explorer" in the Salesforce Help.
Mozilla® Firefox®, most recent stable version	Salesforce makes every effort to test and support the most recent version of Firefox. • Mozilla Firefox is supported for desktop users only for Community Templates for Self-Service. For configuration recommendations, see "Configuring Firefox" in the Salesforce Help.
Google Chrome™, most recent stable version	Google Chrome applies updates automatically; Salesforce makes every effort to test and support the most recent version. There are no configuration recommendations for Chrome. Chrome isn't supported for the **Add Google Doc to Salesforce** browser button or the Console tab (the Salesforce console is supported).
Apple® Safari® versions 5.x and 6.x on Mac OS X	There are no configuration recommendations for Safari. Apple Safari on iOS isn't supported for the full Salesforce site. • Safari isn't supported for the Salesforce console. • Safari isn't supported for Salesforce CRM Call Center built with CTI Toolkit versions below 4.0. • Safari isn't supported for Salesforce Analytics Cloud.

Recommendations and Requirements for All Browsers

- For all browsers, you must enable JavaScript, cookies, and TLS 1.0.
- Salesforce recommends a minimum screen resolution of 1024 x 768 for the best possible user experience. Screen resolutions smaller than 1024 x 768 may not display Salesforce features such as Report Builder and Page Layout Editor properly.
- For Mac OS users on Apple Safari or Google Chrome, make sure the system setting `Show scroll bars` is set to **Always**.
- Some third-party Web browser plug-ins and extensions can interfere with the functionality of Chatter. If you experience malfunctions or inconsistent behavior with Chatter, disable all of the Web browser's plug-ins and extensions and try again.

Certain features in Salesforce—as well as some desktop clients, toolkits, and adapters—have their own browser requirements. For example:

- Internet Explorer is the only supported browser for:
  - Standard mail merge
  - Installing Salesforce Classic on a Windows Mobile device
  - Connect Offline
- Firefox is recommended for the enhanced page layout editor.
- Chrome, with machines with 8 GB of RAM, is recommend for the Salesforce console.
- Browser requirements also apply for uploading multiple files on Chatter.

Discontinued or Limited Browser Support

As of Summer '12, Salesforce discontinued support for Microsoft® Internet Explorer® 6. Existing features that have previously worked in this browser may continue to work through 2014. Note these support restrictions.

- Internet Explorer 6 isn't supported for:
  - Answers
  - Chatter
  - Chatter Answers
  - Cloud Scheduler
  - Enhanced dashboard charting options
  - Enhanced profile user interface
  - Forecasts
  - Global search
  - Joined reports
  - Live Agent
  - Quote Template Editor
  - Salesforce console
  - Salesforce Knowledge
  - Schema Builder
  - Site.com
  - Enterprise Territory Management
  - The new user interface theme

Internet Explorer 7 isn't supported for Site.com and Chatter Messenger.

HTML5 Development Tools

For today's Web developers, open source tools are essential for rapid HTML5 development. These tools can make HTML5 development surprisingly simple. Some are built on popular open source JavaScript frameworks, while others are home-grown solutions. For example, you can couple Google's Polymer framework with Force.com JavaScript libraries to create Salesforce-enabled mobile apps surprisingly quickly. Salesforce provides a beta open source library in GitHub—Mobile UI Elements—that does exactly that.

Mobile UI Elements (BETA)

Mobile apps should be small, fun, and engaging. Mobile app developers should spend their time creating innovative functionality, rather than re-creating yet another list view or detail page bound to a set of APIs. With the open-source Mobile UI Elements, HTML and JavaScript developers can build amazing apps with technologies they already know—using a set of pre-built components that are flexible and surprisingly easy to learn.

You can deploy a Mobile UI Elements app several ways.

- In a Visualforce page
- In a remotely hosted page on www.heroku.com or another third-party service
- As a stand-alone app, using the hybrid container provided by Salesforce Mobile SDK

Mobile UI Elements is an open-source, unsupported library based on Google's Polymer framework. It provides fundamental building blocks that you can combine to create fairly complex mobile apps. The component library enables any HTML developer to quickly and easily build mobile applications without having to dig into complex mobile frameworks and design patterns.

You can find the source code for Mobile UI Elements on Github at https://github.com/ForceDotComLabs/mobile-ui-elements.

Third-Party Code

The Mobile UI Elements library makes use of these third-party components:

- Polymer, a JavaScript library for adding new extensions and features to modern HTML5 browsers. It's built on Web Components and is designed to leverage the evolving Web platform on modern browsers.
- jQuery, the JavaScript library that makes it easy to write JavaScript.
- Backbone.js, a JavaScript library providing the model–view–presenter (MVP) application design paradigm.
- Underscore.js, a "utility belt" library for JavaScript.
- Ratchet, prototype iPhone apps with simple HTML, CSS, and JavaScript components.

The following reference sections describe the elements that are currently available.

force_selector_list

The `force-selector-list` element is an extension of `core-selector` element and provides a wrapper around the `force-sobject-collection` element. `force-selector-list` acts as a base for any list UI element that needs selector functionality. It automatically updates the selected attribute when the user taps a row.

Example

```
<force-selector-list sobject="Account" querytype="mru"></force-selector-list>
```

force-sobject

The `force-sobject` element wraps the SmartSync `Force.SObject` in a Polymer element. The `force-sobject` element:

- Provides automatic management of the offline store for caching
- Provides a simpler DOM-based interface to interact with the SmartSync SObject Model
- Allows other Polymer elements to consume SmartSync easily

Example

```
<force-sobject sobject="Account" recordid="001000000000AAA"></force-sobject>
```

force-sobject-collection

The `force-sobject-collection` element is a low-level Polymer wrapper for the SmartSync `Force.SObjectCollection` object. This element:

- Automatically manages the offline data store for caching (when running inside a container)
- Provides a simple DOM-based interface for SmartSync interactions
- Allows other Polymer elements to easily consume SmartSync data

Example

```
<force-sobject-collection sobject="Account" querytype="mru"></force-sobject-collection>
```

force-sobject-layout

The `force-sobject-layout` element provides the layout information for a particular sObject record. It wraps the `describeLayout` API call. The layout information is cached in memory for the existing session and is stored in SmartStore for offline consumption. The `force-sobject-layout` element also provides a base definition for elements that depend on page layouts, such as `force-ui-detail` and `force-sobject-related`.

Example

```
<force-sobject-layout sobject="Account"></force-sobject-layout>
```

force-selector-relatedlist

The `force-selector-relatedlist` element is an extension of the `core-selector` element and fetches the records of related sObjects using a `force-sobject-collection` element. `force-selector-relatedlist` is a base element for UI elementx that render a record's related list and also require selector functionality.

Example

```
<force-selector-relatedlist related="{{related}}"></force-selector-relatedlist>
```

force-sobject-relatedlists

The `force-sobject-relatedlists` element enables the rendering of related lists of a sObject record. It embeds the `force-sobject-layout` element to fetch the related lists configuraton from the page layout settings. It parses the related lists configuration for a particular sObject type. If the `recordid` attribute is provided, it also generates a SOQL/cache query to fetch the related record items.

Example

```
<force-sobject-relatedlists sobject="Account"
recordid="001000000000AAA"></force-sobject-relatedlists>
```

force-sobject-store

The `force-sobject-store` element wraps the SmartSync `Force.StoreCache` in a Polymer element. This element:

- Automatically manages the lifecycle of the SmartStore soup for each sObject type
- Automatically creates index specs based on the lookup relationships on the sObject
- Provides a simpler DOM-based interface to interact with the SmartSync SObject model
- Allows other Polymer elements to easily consume SmartStore data

.

Example

```
<force-sobject-store sobject="Account"></force-sobject-store>
```

force-ui-app

The `force-ui-app` element is a top-level UI element that provides the basic styling and structure for the application. This element uses Polymer layout features to enable flexible sections on the page. This is useful in a single-page view with split view panels. All children of the main section must specify the "content" class to apply the correct styles.

Example

When used in a Visualforce context:

```
<force-ui-app multipage="true"></force-ui-app>
```

force-ui-detail

The `force-ui-detail` element enables the rendering of a full view of a Salesforce record. This element uses the `force-sobject-layout` element to fetch the page layout for the record. This element also embeds a `force-sobject` element to allow all the CRUD operations on an sObject. To inherit the default styles, this element should always be a child of `force-ui-app`.

Example

```
<force-ui-detail sobject="Account" recordid="001000000000AAA"></force-ui-detail>
```

force-ui-list

The `force-ui-list` element enables the rendering of the list of records for any sObject. Using attributes, you can configure this element to show specific set of records. To inherit the appropriate styles, this element should always be a child of `force-ui-app`.

Example

```
<force-ui-list sobject="Account" querytype="mru"></force-ui-list>
```

force-ui-relatedlist

The `force-ui-relatedlist` element extends `force-selector-relatedlistelement` and renders a list of related records to an `sobject` record. To inherit the default styles, this element should always be a child of `force-ui-app`.

Example

```
<force-ui-relatedlist related="{{related}}"></force-ui-relatedlist>
```

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.mobilesdk.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 flexibility 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 http://www.salesforce.com/us/developer/docs/apexcode/Content/apex_classes_annotation_RemoteAction.htm to learn more about `@RemoteAction` annotations.

Accessing the Salesforce API with ForceTK and jQuery

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 library. To add these resources:

1. Create an archive file, such as a ZIP file, that contains `app.js`, `forcetk.mobilesdk.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.mobilesdk.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.

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 offline.

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

Hybrid Apps Quick Start

Hybrid apps give you the ease of JavaScript and HTML5 development while leveraging Salesforce Mobile SDK

If you're comfortable with the concept of hybrid app development, use the following steps to get going quickly.

1. To develop apps for Android, you need:
   - Java JDK 7 or higher—`http://www.oracle.com/downloads`.
   - Apache Ant 1.8 or later—`http://ant.apache.org`.
   - Minimum Android SDK is 4.2.2 (API level 17). Target Android SDK is 5.0.1 (API 21). The default and target Android SDK version for Mobile SDK hybrid apps is 5.0.1 (API 21). The minimum Android SDK version is 4.2.2 (API level 17) or above.
   - Android SDK Tools, version 24 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—`https://www.eclipse.org`. Check the Android Development Tools website for the minimum supported Eclipse version.
   - In order to run the application in the Emulator, you need to set up at least one Android Virtual Device (AVD) that targets Platform 4.2.2 (API level 17) and 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`.
2. To develop apps for iOS, you need:
   - Xcode—Version 6.0 is the minimum, but we recommend the latest version.
   - iOS 7.0 or higher.
   - A Salesforce Developer Edition organization with a connected app.
3. Install the Mobile SDK.
   - Android Installation
   - iOS Installation
4. If you don't already have a connected app, Create a Connected App. For OAuth scopes, select `api`, `web`, and `refresh_token`.

   Note: When specifying the Callback URL, there's no need to use a real address. Use any value that looks like a URL, such as `myapp:///mobilesdk/oauth/done`.
5. Create a hybrid app.
   - Follow the steps at Create Hybrid Apps on page 123. Use *`hybrid_local`* for the application type.
6. Run your new app.
   - Build and Run Your Hybrid App On Android on page 125
   - Run Your Hybrid App On iOS on page 125

.

Creating Hybrid Apps

Hybrid apps combine the ease of HTML5 Web app development with the power and features of the native platform. They run within a Salesforce mobile container—a native layer that translates the app into device-specific code—and define their functionality in HTML5 and JavaScript files. These apps fall into one of two categories:

- **Hybrid local**—Hybrid apps developed with the `forcetk.mobilesdk.js` library wrap a Web app inside the mobile container. These apps store their HTML, JavaScript, and CSS files on the device.
- **Hybrid remote** — Hybrid apps developed with Visualforce technology deliver Apex pages through the mobile container. These apps store some or all of their HTML, JavaScript, and CSS files either on the Salesforce server or on the device (at `http://localhost`).

In addition to providing HTML and JavaScript code, you also must maintain a minimal container app for your target platform. These apps are little more than native templates that you configure as necessary.

If you're creating libraries or sample apps for use by other developers, we recommend posting your public modules in a version-controlled online repository such as GitHub (https://github.com). For smaller examples such as snippets, GitHub provides *gist*, a low-overhead code sharing forum (https://gist.github.com).

About Hybrid Development

Developing hybrid apps with the Mobile SDK container requires you to recompile and rebuild after you make changes. JavaScript development in a browser is easier. After you've altered the code, you merely refresh the browser to see your changes. For this reason, we recommend you develop your hybrid app directly in a browser, and only run your code in the container in the final stages of testing.

We recommend developing in a browser such as Google Chrome that comes bundled with developer tools. These tools let you access the symbols and code of your web application during runtime.

Building Hybrid Apps With Cordova

Salesforce Mobile SDK 3.2 upgrades its hybrid container to use Apache Cordova 3.6.x (3.6.3 for iOS, 3.6.4 for Android). Architecturally, Mobile SDK hybrid apps are Cordova apps that use the Salesforce Mobile SDK as a Cordova plugin. By leveraging the enhancements in Cordova 3, this architecture simplifies the process of upgrading projects to a new version of Cordova. Cordova also provides a simple command line tool for updating apps. To read more about Cordova 3 benefits, see this blog post.

Create Hybrid Apps

To develop hybrid apps, you must meet the following prerequisites:

- Proficiency in HTML5 and JavaScript development.
- An installed development environment for the intended platform, for building and maintaining the hybrid container app. For Android, see Native Android Requirements. For iOS, see Native iOS Requirements.
- For hybrid remote apps, make sure that you meet the requirements for HTML5 and Hybrid Development.
- For hybrid remote apps, you must have an Apex landing page to point to when you create your app.

To create Mobile SDK hybrid apps, use the forceios or forcedroid utility and the Cordova command line.

1. Open a command prompt or terminal window.
2. Install the Cordova command line:

```
npm -g install cordova
```

3. Follow the instructions for your target platform.

For Android:	For iOS:
a. Install the forcedroid npm package. If you previously installed an earlier version of forcedroid, you must reinstall. b. After installing Mobile SDK for Android, create a new hybrid app as described in Creating an Android Project. When you're prompted for the application type: • Specify `hybrid_local` for a hybrid app that stores its code in the local project. • Specify `hybrid_remote` for a hybrid app with code in a Visualforce app on the server. When forcedroid asks for the start page, specify the relative URL of your Apex landing page.	a. Install the forceios npm package. If you previously installed an earlier version of forceios, you must reinstall. b. After installing Mobile SDK for iOS, create a new hybrid app as described in Creating an iOS Project. When you're prompted for the application type: • Specify `hybrid_local` for a hybrid app that store its code in the local project. • Specify `hybrid_remote` for a hybrid app with code in a Visualforce app on the server. When forceios asks for the start page, specify the relative URL of your Apex landing page.

4. Put your HTML, JavaScript, and CSS files and your `bootconfig.json` file in the `${target.dir}/www/` directory of the project directory.

   Important: Do not include `cordova.js`, `cordova.force.js` or any Cordova plugins.

5. Use the *cd* command to change to the project directory.
6. For each Cordova plugin you need, type:

   ```
   cordova plugin add <plugin repo or plugin name>
   ```

   Note: Go to https://plugins.cordova.io to search for available plugins.

7. (Optional) To add iOS support to a new Android hybrid app, type:

   ```
   cordova platform add ios
   ```

   This step creates a `platforms/ios` directory in your app directory and then creates an Xcode project in the `ios` directory. The Xcode project includes the plugins you've added to your app.

8. (Optional) To add Android support to a new iOS hybrid app, type:

   ```
   cordova platform add android
   ```

   This step creates a `platforms/android` directory in your app directory and then creates an Eclipse project in the `android` directory. The Eclipse project includes the plugins you've added to your app.

9. Type:

   ```
   cordova prepare
   ```

   to deploy your web assets to their respective platform-specific directories under the `www/` directory.

   Important: During development, always run `cordova prepare` after you've changed the contents of the www/ directory, to deploy your changes to the platform-specific project folders.

See "The Command-Line Interface" in the Cordova 3.5 documentation for more information on the Cordova command line.

Build and Run Your Hybrid App On Android

Before building, be sure that you've installed the Android SDK and have configured some device emulators in AVD.

After you've run `cordova prepare`, you can build the project two ways.

- Open the project in Eclipse and configure the workspace to run the app in an emulator. If you're running node.js version 0.11 or older, you must use this build option.

  OR

- **Only if you are running node.js version 0.12 or higher:** Continue using the Cordova command line by running `cordova compile [ios | android]` or `cordova build [ios | android]`. The `cordova build` command is slightly redundant because it's shorthand for the following:

  ```
  cordova prepare
  cordova compile
  ```

To run the app from the command line, type:

```
cordova emulate android
```

To run the app in Eclipse:

1. Start Eclipse.
2. Select your new app directory as the root of your workspace.
3. Select **File** > **Import**.
4. Expand the Android directory and choose **Existing Android Code into Workspace**.
5. Click **Next**.
6. Choose your new app directory as the root directory.
7. Click **Deselect All**, then select the following projects:
   - `platforms/android`
   - `platforms/android/CordovaLib`
   - `plugins/com.salesforce/android/libs/SalesforceSDK`
   - `plugins/com.salesforce/android/libs/SmartStore` (if you plan to use SmartStore)
8. Once everything is built, right-click your new app project and choose **Run** > **As Android Application**.

Run Your Hybrid App On iOS

After you've run `cordova prepare` on an iOS hybrid app, you can either open the project in Xcode to run the app in an iOS simulator, or you can continue using the Cordova command line. In both cases, be sure that you've installed Xcode.

To run the iOS simulator from the command line, type:

```
cordova emulate ios
```

To run the app in Xcode:

1. In Xcode, select **File** > **Open**.
2. Navigate to the `platforms/ios/` directory in your new app's directory.
3. Double-click the <*app name*>.xcodeproj file.

4. Click the Run button in the upper left corner, or press *COMMAND-R*.

Developing Hybrid Remote Apps

For hybrid remote applications, you no longer need to host `cordova.js` or any plugins on the server. Instead, you can include `cordova.js` as `https://localhost/cordova.js` in your HTML source. For example:

```
<script src="https://localhost/cordova.js"></script>
```

You can also use `https://localhost` for all of your CSS and JavaScript resources and then bundle those files with the app, rather than delivering them from the server. This approach gives your hybrid remote apps a performance boost while letting you develop with Visualforce and Apex.

Note:

- Mobile SDK 2.3 and later automatically whitelists `https://localhost` in hybrid remote apps. If your app was developed in an earlier version of Mobile SDK, you can manually whitelist `https://localhost` in your `config.xml` file.
- A Visualforce page that uses `https://localhost` to include source files works only in the Salesforce Mobile SDK container application. To make the page run in a web browser as well, use Apex to examine the user agent and detect whether the client is a Mobile SDK container. Based on your findings, use the appropriate script include tags.

Example: You can easily convert the FileExplorer SDK sample, which is a hybrid local app, into a hybrid remote app. To convert the app, you redefine the main HTML page as an Apex page that will be delivered from the server. You can then bundle the CSS and JavaScript resources with the app so that they're stored on the device.

1. Update the `bootconfig.json` file to redefine the app as hybrid remote. Change:

   ```
   "isLocal": true,
   "startPage": "FileExplorer.html",
   ```

   to:

   ```
   "isLocal": false,
   "startPage": "apex/FileExplorer",
   ```

2. In your Visualforce-enabled Salesforce organization, create a new Apex page named "FileExplorer" that replicates the `FileExplorer.html` file.

   ```
   <apex:page showHeader="false" sidebar="false">
   <!-- Paste content of FileExplorer.html here -->
   </apex:page>
   ```

3. Update all references to CSS files so that they will be loaded from `https://localhost`. Replace:

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

   with:

   ```
   <link rel="stylesheet" href="https://localhost/css/styles.css"/>
   <link rel="stylesheet" href="https://localhost/css/ratchet.css"/>
   ```

4. Repeat step 3 for all script references. Replace:

   ```
   <!-- Container -->
   <script src="js/jquery.min.js"></script>
   ```

```
<script src="js/underscore-min.js"></script>
<script src="js/backbone-min.js"></script>
<script src="cordova.js"></script>
<script src="js/forcetk.mobilesdk.js"></script>
<script src="js/smartsync.js"></script>
<script src="js/fastclick.js"></script>
<script src="js/stackrouter.js"></script>
<script src="js/auth.js"></script>
<!-- End Container -->
```

with:

```
<!-- Container -->
<script src="https://localhost/js/jquery.min.js"></script>
<script src="https://localhost/js/underscore-min.js"></script>
<script src="https://localhost/js/backbone-min.js"></script>
<script src="https://localhost/cordova.js"></script>
<script src="https://localhost/js/forcetk.mobilesdk.js"></script>
<script src="https://localhost/js/smartsync.js"></script>
<script src="https://localhost/js/fastclick.js"></script>
<script src="https://localhost/js/stackrouter.js"></script>
<script src="https://localhost/js/auth.js"></script>
<!-- End Container -->
```

When you test this sample, be sure to log into the organization where you created the Apex page.

Hybrid Sample Apps

Since Salesforce Mobile SDK 2.3 adopted Cordova as its hybrid project generator, the means of accessing and building hybrid sample apps has likewise undergone a transformation. Here's a summary of the changes:

- We've discontinued the `samples` targets of the forcedroid and forceios utilities.
- You can access the iOS samples through the Mobile SDK workspace (`SalesforceMobileSDK.xcodeproj`) in the root directory of the SalesforceMobileSDK-iOS GitHub repository. Also, you can access the Android samples from the `hybrid/SampleApps` directory of a cloned SalesforceMobileSDK-Android repository.
- If you prefer, you can download just the shared source code for the hybrid samples from the SalesforceMobileSDK-Shared GitHub repo and build the samples with the Cordova command line.

Salesforce Mobile SDK provides the following hybrid sample apps.

- **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.
- **NoteSync**: Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.
- **HybridFileExplorer**: Demonstrates the Files API.
- **SimpleSync:**: Demonstrates how to use the SmartSync plugin.
- **SmartStoreExplorer**: Lets you explore SmartStore APIs.

- **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.

Build Hybrid Sample Apps

You can build hybrid sample apps using the forcedroid or forceios tools.The web assets—HTML, JavaScript, and CSS files—and the `bootconfig.json` file for sample hybrid applications are available in the SalesforceMobileSDK-Shared GitHub repository.

Note: The ContactExplorer sample requires the `org.apache.cordova.contacts` and `org.apache.cordova.statusbar` plugins.

The other hybrid sample apps do not require special Cordova plugins.

To build one of the sample apps:

1. Open a command prompt or terminal window.
2. Clone the shared repo:

```
git clone https://github.com/forcedotcom/SalesforceMobileSDK-Shared
```

3. Using forcedroid or forceios, create a new app. For `type`, enter "hybrid_local".
4. Change to your new app directory:

```
cd <app_target_directory>
```

5. If you're building the ContactExplorer sample app, add the required Cordova plugins:

```
cordova plugin add org.apache.cordova.contacts
cordova plugin add org.apache.cordova.statusbar
```

6. To add Android support to a forceios project:

```
cordova platform add android
```

7. (Mac only) To add iOS support to a forcedroid project:

```
cordova platform add ios
```

8. Copy the sample source files to the `www` folder of your new project directory.

   On Mac:

```
cp -RL <local path to SalesforceMobileSDK-Shared>/SampleApps/<template>/* www/
```

   On Windows:

```
copy <local path to SalesforceMobileSDK-Shared>\SampleApps\<template>\*.* www
```

   If you're asked, affirm that you want to overwrite existing files.

9. Do the final Cordova preparation:

```
cordova prepare
```

Running the ContactExplorer Hybrid Sample

Let's take a look at the ContactExplorer, one of the hybrid local sample apps.

Note: Be sure that you've installed Apache Ant and added it to your system path before running the following Cordova scripts.

Source code for the sample apps lives on GitHub, so start by cloning the shared repository.

1. Open a command prompt or terminal window.
2. Clone the shared repo: `git clone https://github.com/forcedotcom/SalesforceMobileSDK-Shared`
3. After the cloning finishes, define an `$appname` environment variable on Mac OS X, or `%appname%` on Windows. Use any name you want, such as "contactsApp":

   ```
   export appname=contactsApp
   ```

   or, on Windows:

   ```
   set appname=contactsApp
   ```

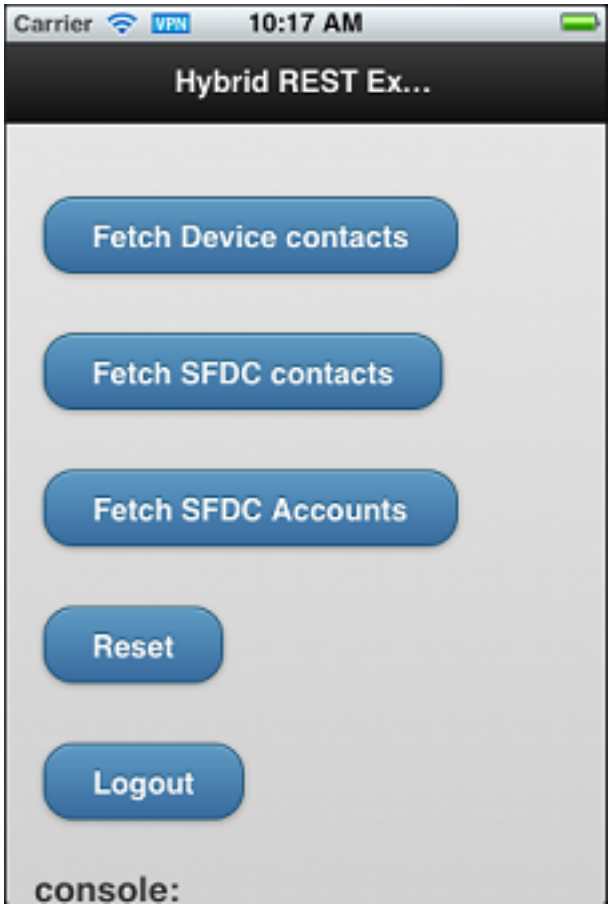
4. Run the following script. Though this script is Mac-compatible, you can easily run it on Windows by substituting `%appname%` for `$appname`, and using the Windows `copy` command instead of `cp -RL`. Also, remove the `cordova platform add ios` command, which isn't Windows-compatible.

   ```
   cordova create $appname com.salesforce.contactexplorer $appname
   cd $appname
   cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
   cordova plugin add org.apache.cordova.contacts
   cordova plugin add org.apache.cordova.statusbar
   cordova platform add android
   cordova platform add ios
   cp -RL <local path to SalesforceMobileSDK-Shared>/samples/$template/* www/
   cordova prepare
   ```

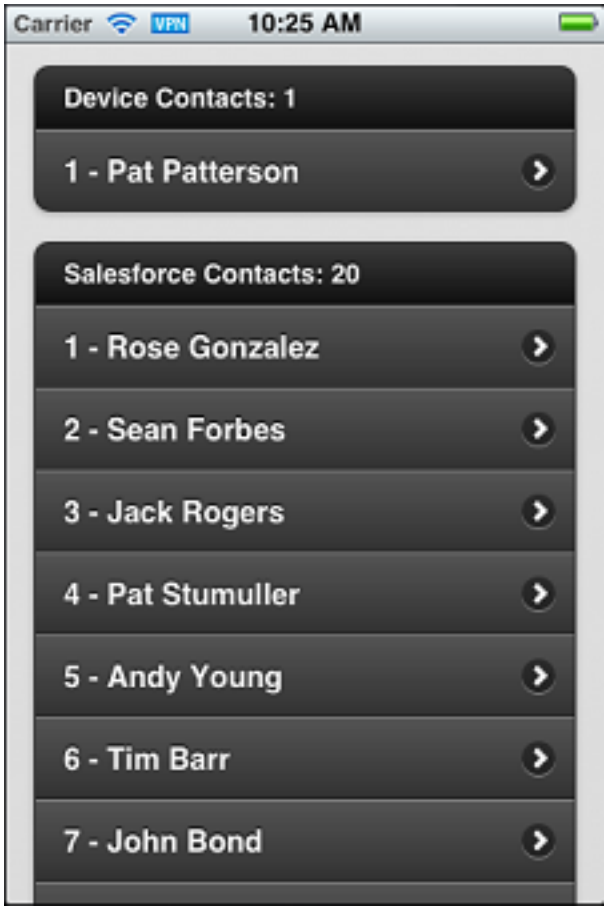
The script creates an iOS project and an Android project, both of which wrap the ContactsExplorer sample app. Now we're ready to run the app on one of these platforms. If you're using an iOS device, you must configure a profile as described in the Xcode User Guide at developer.apple.com/library. Similarly, Android devices must be set up as described at developer.android.com/tools. If you need more help getting the app to run, see Build and Run Your Hybrid App On Android on page 125 or Run Your Hybrid App On iOS on page 125.

When you run the app, after an initial splash screen, you should see the Salesforce login screen.

Log in with your DE username and password. When you're prompted to allow your app access to your data in Salesforce, tap **Allow** . You should now be able to retrieve lists of contacts and accounts from your DE account.



Tap to retrieve Contact and Account records from your DE account. Scroll down to see the list.



How the Sample App Works

Notice the app can also retrieve contacts from the device - something that an equivalent web app would be unable to do. Let's take a closer look at how the app can do this.

After completing the login process, the sample app displays `index.html` (located in the www folder). When the page has completed loading and the mobile framework is ready, the `onDeviceReady()` function calls `regLinkClickHandlers()` (in `inline.js`). `regLinkClickHandlers()` sets up five click handlers for the various functions in the sample app.

```
$j('#link_fetch_device_contacts').click(function() {
    logToConsole("link_fetch_device_contacts clicked");
    var contactOptionsType =
cordova.require("org.apache.cordova.contacts.ContactFindOptions");
    var options = new contactOptionsType();
    options.filter = ""; // empty search string returns all contacts
    options.multiple = true;
    var fields = ["name"];
    var contactsObj = cordova.require("org.apache.cordova.contacts.contacts");
    contactsObj.find(fields, onSuccessDevice, onErrorDevice, options);
    });
});
```

This handler calls `find()` on the `org.apache.cordova.contacts.contacts` object to retrieve the contact list from the device. The `onSuccessDevice()` function (not shown here) renders the contact list into the `index.html` page.

```
$j('#link_fetch_sfdc_contacts').click(function() {
    logToConsole("link_fetch_sfdc_contacts clicked");
    forcetkClient.query("SELECT Name FROM Contact",
        onSuccessSfdcContacts, onErrorSfdc);
});
```

The `#link_fetch_sfdc_contacts` handler runs a query using the `forcetkClient` object. This object is set up during the initial OAuth 2.0 interaction, and gives access to the Force.com REST API in the context of the authenticated user. Here we retrieve the names of all the contacts in the DE account, and `onSuccessSfdcContacts()` then renders them as a list on the `index.html` page.

```
$j('#link_fetch_sfdc_accounts').click(function() {
    logToConsole("link_fetch_sfdc_accounts clicked");
    forcetkClient.query("SELECT Name FROM Account",
        onSuccessSfdcAccounts, onErrorSfdc);
});
```

The `#link_fetch_sfdc_accounts` handler is very similar to the previous one, fetching Account records via the Force.com REST API. The remaining handlers, `#link_reset` and`#link_logout`, clear the displayed lists and log out the user respectively.

Create a Mobile Page to List Information

The ContactExplorer sample hybrid app is useful in many respects, and serves as a good starting point to learn hybrid mobile app development. You can have more fun with it by modifying it to display merchandise records from a custom Salesforce schema named Warehouse.

You can build the Warehouse schema quickly using the getting started content online: http://wiki.developerforce.com/page/Developing_Cloud_Apps_—_Coding_Optional. If you're familiar with Salesforce packages, you can just install it as a package in your Developer Edition org: http://goo.gl/1FYg90.

Note:

- If you're modifying a Cordova iOS project in Xcode, you may need to copy your code to the `Staging/www/` project folder to test your changes. If you use only the Cordova command line instead of Xcode to build Cordova iOS apps, you should modify only the *<projectname>*`/www/` folder.

Modify the App's Initialization Block (index.html)

In this section, you modify the view file (`index.html`) and the controller (`inline.js`) to make the app specific to the Warehouse schema and display all records in the Merchandise custom object.

In your app, you want a list of Merchandise records to appear on the default Home page of the mobile app. Consequently, the first thing to do is to modify what happens automatically when the app calls the `onDeviceReady` function. Comment out two calls to `regLinkClickHanders()`--one in the `onDeviceReady()` function, and the other in the `salesforceSessionRefreshed()` function. Then, add the following code to the tail end of the `salesforceSessionRefreshed()` function in `index.html`.

```
// log message
logToConsole("Calling out for records");
// register click event handlers -- see inline.js
// regLinkClickHandlers();
```

```
// retrieve Merchandise records, including the Id for links
forcetkClient.query("SELECT Id, Name, Price__c, Quantity__c
    FROM Merchandise__c", onSuccessSfdcMerchandise, onErrorSfdc);
```

Notice that this JavaScript code leverages the ForceTK library to query the Force.com database with a basic SOQL statement and retrieve records from the Merchandise custom object. On success, the function calls the JavaScript function `onSuccessSfdcMerchandise` (which you build in a moment).

Create the App's mainpage View (index.html)

To display the Merchandise records in a standard mobile, touch-oriented user interface, scroll down in `index.html` and replace the entire contents of the `<body>` tag with the following HTML.

```
<!-- Main page, to display list of Merchandise once app starts -->
<div data-role="page" data-theme="b" id="mainpage">
   <!-- page header -->
   <div  data-role="header">
      <!-- button for logging out -->
      <a href='#' id="link_logout" data-role="button"
         data-icon='delete'>
            Log Out
      </a>
      <!-- page title -->
      <h2>List</h2>
   </div>
   <!-- page content -->
   <div id="#content" data-role="content">
      <!-- page title -->
      <h2>Mobile Inventory</h2>
      <!-- list of merchandise, links to detail pages -->
      <div id="div_merchandise_list">
      <!-- built dynamically by function onSuccessSfdcMerchandise -->
      </div>
    </div>
</div>
```

Overall, notice that the updated view uses standard HTML tags and jQuery Mobile markup (e.g., data-role, data-theme, data-icon) to format an attractive touch interface for your app. Developing hybrid-based mobile apps is straightforward if you already know some basic standard Web development technology, such as HTML, CSS, JavaScript, and jQuery.

Modify the App's Controller (inline.js)

In the previous section, the initialization block in the view defers to the `onSuccessSfdcMerchandise` function of the controller to dynamically generate the HTML that renders Merchandise list items in the encompassing div, `div_merchandise_list`. In this step, you build the `onSuccessSfdcMerchandise` function.

Open the `inline.js` file and add the following controller action, which is somewhat similar to the sample functions.

Important: Be careful if you cut and paste this or any code from a binary file! It's best to purify it first by pasting it into a plain text editor and then copying it from there. Also, remove any line breaks that occur in the middle of code statements.

```
// handle successful retrieval of Merchandise records
function onSuccessSfdcMerchandise(response) {
```

```
// avoid jQuery conflicts
var $j = jQuery.noConflict();
var logToConsole =
  cordova.require("com.salesforce.util.logger".logToConsole;
// debug info to console
logToConsole("onSuccessSfdcMerchandise: received " +
   response.totalSize + " merchandise records");

// clear div_merchandise_list HTML
$j("#div_merchandise_list").html("");

// set the ul string var to a new UL
var ul = $j('<ul data-role="listview" data-inset="true"
   data-theme="a" data-dividertheme="a"></ul>');

// update div_merchandise_list with the UL
$j("#div_merchandise_list").append(ul);

// set the first li to display the number of records found
// formatted using list-divider
ul.append($j('<li data-role="list-divider">Merchandise records: '
   + response.totalSize + '</li>'));

// add an li for the merchandise being passed into the function
// create array to store record information for click listener
inventory = new Array();
// loop through each record, using vars i and merchandise
$j.each(response.records, function(i, merchandise) {
    // create an array element for each merchandise record
    inventory[merchandise.Id] = merchandise;
    // create a new li with the record's Name
    var newLi = $j("<li class='detailLink' data-id='" + merchandise.Id
        + "'><a href='#'>" + merchandise.Name + "</a></li>");
    ul.append(newLi);
});

// render (create) the list of Merchandise records
$j("#div_merchandise_list").trigger( "create" );
// send the rendered HTML to the log for debugging
logToConsole($j("#div_merchandise_list").html());

// set up listeners for detailLink clicks
$j(".detailLink").click(function() {
    // get the unique data-id of the record just clicked
    var id = $j(this).attr('data-id');
    //  using the id, get the record from the array created above
    var record = inventory[id];

    // use this info to set up various detail page information
    $j("#name").html(record.Name);
    $j("#quantity").val(record.Quantity__c);
    $j("#price").val(record.Price__c);
    $j("#detailpage").attr("data-id",record.Id);
```

```
        // change the view to the detailpage
        $j.mobile.changePage('#detailpage', {changeHash: true});

    });

}
```

The comments in the code explain each line. Notice the call to `logToConsole();` the JavaScript outputs rendered HTML to the console log so that you can see what the code creates. Here's an excerpt of some sample output.

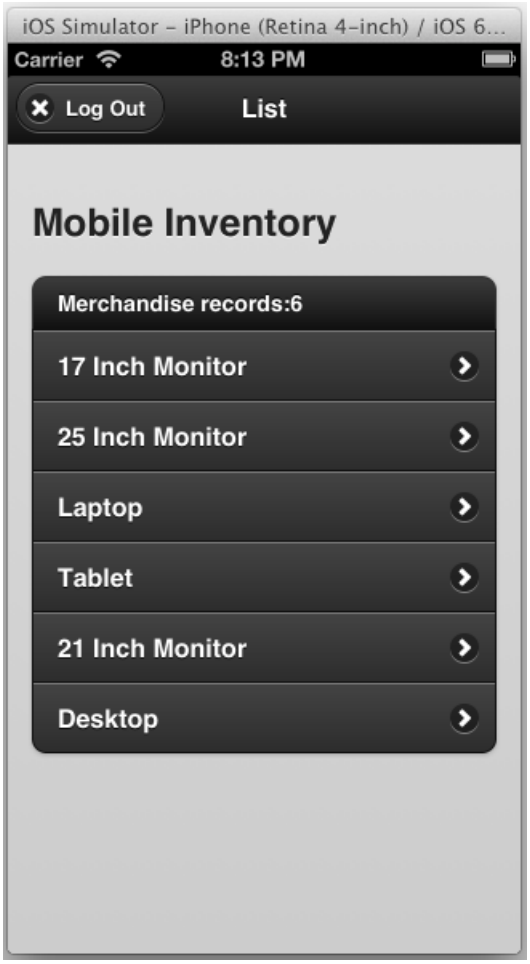
```
<ul data-role="listview" data-inset="true" data-theme="a"
    data-dividertheme="a" class="ui-listview ui-listview-inset
    ui-corner-all ui-shadow">
  <li data-role="list-divider" role="heading"
    class="ui-li ui-li-divider ui-btn ui-bar-a ui-corner-top">Merchandise records: 6
  </li>
  <li class="detailLink ui-btn ui-btn-up-a ui-btn-icon-right ui-li"
     data-id="a00E0000003BzSfIAK" data-theme="a">
    <div class="ui-btn-inner ui-li">
      <div class="ui-btn-text">
        <a href="#" class="ui-link-inherit">Tablet</a>
      </div>
    </div>
  </li>
  <li class="detailLink ui-btn ui-btn-up-a ui-btn-icon-right ui-li"
      data-id="a00E0000003BuUpIAK" data-theme="a">
    <div class="ui-btn-inner ui-li">
      <div class="ui-btn-text">
        <a href="#" class="ui-link-inherit">Laptop</a>
      </div>
    </div>
  </li>

...

</ul>
```

In particular, notice how the code:

- creates a list of Merchandise records for display on the app's primary page
- creates each list item to display the Name of the Merchandise record
- creates each list item with unique link information that determines what the target detail page displays

Test the New App

Restart the simulator for your mobile app. When you do, the initial page should look similar to the following screen.



If you click any particular Merchandise record, nothing happens yet. The list functionality is useful, but even better when paired with the detail view. The next section helps you build the *detailpage* that displays when a user clicks a specific Merchandise record.

Create a Mobile Page for Detailed Information

In the previous topic, you modified the sample hybrid app so that, after it starts, it lists all Merchandise records and provides links to detail pages. In this topic, you finish the job by creating a *detailpage* view and updating the app's controller.

Create the App's detailpage View (index.html)

When a user clicks on a Merchandise record in the app's *mainpage* view, click listeners are in place to generate record-specific information and then load a view named *detailpage* that displays this information. To create the *detailpage* view, add the following div tag after the *mainpage* div tag.

```
<!-- Detail page, to display details when user clicks specific Merchandise record -->

<div data-role="page" data-theme="b" id="detailpage">
    <!-- page header -->
    <div  data-role="header">
        <!-- button for going back to mainpage -->
        <a href='#mainpage' id="backInventory"
```

```
            class='ui-btn-left' data-icon='home'>
             Home
        </a>
        <!-- page title -->
        <h1>Edit</h1>
    </div>
    <!-- page content -->
    <div id="#content" data-role="content">
        <h2 id="name"></h2>
        <label for="price" class="ui-hidden-accessible">
            Price ($):</label>
        <input type="text" id="price" readonly="readonly"></input>
        <br/>
        <label for="quantity" class="ui-hidden-accessible">
           Quantity:</label>
        <!-- note that number is not universally supported -->
        <input type="number" id="quantity"></input>
        <br/>
        <a href="#" data-role="button" id="updateButton"
           data-theme="b">Update</a>
    </div>
</div>
```

The comments explain each part of the HTML. Basically, the view is a form that lets the user see a Merchandise record's Price and Quantity fields, and optionally update the record's Quantity.

Recall, the jQuery calls in the last part of the `onSuccessSfdcMerchandise` function (in `inline.js`) and updates the detail page elements with values from the target Merchandise record. Review that code, if necessary.

Modify the App's Controller (`inline.js`)

What happens when a user clicks the Update button in the new *detailpage* view? Nothing, yet. You need to modify the app's controller (`inline.js`) to handle clicks on that button.

In `inline.js`, add the following JavaScript to the tail end of the `onSuccessSfdcMerchandise` function.

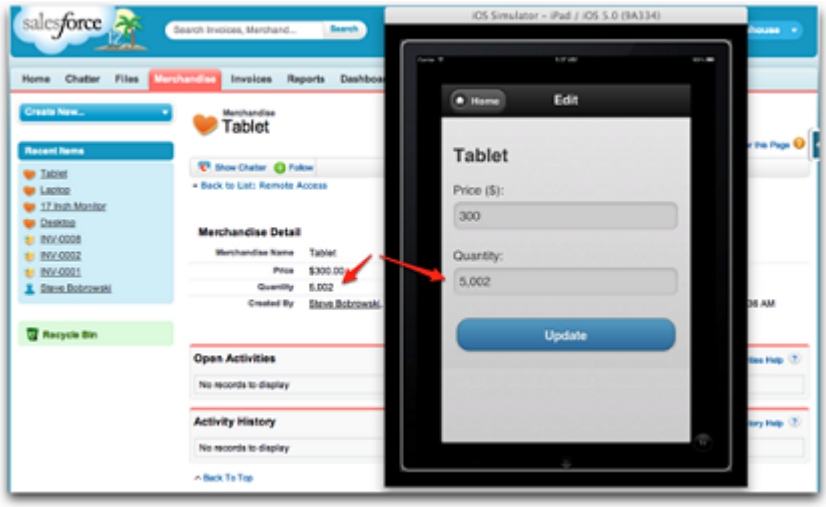
```
// handle clicks to Update on detailpage
$j("#updateButton").click(function() {
    // update local information in the inventory array
    inventory[$j("#detailpage").attr("data-id")].Quantity__c = $j("#quantity").val();
    currentRecord = inventory[$j("#detailpage").attr("data-id")];

    // strip out ID before updating the database
    var data = new Object();
    data.Quantity__c = $j("#quantity").val();
    // update the database
    forcetkClient.update("Merchandise__c", currentRecord.Id,
        data,updateSuccess,onErrorSfdc);
});
```

The comments in the code explain each line. On success, the new handler calls the `updateSuccess` function, which is not currently in place. Add the following simple function to inline.js.

```
function updateSuccess(message) {
    alert("Item Updated");
}
```

Test the App

Restart the simulator for your mobile app. When you do, a detail page should appear when you click a specific Merchandise record and look similar to the following screen.



Feel free to update a record's quantity, and then check that you see the same quantity when you log into your DE org and view the record using the Force.com app UI (see above).

Debugging Hybrid Apps That Are Running On a Mobile Device

You can debug hybrid apps while they're running on a mobile device. How you do it depends on your development platform.

If you run into bugs that show up only when your app runs on a real device, you'll want to use your desktop developer tools to troubleshoot those issues. It's not always obvious to developers how to connect the two runtimes, and it's not well documented in some cases. Here are general platform-specific steps for attaching a Web app debugger on your machine to a running app on a connected device.

Debugging a Hybrid App Running On an Android Device

To debug hybrid apps on Android devices, use Google Chrome.

The following steps summarize the full instructions posted at https://developer.chrome.com/devtools/docs/remote-debugging

1. Enable USB debugging on your device as described at https://developer.chrome.com/devtools/docs/remote-debugging.
2. Open Chrome on your desktop (development) machine and navigate to *`chrome://inspect`*.
3. Select **Discover USB Devices**.
4. Select your device.
5. To use your device to debug a Web application that's running on your development machine:

a. Click **Port forwarding…**.

b. Set the device port and the localhost port.

c. Select **Enable port forwarding**. See https://developer.chrome.com/devtools/docs/remote-debugging#port-forwarding for details.

Debugging a Hybrid App Running On an iOS Device

To debug hybrid apps on iOS devices, use the Safari on the desktop and the device.

1. Open Safari on the desktop.
2. Select **Safari** > **Preferences**.
3. Click the **Advanced** tab.
4. Click **Show Develop menu in menu bar**.
5. If you're using the iOS simulator:
   - If Xcode is open, press `CONTROL` and click the Xcode icon in the task bar, then select **Open Developer Tool** > **iOS Simulator**.
   - Or, in a Terminal window, type `open -a iOS\ Simulator`.
6. In the iOS Simulator menu, select **Hardware** > **Device**.
7. Select a device.
8. Open Safari from the home screen of the device or iOS Simulator.
9. Navigate to the location of your web app.
10. In Safari on your desktop, select **Developer** > ***<your device>***, and then select the URL that you opened in Safari on the device or simulator.

    The Web Inspector window opens and attaches itself to the running Safari instance on your device.

PhoneGap provides instructions for debugging PhoneGap (Cordova) hybrid apps on iOS here. For pre-release documentation from Apple, see https://developer.apple.com/library/prerelease/ios/documentation/IDEs/Conceptual/iOS_Simulator_Guide/.

Controlling the Status Bar in iOS 7 Hybrid Apps

In iOS 7 you can choose to show or hide the status bar, and you can control whether it overlays the web view. You use the Cordova status bar plugin to configure these settings. By default, the status bar is shown and overlays the web view in Salesforce Mobile SDK 2.3 and later.

To hide the status bar, add the following keys to the application plist:

```
<key>UIStatusBarHidden</key>
<true/>
<key>UIViewControllerBasedStatusBarAppearance</key>
<false/>
```

For an example of a hidden status bar, see the AccountEditor sample app.

To control status bar appearance--overlaying, background color, translucency, and so on--add org.apache.cordova.statusbar to your app:

```
cordova plugin add org.apache.cordova.statusbar
```

You control the appearance either from the `config.xml` file or from JavaScript. See https://github.com/apache/cordova-plugin-statusbar for full instructions. For an example of a status bar that doesn't overlay the web view, see the ContactExplorer sample app.

SEE ALSO:

Hybrid Sample Apps

JavaScript Files for Hybrid Apps

External Dependencies

Mobile SDK uses the following external dependencies for various features of hybrid apps.

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

Which JavaScript Files Do I Include?

Beginning with Mobile SDK 2.3, the Cordova utility copies the Cordova plugin files your application needs to your project's platform directories. You don't need to add those files to your www/ folder.

Files that you include in your HTML code (with a <script> tag> depend on the type of hybrid project. For each type described here, include all files in the list.

For basic hybrid apps:

- `cordova.js`

To make REST API calls from a basic hybrid app:

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

To use SmartSync in a hybrid app:

- `jquery.js`
- `underscore.js`
- `backbone.js`
- `cordova.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, if 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 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
2.1	Cordova 2.3
2.2	Cordova 2.3
2.3	Cordova 3.5
3.0	Cordova 3.6

Using the User Agent to Find the Mobile SDK Version

You can leverage the user agent string to look up the Mobile SDK version. 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("com.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 Apps

To help iron out common difficulties that can affect 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 uses 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 strategy gives developers more control over managing sessions in the Salesforce Touch Platform.

If you're updating an app that was developed in Mobile SDK 1.3 or earlier, you'll need to upgrade your session management code. 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 and Later	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("com.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 at `SalesforceMobileSDK-Android/hybrid/SampleApps/ContactExplorer` or `SalesforceMobileSDK-iOS/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 use case: Reloading the entire page doesn't always provide the best user experience. 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

Instead of fully reloading the page as follows:

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

Do something like this:

```
// Refresh oauth token
cordova.require("com.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
        // Get the provider array
        var providers = Visualforce.remoting.Manager.providers;
        // Get the last provider in the arrays (usually only one)
        var provider = Visualforce.remoting.last;
        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();
  }
});
```

Remove SmartStore and SmartSync From an Android Hybrid App

When you create a hybrid Android app with forcedroid 3.0 or later, SmartStore and SmartSync modules are automatically included. If your app doesn't use these modules, you can easily remove them.

1. Open your project's `AndroidManifest.xml` in an XML editor, and find the `<application>` node.
2. Change the `android:name` attribute to the following:

   ```
   android:name="com.salesforce.androidsdk.app.HybridApp"
   ```

3. Edit your `project.properties` file. In Eclipse, right-click the project name in the Package Explorer and choose **Properties**.
4. Select **Android** in the left panel.
5. In the Library section, highlight the entry for SmartSync and click **Remove**.
6. Click **Add...** and select "SalesforceSDK", then click **OK**.

   Note: If you prefer to edit the `project.properties` file directly, replace this path:

   ```
   ..\\..\\plugins\\com.salesforce\\src\\android\\libs\\SmartSync
   ```

   with this one:

   ```
   ..\\..\\plugins\\com.salesforce\\src\\android\\libs\\SalesforceSDK
   ```

7. (Optional) In the Project Explorer, select the SmartStore and SmartSync libraries, then right-click and click **Delete**.

These steps remove SmartStore and SmartSync library references, and change your app's base application class from the `HybridAppWithSmartSync` Mobile SDK class to the more generic `HybridApp` class.

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 6 Offline Management

In this chapter ...

Salesforce Mobile SDK provides two modules that help you store and synchronize data for offline use:

- SmartStore lets you store app data in encrypted databases, or *soups*, on the device. When the device goes back online, you can use SmartStore APIs to synchronize data changes with the Salesforce server.
- SmartSync is a data framework that provides a mechanism for easily fetching Salesforce data, modeling it as JavaScript objects, and caching it for offline use. When it's time to upload offline changes to the Salesforce server, SmartSync gives you highly granular control over the synchronization process. SmartSync is built on the popular `Backbone.js` open source library and uses SmartStore as its default cache.

Using SmartStore to Securely Store Offline Data

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.

Mobile SDK provides SmartStore, a multi-threaded, secure solution for offline storage on mobile devices. With SmartStore, your users can continue working with data in a secure environment even when the device loses connectivity.

About SmartStore

SmartStore stores data as JSON documents in a simple, single-table database. You can define indexes for this database, and you can query the data either with SmartStore helper methods that implement standard queries, or with custom queries using SmartStore's Smart SQL language.

SmartStore uses StoreCache, a Mobile SDK caching mechanism, to provide offline synchronization and conflict resolution services. We recommend that you use StoreCache to manage operations on Salesforce data.

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.

About the Sample Code

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.

SEE ALSO:

- Using StoreCache For Offline Caching
- Conflict Detection
- Smart SQL Queries

SmartStore Soups

SmartStore stores offline data 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 typically maps to a standard or custom object that you intend to store offline. In addition to storing the data, you can also specify indices that map to fields within the data, for greater ease in customizing data queries.

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's no direct correlation between soups.

Warning: 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 organization sets a short lifetime for the refresh token.

SmartStore Data Types

SmartStore supports the following data types:

Type	Description
`integer`	Signed integer, stored in 4 bytes (SDK 2.1 and earlier) or 8 bytes (SDK 2.2 and later)
`floating`	Floating point value, stored as an 8-byte IEEE floating point number
`string`	Text string, stored with database encoding (UTF-8)

IN THIS SECTION:

Date Representation

Date Representation

SmartStore does not specify a type for dates and times. We recommend that you represent these values with SmartStore data types as shown in the following table:

Type	Format As	Description
`string`	An ISO 8601 string	"YYYY-MM-DD HH:MM:SS.SSS"
`floating`	A Julian day number	The number of days since noon in Greenwich on November 24, 4714 BC according to the proleptic Gregorian calendar. This value can include partial days that are expressed as decimal fractions.
`integer`	Unix time	The number of seconds since 1970-01-01 00:00:00 UTC

Enabling SmartStore in Hybrid Apps

Hybrid apps access SmartStore from JavaScript. To enable offline access in a hybrid mobile application, your Visualforce or HTML page must include the `cordova.js` library file.

In Android apps, SmartStore is an optional component. When you use the Mobile SDK npm scripts to create Android SmartStore apps:

- If you create an Android project by using *`forcedroid create`* with prompts, specify *`yes`* when you're asked if you want to use SmartStore.
- If you're using *`forcedroid create`* with command-line parameters, specify the optional `--usesmartstore=yes` parameter.

In iOS apps, SmartStore is always included. If you create an iOS project by using *`forceios create`*, you won't find an option to use SmartStore because the libraries are included implicitly.

SEE ALSO:

Creating an Android Project
Creating an iOS Project

Adding SmartStore to Existing Android Apps

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 `libs/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`.

Using Global SmartStore

Under certain circumstances, some applications require access to a SmartStore instance that is not tied to Salesforce authentication. This situation can occur in apps that store application state or other data that does not depend on a Salesforce user, organization, or community. Mobile SDK provides access to a *global* instance of SmartStore that persists throughout the app's life cycle.

Data stored in global SmartStore does not depend on user authentication and therefore is not deleted at logout. Since global SmartStore remains intact after logout, you are responsible for clearing its data when the app exits. Mobile SDK provides APIs for this purpose.

Important: Do not store user-specific data in global SmartStore. Doing so violates Mobile SDK security requirements because user data can persist after the user logs out.

Android APIs

In Android, you access global SmartStore through an instance of `SalesforceSDKManagerWithSmartStore`.

public SmartStore getGlobalSmartStore(String dbName)
Returns a global SmartStore instance with the specified database name. You can set `dbName` to any string other than "smartstore". Set `dbName` to null to use the default global SmartStore database.

public boolean hasGlobalSmartStore(String dbName)
Checks if a global SmartStore instance exists with the specified database name. Set `dbName` to null to verify the existence of the default global SmartStore.

public void removeGlobalSmartStore(String dbName)
Deletes the specified global SmartStore database. You can set this name to any string other than "smartstore". Set `dbName` to null to remove the default global SmartStore.

iOS APIs

In iOS, you access global SmartStore through an instance of `SFSmartStore`.

+ (id)sharedGlobalStoreWithName:(NSString *)storeName
Returns a global SmartStore instance with the specified database name. You can set `storeName` to any string other than "defaultStore". Set `storeName` to `kDefaultSmartStoreName` to use the default global SmartStore.

+ (void)removeSharedGlobalStoreWithName:(NSString *)storeName

Deletes the specified global SmartStore database. You can set `storeName` to any string other than "defaultStore". Set `storeName` to `kDefaultSmartStoreName` to use the default global SmartStore.

Hybrid APIs

JavaScript methods for the SmartStore plugin take an optional Boolean argument that specifies whether to use global SmartStore. If this argument is false or absent, Mobile SDK uses the default user store. For example:

```
var querySoup = function ([isGlobalStore, ]soupName, querySpec, successCB, errorCB);
```

Registering a Soup

Before you try to access a soup, you need to register it.

To register a soup, you provide a soup name and a list of one or more index specifications. Each of the following examples—hybrid, Android, and iOS—builds an index spec array consisting of name, ID, and owner (or parent) ID fields.

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.

Hybrid Apps

The JavaScript function for registering a soup requires 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.

`indexSpecs`

Use the `indexSpecs` array 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".

```
 "indexSpecs":[
    {
        "path":"Name",
        "type":"string"
    }
    {
        "path":"Id",
        "type":"string"
    }
    {
        "path":"ParentId",
        "type":"string"
    }
]
```

Note:

- Index paths are case-sensitive and can include compound paths, such as `Owner.Name`.

- If index entries are missing any fields described in a particular `indexSpecs` array, they will not be tracked in that index.
- The type of the index applies only to the index. When you query an indexed field (for example, "`select {soup:path} from {soup}`") the query returns data of the type that you specified in the index specification.
- It's OK to have a null field in an index column.

`successCallback`

The success callback function you supply takes one argument: the soup name. For example:

```
function(soupName) { alert("Soup " + soupName + " was successfully created"); };
```

When the soup is successfully created, `registerSoup()` calls the success callback function to indicate that 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 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

To find out if a soup already exists, use:

```
navigator.smartstore.soupExists(soupName, successCallback, errorCallback);
```

Android Native Apps

For Android, you define index specs in an array of type `com.salesforce.androidsdk.smartstore.store.IndexSpec`.

```
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 Native Apps

For iOS, you define index specs in an array of `SFSoupIndex` objects.

```
NSString* const kAccountSoupName = @"Account";
```

```
...

- (SFSmartStore *)store
{
    return [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
}

...

- (void)createAccountsSoup {
    if (![self.store soupExists:kAccountSoupName]) {
        NSArray *keys = @[@"path", @"type"];
        NSArray *nameValues = @[@"Name", kSoupIndexTypeString];
        NSDictionary *nameDictionary = [NSDictionary
            dictionaryWithObjects:nameValues forKeys:keys];

        NSArray *idValues = @[@"Id", kSoupIndexTypeString];
        NSDictionary *idDictionary =
            [NSDictionary dictionaryWithObjects:idValues forKeys:keys];

        NSArray *ownerIdValues = @[@"OwnerId", kSoupIndexTypeString];
        NSDictionary *ownerIdDictionary =
            [NSDictionary dictionaryWithObjects:ownerIdValues
                forKeys:keys];

        NSArray *accountIndexSpecs =
            [SFSoupIndex asArraySoupIndexes:@[nameDictionary,
                idDictionary, ownerIdDictionary]];

        [self.store registerSoup:kAccountSoupName
            withIndexSpecs:accountIndexSpecs];
    }
}
```

SEE ALSO:

SmartStore Data Types

Populating a Soup

To add Salesforce records to a soup for offline access, use the REST API in conjunction with SmartStore APIs.

When you register a soup, you create an empty named structure in memory that's waiting for data. You typically initialize the soup with data from a Salesforce organization. To obtain the Salesforce data, use the Mobile SDK REST request mechanism for your app's platform. When a successful REST response arrives, extract the data from the response object and then upsert it into your soup.

Hybrid Apps

Hybrid apps use SmartStore functions defined in the `forcetk.mobilesdk.js` library. In this example, the click handler for the Fetch Contacts button calls `forcetkClient.query()` to send a simple SOQL query ("`SELECT Name, Id FROM Contact`") to Salesforce. This call designates `onSuccessSfdcContacts(response)` as the callback function that receives the REST

response. The `onSuccessSfdcContacts(response)` function iterates through the returned records in `response` and populates UI controls with Salesforce values. Finally, it upserts all records from the response into the sample soup.

```
// Click handler for the "fetch contacts" button
$('#link_fetch_sfdc_contacts').click(function() {
    logToConsole()("link_fetch_sfdc_contacts clicked");
    forcetkClient.query("SELECT Name,Id FROM Contact",
        onSuccessSfdcContacts, onErrorSfdc);
});

function onSuccessSfdcContacts(response) {
    logToConsole()("onSuccessSfdcContacts: received " + response.totalSize +
        " contacts");
    var entries = [];

    $("#div_sfdc_contact_list").html("");
    var ul = $('<ul data-role="listview" data-inset="true" data-theme="a"
        data-dividertheme="a"></ul>');
    $("#div_sfdc_contact_list").append(ul);

    ul.append($('<li data-role="list-divider">Salesforce Contacts: ' +
        response.totalSize + '</li>'));
    $.each(response.records, function(i, contact) {
           entries.push(contact);
           logToConsole()("name: " + contact.Name);
           var newLi = $("<li><a href='#'>" + (i+1) + " - " + contact.Name +
               "</a></li>");
           ul.append(newLi);
    });

    if (entries.length > 0) {
        sfSmartstore().upsertSoupEntries(SAMPLE_SOUP_NAME,
            entries,
            function(items) {
                var statusTxt = "upserted: " + items.length + " contacts";
                logToConsole()(statusTxt);
                $("#div_soup_status_line").html(statusTxt);
                $("#div_sfdc_contact_list").trigger( "create" );
            },
         onErrorUpsert);
    }
}

function onErrorUpsert(param) {
    logToConsole()("onErrorUpsert: " + param);
    $("#div_soup_status_line").html("Soup upsert ERROR");
}
```

iOS Native Apps

iOS native apps use the `SFRestAPI` protocol for REST API interaction. The following code creates and sends a REST request for the SOQL query `SELECT Name, Id, OwnerId FROM Account`. If the request is successful, Salesforce sends the REST response

to the `requestForQuery:send:delegate:` delegate method. The response is parsed, and each returned record is upserted into the SmartStore soup.

```
- (void)requestAccounts
{
    SFRestRequest *request = [[SFRestAPI sharedInstance]
        requestForQuery:@"SELECT Name, Id, OwnerId FROM Account"];
    [[SFRestAPI sharedInstance] send:request delegate:self];
}

//SFRestAPI protocol for successful response
- (void)request:(SFRestRequest *)request didLoadResponse:(id)dataResponse
{
    NSArray *records = dataResponse[@"records"];
    if (nil != records) {
        for (int i = 0; i < records.count; i++) {
            [self.store upsertEntries:@[records[i]] toSoup:kAccountSoupName];
        }
    }
}
```

Android Native Apps

For REST API interaction, Android native apps typically use the `RestClient.sendAsync()` method with an anonymous inline definition of the `AsyncRequestCallback` interface. The following code creates and sends a REST request for the SOQL query `SELECT Name, Id, OwnerId FROM Account`. If the request is successful, Salesforce sends the REST response to the provided `AsyncRequestCallback.onSuccess()` callback method. The response is parsed, and each returned record is upserted into the SmartStore soup.

```
private void sendRequest(String soql, final String obj) throws UnsupportedEncodingException
 {
 final RestRequest restRequest =
RestRequest.getRequestForQuery(getString(R.string.api_version),
            "SELECT Name, Id, OwnerId FROM Account", "Account");
 client.sendAsync(restRequest, new AsyncRequestCallback() {
  @Override
  public void onSuccess(RestRequest request, RestResponse result) {
   try {
    final JSONArray records = result.asJSONObject().getJSONArray("records");
    insertAccounts(records);
   } catch (Exception e) {
    onError(e);
   } finally {
    Toast.makeText(MainActivity.this, "Records ready for offline access.",
      Toast.LENGTH_SHORT).show();
   }
  }
  }
 });
}

/**
 * Inserts accounts into the accounts soup.
 *
```

```
 * @param accounts Accounts.
 */
public void insertAccounts(JSONArray accounts) {
    try {
        if (accounts != null) {
            for (int i = 0; i < accounts.length(); i++) {
                if (accounts[i] != null) {
                    try {
                        smartStore.upsert(ACCOUNTS_SOUP, accounts[i]);
                    } catch (JSONException exc) {
                        Log.e(TAG, "Error occurred while attempting to
                            insert account. Please verify validity of JSON data set.");
                    }
                }
            }
        }
    } catch (JSONException e) {
  Log.e(TAG, "Error occurred while attempting to insert
                  accounts. Please verify validity of JSON data set.");
    }
}
```

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 Smart SQL 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 Smart SQL support for aggregate functions and joins.

SEE ALSO:

NativeSqlAggregator Sample App: Using SmartStore in Native Apps

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");
});
```

Managing Soups

SmartStore provides utility functionality that lets you retrieve soup metadata and perform other soup-level operations. This functionality is available for hybrid, Android native, and iOS native apps.

Hybrid Apps

Each soup management function in JavaScript takes two callback functions: a success callback that returns the requested data, and an error callback. Success callbacks vary according to the soup management functions that use them. Error callbacks take a single argument, which contains an error description string. For example, you can define an error callback function as follows:

```
function(e) { alert("ERROR: " + e);}
```

To call a soup management function in JavaScript, first invoke the Cordova plugin to initialize the SmartStore object, then call the function. The following example defines named callback functions discretely, but you can also define them inline and anonymously.

```
var sfSmartstore = function() {return cordova.require("com.salesforce.plugin.smartstore");};
function onSuccessRemoveSoup(param) {
    logToConsole()("onSuccessRemoveSoup: " + param);
    $("#div_soup_status_line").html("Soup removed: " + SAMPLE_SOUP_NAME);
}
function onErrorRemoveSoup(param) {
    logToConsole()("onErrorRemoveSoup: " + param);
    $("#div_soup_status_line").html("removeSoup ERROR");
}
sfSmartstore().removeSoup(SAMPLE_SOUP_NAME,
     onSuccessRemoveSoup,
     onErrorRemoveSoup);
```

Android Native Apps

To use soup management APIs in a native Android app that's SmartStore-enabled, you call methods on the shared SmartStore instance:

```
private SalesforceSDKManagerWithSmartStore sdkManager;
private SmartStore smartStore;
sdkManager = SalesforceSDKManagerWithSmartStore.getInstance();
```

```
smartStore = sdkManager.getSmartStore();
smartStore.clearSoup("user1Soup");
```

iOS Native Apps

To use soup management APIs in a native iOS app, import `SFSmartStore.h`. You call soup management methods on a `SFSmartStore` shared instance. Obtain the shared instance by using one of the following `SFSmartStore` class methods.

Using the SmartStore instance for the current user:

```
+ (id)sharedStoreWithName:(NSString*)storeName;
```

Using the SmartStore instance for a specified user:

```
+ (id)sharedStoreWithName:(NSString*)storeName user:(SFUserAccount *)user;
```

For example:

```
self.store = [SFSmartStore sharedStoreWithName:kDefaultSmartStoreName];
if ([self.store soupExists:@"Accounts"]) {
    [self.store removeSoup:@"Accounts"];
}
```

IN THIS SECTION:

Get the Database Size
To query the amount of disk space consumed by the database, call the applicable database size method.

Clear a Soup
To remove all entries from a soup, call the applicable soup clearing method.

Retrieve a Soup's Index Specs
If you want to examine or display the index specifications for a soup, call the applicable index specs retrieval method.

Change Existing Index Specs On a Soup
To change existing index specs, call the applicable soup alteration method.

Reindex a Soup
Use reindexing if you previously altered a soup without reindexing the data, but later you want to make sure all elements in the soup are properly indexed.

Remove a Soup
Removing a soup deletes it. When a user signs out, all soups are deleted automatically. For other occasions in which you want to delete a soup, call the applicable soup removal method.

SEE ALSO:

Adding SmartStore to Existing Android Apps

Get the Database Size

To query the amount of disk space consumed by the database, call the applicable database size method.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.getDatabaseSize(successCallback, errorCallback)
```

The success callback supports a single parameter that contains the database size in bytes. For example:

```
function(dbSize) { alert("db file size is:" + dbSize + " bytes"); }
```

Android Native Apps

In Android apps, call:

```
public int getDatabaseSize ()
```

iOS Native Apps

In Android apps, call:

```
- (long)getDatabaseSize
```

Clear a Soup

To remove all entries from a soup, call the applicable soup clearing method.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.clearSoup(soupName, successCallback, errorCallback)
```

The success callback supports a single parameter that contains the soup name. For example:

```
function(soupName) { alert("Soup " + soupName + " was successfully emptied"); }
```

Android Apps

In Android apps, call:

```
public void clearSoup ( String soupName )
```

iOS Apps

In iOS apps, call:

```
- (void)clearSoup:(NSString*)soupName;
```

Retrieve a Soup's Index Specs

If you want to examine or display the index specifications for a soup, call the applicable index specs retrieval method.

Hybrid Apps

In hybrid apps, call:

```
getSoupIndexSpecs()
```

In addition to the success and error callback functions, this function takes a single argument, `soupName`, which is the name of the soup. For example:

```
navigator.smartstore.getSoupIndexSpecs(soupName, successCallback, errorCallback)
```

The success callback supports a single parameter that contains the array of index specs. For example:

```
function(indexSpecs) { alert("Soup " + soupName + " has the following indexes:" +
JSON.stringify(indexSpecs); }
```

Android Apps

In Android apps, call:

```
public IndexSpec [] getSoupIndexSpecs ( String soupName )
```

iOS Apps

In iOS apps, call:

```
- (NSArray*)indicesForSoup:(NSString*)soupName
```

Change Existing Index Specs On a Soup

To change existing index specs, call the applicable soup alteration method.

Keep these important points in mind when reindexing data:

- The `reIndexData` argument is optional, because re-indexing can be expensive. When `reIndexData` is set to false, expect your throughput to be faster by an order of magnitude.
- Altering a soup that already contains data can degrade your app's performance. Setting `reIndexData` to true worsens the performance hit.
- As a performance guideline, expect the `alterSoup()` operation to take one second per thousand records when `reIndexData` is set to true. Individual performance varies according to device capabilities, the size of the elements, and the number of indexes.
- Be aware that other SmartStore operations must wait for the soup alteration to complete.
- If the operation is interrupted--for example, if the user exits the application--the operation automatically resumes when the application re-opens the SmartStore database.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.alterSoup(soupName, indexSpecs, reIndexData, successCallback,
errorCallback)
```

In addition to the success and error callback functions, this function takes additional arguments:

Parameter Name	Argument Description
`soupName`	String. Pass in the name of the soup.
`indexSpecs`	Array. Pass in the set of index entries in the index specification.
`reIndexData`	Boolean. Indicate whether you want the function to re-index the soup after replacing the index specifications.

The success callback supports a single parameter that contains the soup name. For example:

```
function(soupName) { alert("Soup " + soupName + " was successfully altered"); }
```

The following example demonstrates a simple soup alteration. To start, the developer defines a soup that's indexed on `name` and `address` fields, and then upserts an agent record.

```
navigator.smartstore.registerSoup("myAgents", [{path:'name',type:'string'}, {path:'address',
 type:'string'}]);
navigator.smartstore.upsertSoupEntries("myAgents", [{name:'James Bond', address:'1 market
 st', agentNumber:"007"}]);
```

When time and experience show that users really wanted to query their agents by "agentNumber" rather than `address`, the developer decides to drop the index on `address` and add an index on `agentNumber`.

```
navigator.smartstore.alterSoup("myAgents", [{path:'name',type:'string'}, {path:'agentNumber',
 type:'string'}], true);
```

Note: If the developer sets the `reIndexData` parameter to false, a query on `agentNumber` does not find the already inserted entry ("James Bond"). However, you can query that record by `name`. To support queries by `agentNumber`, you'd first have to call `navigator.smartstore.reIndexSoup("myAgents", ["agentNumber"])`

Android Native Apps

In an Android native app, call:

```
public void alterSoup(String soupName, IndexSpec [] indexSpecs, boolean reIndexData) throws
 JSONException
```

iOS Native Apps

In an iOS native app, call:

```
- (BOOL) alterSoup:(NSString*)soupName withIndexSpecs:(NSArray*)indexSpecs
reIndexData:(BOOL)reIndexData;
```

Reindex a Soup

Use reindexing if you previously altered a soup without reindexing the data, but later you want to make sure all elements in the soup are properly indexed.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.reIndexSoup(soupName, listOfPaths, successCallback, errorCallback)
```

In addition to the success and error callback functions, this function takes additional arguments:

Parameter Name	Argument Description
`soupName`	String. Pass in the name of the soup.
`listOfPaths`	Array. List of index paths on which you want to re-index.

The success callback supports a single parameter that contains the soup name. For example:

```
function(soupName) { alert("Soup " + soupName + " was successfully re-indexed."); }
```

Android Apps

In Android apps, call:

```
public void reIndexSoup(String soupName, String[] indexPaths, boolean handleTx)
```

iOS Apps

In iOS apps, call:

```
- (BOOL) reIndexSoup:(NSString*)soupName withIndexPaths:(NSArray*)indexPaths
```

Remove a Soup

Removing a soup deletes it. When a user signs out, all soups are deleted automatically. For other occasions in which you want to delete a soup, call the applicable soup removal method.

Hybrid Apps

In hybrid apps, call:

```
navigator.smartstore.removeSoup(soupName,successCallback,errorCallback);
```

Android Apps

In Android apps, call:

```
public void dropSoup ( String soupName )
```

iOS Apps

In iOS apps, call:

```
- (void)removeSoup:(NSString*)soupName
```

Testing With the SmartStore Inspector

During testing, you'll want to be able to see if your code is handling SmartStore data as you intended. The SmartStore Inspector provides a mobile UI for that purpose. With the SmartStore Inspector you can:

- Examine soup metadata, such as soup names and index specs for any soup
- Clear a soup's contents
- Perform Smart SQL queries

Note: SmartStore Inspector is for testing and debugging only. Be sure to remove all references to SmartStore Inspector before you build the final version of your app.

Hybrid Apps

To launch the SmartStore Inspector, call `showInspector()` on the SmartStore plugin object. In HTML:

```
<!-- include Cordova -->
<script src="cordova.js"></script>
```

In a `<script>` block or a referenced JavaScript library:

```
var sfSmartstore = function() {return cordova.require("com.salesforce.plugin.smartstore");};
sfSmartstore().showInspector();
```

Android Native Apps

In native Android apps, use the `SmartStoreInspectorActivity` class to launch the SmartStore Inspector:

```
final Intent i = new Intent(activity, SmartStoreInspectorActivity.class);
activity.startActivity(i);
```

iOS Native Apps

In native iOS apps, send the class-level `SFSmartStoreInspectorViewController:present` message to launch the SmartStore Inspector:

```
#import <SalesforceSDKCore/SFSmartStoreInspectorViewController.h>
...
[SFSmartStoreInspectorViewController present];
```

The `SFSmartStoreInspectorViewController:present` class manages its own life cycle, but if you need to dismiss the `SFSmartStoreInspectorViewController:present` for some unusual reason, send the class-level `SFSmartStoreInspectorViewController:dismiss` message:

```
[SFSmartStoreInspectorViewController dismiss];
```

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.

MockSmartStore is a JavaScript implementation of SmartStore that stores the data in local storage (or optionally just in memory).

In the `external/shared/test` directory, you'll find the following files:

- `MockCordova.js`—A minimal implementation of Cordova functions meant only for testing plugins outside the container. Intercepts Cordova plugin calls
- `MockSmartStore.js`—A JavaScript implementation of the SmartStore meant only for development and testing outside the container. Also intercepts SmartStore Cordova plugin calls and handles them using a MockSmartStore.

When you're developing an application using SmartStore, make the following changes to test your app outside the container:

- Include `MockCordova.js` instead of `cordova.js`.
- Include `MockSmartStore.js`.

To see a MockSmartStore example, check out `external/shared/test/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.

Note:

- MockSmartStore doesn't encrypt data and is not meant to be used in production applications.
- MockSmartStore currently supports the following forms of Smart SQL queries:
  - `SELECT...WHERE...`. For example:

    ```
    SELECT {soupName:selectField} FROM {soupName} WHERE {soupName:whereField} IN
    (values)
    ```

  - `SELECT...WHERE...ORDER BY...`. For example:

    ```
    SELECT {soupName:_soup} FROM {soupName} WHERE {soupName:whereField} LIKE 'value'
     ORDER BY LOWER({soupName:orderByField})
    ```

  - `SELECT count(*) FROM {soupName}`

  MockSmartStore doesn't directly support the simpler types of Smart SQL statements that are handled by the `build*QuerySpec()` functions. Instead, use the query spec function that suits your purpose.

SEE ALSO:

Retrieving Data From a Soup

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:

```
NSString* const kAccountSoupName = @"Account";

...

- (void)createAccountsSoup {
    if (![self.store soupExists:kAccountSoupName]) {
        NSArray *keys = @[@"path", @"type"];
        NSArray *nameValues = @[@"Name", kSoupIndexTypeString];
        NSDictionary *nameDictionary = [NSDictionary
            dictionaryWithObjects:nameValues forKeys:keys];
        NSArray *idValues = @[@"Id", kSoupIndexTypeString];
        NSDictionary *idDictionary =
            [NSDictionary dictionaryWithObjects:idValues forKeys:keys];
        NSArray *ownerIdValues = @[@"OwnerId", kSoupIndexTypeString];
        NSDictionary *ownerIdDictionary =
            [NSDictionary dictionaryWithObjects:ownerIdValues
                forKeys:keys];
        NSArray *accountIndexSpecs =
            [SFSoupIndex asArraySoupIndexes:@[nameDictionary,
                idDictionary, ownerIdDictionary]];
        [self.store registerSoup:kAccountSoupName
            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.

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];
```

Using SmartSync to Access Salesforce Objects

The SmartSync library is a collection of APIs that make it easy for developers to sync data between Salesforce databases and their mobile apps. It provides the means for getting and posting data to a server endpoint, caching data on a device, and reading cached data. For sync operations, SmartSync predefines cache policies for fine-tuning interactions between cached data and server data in offline and online scenarios. A set of SmartSync convenience methods automate common network activities, such as fetching sObject metadata, fetching a list of most recently used objects, and building SOQL and SOSL queries.

Native

Using SmartSync in Native Apps

The native SmartSync library provides native iOS and Android APIs that simplify the development of offline-ready apps. A subset of this native functionality is also available to hybrid apps through a Cordova plugin.

SmartSync libraries offer parallel architecture and functionality for Android and iOS, expressed in each platform's native language. The shared functional concepts are straightforward:

- Query Salesforce object metadata by calling Salesforce REST APIs.
- Store the retrieved object data locally and securely for offline use.
- Sync data changes when the device goes from an offline to an online state.

With SmartSync native libraries, you can:

- Get and post data by interacting with a server endpoint. SmartSync helper APIs encode the most commonly used endpoints. These APIs help you fetch sObject metadata, retrieve the list of most recently used (MRU) objects, and build SOQL and SOSL queries. You can also use arbitrary endpoints that you specify in a custom class.
- Fetch Salesforce records and metadata and cache them on the device, using one of the pre-defined cache policies.
- Edit records offline and save them offline in SmartStore.
- Synchronize batches of records by pushing locally modified data to the Salesforce cloud.

SmartSync Components

The following components form the basis of SmartSync architecture.

Sync Manager

- **Android class:** `com.salesforce.androidsdk.smartsync.manager.SyncManager`
- **iOS class:** `SFSmartSyncSyncManager`

Provides APIs for synchronizing large batches of sObjects between the server and SmartStore. This class works independently of the metadata manager and is intended for the simplest and most common sync operations. Sync managers can "sync down"—download sets of sObjects from the server to SmartStore—and "sync up"—upload local sObjects to the server.

The sync manager works in tandem with the following utility classes:

• **Android:** `com.salesforce.androidsdk.smartsync.util.SyncState` • **iOS:** `SFSyncState`	Tracks the state of a sync operation. States include: • New—The sync operation has been initiated but has not yet entered a transaction with the server.

	• Running—The sync operation is negotiating a sync transaction with the server. • Done—The sync operation finished successfully. • Failed—The sync operation finished unsuccessfully.
• **Android:** `com.salesforce.androidsdk.smartsync.util.SyncTarget` • **iOS:** `SFSyncTarget`	Specifies the sObjects to be downloaded during a "sync down" operation.
• **Android:** `com.salesforce.androidsdk.smartsync.util.SyncOptions` • **iOS:** `SFSyncOptions`	Specifies configuration options for a "sync up" operation. Options include the list of field names to be synced.

Metadata Manager

- **Android class:** `com.salesforce.androidsdk.smartsync.manager.MetadataManager`
- **iOS class:** `SFSmartSyncMetadataManager`

Performs data loading functions. This class helps you handle more full-featured queries and configurations than the sync manager protocols support. For example, metadata manager APIs can:

- Load SmartScope object types.
- Load MRU lists of sObjects. Results can be either global or limited to a specific sObject.
- Load the complete object definition of an sObject, using the `describe` API.
- Load the list of all sObjects available in an organization.
- Determine if an sObject is searchable, and, if so, load the search layout for the sObject type.
- Load the color resource for an sObject type.
- Mark an sObject as viewed on the server, thus moving it to the top of the MRU list for its sObject type.

To interact with the server, `MetadataManager` uses the standard Mobile SDK REST API classes:

- **Android:** `RestClient`, `RestRequest`
- **iOS:** `SFRestAPI`, `SFRestRequest`

It also uses the SmartSync cache manager to read and write data to the cache.

Cache Manager

- **Android class:** `com.salesforce.androidsdk.smartsync.manager.CacheManager`
- **iOS class:** `SFSmartSyncCacheManager`

Reads and writes objects, object types, and object layouts to the local cache on the device. It also provides a method for removing a specified cache type and cache key. The cache manager stores cached data in a SmartStore database backed by SQLCipher. Though the cache manager is not off-limits to apps, the metadata manager is its principle client and typically handles all interactions with it.

SOQL Builder

- **Android class:** `com.salesforce.androidsdk.smartsync.util.SOQLBuilder`
- **iOS class:** `SFSmartSyncSoqlBuilder`

Utility class that makes it easy to build a SOQL query statement, by specifying the individual query clauses.

SOSL Builder

- **Android class:** `com.salesforce.androidsdk.smartsync.util.SOSLBuilder`
- **iOS class:** `SFSmartSyncSoslBuilder`

Utility class that makes it easy to build a SOSL query statement, by specifying the individual query clauses.

SmartSyncSDKManager (Android only)

For Android, SmartSync apps use a different SDK manager object than basic apps. Your `App` class extends `SmartSyncSDKManager` instead of `SalesforceSDKManager`. If you create a SmartStore app with forcedroid version 3.0 or later, this substitution happens automatically. This change applies to both native and hybrid SmartSync apps on Android.

Note: To support multi-user switching, SmartSync creates unique instances of its components for each user account.

Cache Policies

When you're updating your app data, you can specify a cache policy to tell SmartSync how to handle the cache. You can choose to sync with server data, use the cache as a fallback when the server update fails, clear the cache, ignore the cache, and so forth. For Android, cache policies are defined in the `com.salesforce.androidsdk.smartsync.manager.CacheManager.CachePolicy` class. For iOS, they're part of the `SFDataCachePolicy` enumeration defined in `SFSmartSyncCacheManager.h`.

You specify a cache policy every time you call any metadata manager method that loads data. For example, here are the Android `MetadataManager` data loading methods:

```
public List<SalesforceObjectType>
   loadSmartScopeObjectTypes(CachePolicy cachePolicy,
   long refreshCacheIfOlderThan);

public List<SalesforceObject> loadMRUObjects(String objectTypeName,
   int limit, CachePolicy cachePolicy, long refreshCacheIfOlderThan,
   String networkFieldName);

public List<SalesforceObjectType> loadAllObjectTypes(CachePolicy cachePolicy,
   long refreshCacheIfOlderThan);

public SalesforceObjectType loadObjectType(String objectTypeName,
    CachePolicy cachePolicy, long refreshCacheIfOlderThan);

public List<SalesforceObjectType> loadObjectTypes(List<String> objectTypeNames,
    CachePolicy cachePolicy, long refreshCacheIfOlderThan);
```

You also specify cache policy to help the cache manager decide if it's time to reload the cache:

Android:

```
 public boolean needToReloadCache(boolean cacheExists,
    CachePolicy cachePolicy, long lastCachedTime, long refreshIfOlderThan);
```

iOS:

```
- (BOOL)needToReloadCache:(BOOL)cacheExists
    cachePolicy:(SFDataCachePolicy)cachePolicy
    lastCachedTime:(NSDate *)cacheTime
    refreshIfOlderThan:(NSTimeInterval)refreshIfOlderThan;
```

Here's a list of cache policies.

Table 5: Cache Policies

Cache Policy (iOS)	Cache Policy (Android)	Description
`IgnoreCacheData`	`IGNORE_CACHE_DATA`	Ignores cached data. Always goes to the server for fresh data.
`ReloadAndReturnCacheOnFailure`	`RELOAD_AND_RETURN_CACHE_ON_FAILURE`	Attempts to load data from the server, but falls back on cached data if the server call fails.
`ReturnCacheDataDontReload`	`RETURN_CACHE_DATA_DONT_RELOAD`	Returns data from the cache,\ and doesn't attempt to make a server call.
`ReloadAndReturnCacheData`	`RELOAD_AND_RETURN_CACHE_DATA`	Reloads data from the server and updates the cache with the new data.
`ReloadIfExpiredAndReturnCacheData`	`RELOAD_IF_EXPIRED_AND_RETURN_CACHE_DATA`	Reloads data from the server if cache data has become stale (that is, if the specified timeout has expired). Otherwise, returns data from the cache.
`InvalidateCacheDontReload`	`INVALIDATE_CACHE_DONT_RELOAD`	Clears the cache and does not reload data from the server.
`InvalidateCacheAndReload`	`INVALIDATE_CACHE_AND_RELOAD`	Clears the cache and reloads data from the server.

Object Representation

When you use the metadata manager, SmartSync model information arrives as a result of calling metadata manager load methods. The metadata manager loads the data from the current user's organization and presents it in one of three classes:

- Object
- Object Type
- Object Type Layout

Object

- **Android class**: `com.salesforce.androidsdk.smartsync.model.SalesforceObject`
- **iOS class**: `SFObject`

These classes encapsulate the data that you retrieve from an sObject in Salesforce. The object class reads the data from a `JSONObject` in Android, or an `NSDictionary` object in iOS, that contains the results of your query. It then stores the object's ID, type, and name as properties. It also stores the `JSONObject` itself as raw data.

Object Type

- **Android class** `com.salesforce.androidsdk.smartsync.model.SalesforceObjectType`
- **iOS class** `SFObjectType`

The object type class stores details of an sObject, including the prefix, name, label, plural label, and fields.

Object Type Layout

- **Android class** `com.salesforce.androidsdk.smartsync.model.SalesforceObjectTypeLayout`
- **iOS class** `SFObjectTypeLayout`

The object type layout class retrieves the columnar search layout defined for the sObject in the organization, if one is defined. If no layout exists, you're free to choose the fields you want your app to display and the format in which to display them.

SEE ALSO:

Cache Policies

Creating SmartSync Native Apps

Creating native SmartSync apps in forceios version 3.0 and later literally requires no extra effort. Any native forceios app you create automatically includes the SmartStore and SmartSync libraries.

In Android, you simply need to specify an extra parameter in forcedroid version 3.0 or later. Set `--usesmartstore=yes` if you hard-code the forcedroid parameters. If you use `forcedroid create` interactively, answer "yes" when forcedroid asks, "Do you want to use SmartStore or SmartSync in your app?". In forcedroid 3.0 and later, SmartStore support includes the SmartSync library.

Adding SmartSync to Existing Android Apps

The following steps show you how to add SmartSync to an existing Android project (hybrid or native) created with Mobile SDK 2.3 or later.

1. If your app is currently built on Mobile SDK 2.2 or earlier, upgrade your project to the latest SDK version as described in Migrating from the Previous Release on page 309.
2. Add the SmartSync library project to your project. SmartSync uses SmartStore, so you also need to add that library if your project wasn't originally built with SmartStore.
   a. In Eclipse, choose **Project** > **Properties**.
   b. In the left panel, select **Android**, then click **Add**.
   c. Choose the `libs/SmartSync` project and, if it isn't already referenced, the `libs/SmartStore` project.
3. Throughout your project, change all code that uses the `SalesforceSDKManager` object to use `SmartSyncSDKManager` instead.

   Note: If you do a project-wide search and replace, be sure *not* to change the `KeyInterface` import, which should remain

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

Adding SmartSync to Existing iOS Apps

Adding SmartSync to existing native iOS apps is easy. Any app that is created with the 3.0 version of forceios automatically includes the libraries you need. If your app is based on an earlier version of Mobile SDK, first follow the upgrade steps outlined in Migrating from the Previous Release on page 309. Once your app is up-to-date with Mobile SDK 3.0, you're ready to start using the native SmartSync classes.

Syncing Data

In native SmartSync apps, you can use the sync manager to sync data easily between the device and the Salesforce server. The sync manager provides methods for syncing "up"—from the device to the server—or "down"—from the server to the device.

All data requests in SmartSync apps are asynchronous. Asynchronous means that the sync method that you call returns the server response in a callback method or update block that you define.

Each sync up or sync down method returns a sync state object. This object contains the following information:

- Sync operation ID. You can check the progress of the operation at any time by passing this ID to the sync manager's `getSyncStatus` method.
- Your sync parameters (soup name, target for sync down operations, options for sync up operations).
- Type of operation (up or down).
- Progress percentage (integer, 0–100).
- Total number of records in the transaction.

Using the Sync Manager

The sync manager object performs simple sync up and sync down operations. For sync down, it sends authenticated requests to the server on your behalf, and stores response data locally in SmartStore. For sync up, it collects the records you specify from SmartStore and merges them with corresponding server records according to your instructions. Sync managers know how to handle authentication for Salesforce users and community users. Sync managers can store records in any SmartStore instance—the default SmartStore, the global SmartStore, or a named instance.

Sync manager classes provide factory methods that return customized sync manager instances. To use the sync manager, you create an instance that matches the requirements of your sync operation. It is of utmost importance that you create the correct type of sync manager for every sync activity. If you don't, your customers can encounter runtime authentication failures.

Once you've created an instance, you can use it to call typical sync manager functionality:

- Sync down
- Sync up
- Resync

Sync managers can perform three types of actions on SmartStore soup entries and Salesforce records:

- Create
- Update
- Delete

If you provide custom targets, sync managers can use them to synchronize data at arbitrary REST endpoints.

SyncManager Instantiation (Android)

In Android, you use a different factory method for each of the following scenarios:

For the current user:

```
public static synchronized SyncManager getInstance();
```

For a specified user:

```
public static synchronized SyncManager getInstance(UserAccount account);
```

For a specified user in a specified community:

```
public static synchronized SyncManager getInstance(UserAccount account, String
communityId);
```

For a specified user in a specified community using the specified SmartStore database:

```
public static synchronized SyncManager getInstance(UserAccount account, String
communityId, SmartStore smartStore);
```

SFSmartSyncSyncManager Instantiation (iOS)

In iOS, you use pairs of access and removal methods. You call the sharedInstance: class methods on the `SFSmartSyncSyncManager` class to access a preconfigured shared instance for each scenario. When you're finished using the shared instance for a particular use case, remove it with the corresponding `removeSharedInstance*:...` method.

For a specified user:

```
+ (instancetype)sharedInstance:(SFUserAccount *)user;
+ (void)removeSharedInstance:(SFUserAccount *)user;
```

For a specified user using the specified SmartStore database:

```
+ (instancetype)sharedInstanceForUser:(SFUserAccount *)user storeName:(NSString
*)storeName;
+ (void)removeSharedInstanceForUser:(SFUserAccount *)user storeName:(NSString *)storeName;
```

For a specified SmartStore database:

```
+ (instancetype)sharedInstanceForStore:(SFSmartStore *)store;
+ (void)removeSharedInstanceForStore:(SFSmartStore *)store;
```

Syncing Down

To download sObjects from the server to your local SmartSync soup, use the "sync down" method:

- **Android `SyncManager` methods:**

```
public SyncState syncDown(SyncTarget target, String soupName,
    SyncUpdateCallback callback) throws JSONException;

public SyncState syncDown(SyncTarget target, SyncOptions options, String soupName,
    SyncUpdateCallback callback) throws JSONException;
```

- **iOS `SFSmartSyncSyncManager` methods:**

```
- (SFSyncState*) syncDownWithTarget:(SFSyncTarget*)target
                           soupName:(NSString*)soupName
                      updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

```
- (SFSyncState*) syncDownWithTarget:(SFSyncTarget*)target
                            options:(SFSyncOptions*)options
                         soupName:(NSString*)soupName
                       updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

For "sync down" methods, you define a target that provides the list of sObjects to be downloaded. To provide an explicit list, use `JSONObject` on Android, or `NSDictionary` on iOS. However, you can also define the target with a query string. The sync target class provides factory methods for creating target objects from a SOQL, SOSL, or MRU query.

You also specify the name of the SmartStore soup that receives the downloaded data. This soup is required to have an indexed string field named `__local__`. Mobile SDK reports the progress of the sync operation through the callback method or update block that you provide.

Merge Modes

The `options` parameter lets you control what happens to locally modified records. You can choose one of the following behaviors:

1. Overwrite modified local records and lose all local changes. Set the `options` parameter to the following value:
   - **Android:** `SyncOptions.optionsForSyncDown(MergeMode.OVERWRITE)`
   - **iOS:** `[SFSyncOptions newSyncOptionsForSyncDown:SFSyncStateMergeModeOverwrite]`
2. Preserve all local changes and locally modified records. Set the `options` parameter to the following value:
   - **Android:** `SyncOptions.optionsForSyncDown(MergeMode.LEAVE_IF_CHANGED)`
   - **iOS:** `[SFSyncOptions newSyncOptionsForSyncDown:SFSyncStateMergeModeLeaveIfChanged])`

Important: If you use a version of `syncDown` that doesn't take an `options` parameter, existing sObjects in the cache can be overwritten. To preserve local changes, always run sync up before running sync down.

Example: **Android:**

The native SmartSyncExplorer sample app demonstrates how to use SmartSync with Contact records. In Android, it defines a `ContactObject` class that represents a Salesforce Contact record as a Java object. To sync Contact data down to the SmartStore soup, the `syncDownContacts` method creates a sync target from a SOQL query that's built with information from the `ContactObject` instance.

In the following snippet, note the use of `SOQLBuilder`. `SOQLBuilder` is a SmartSync factory class that makes it easy to specify a SOQL query dynamically in a format that reads like an actual SOQL string. Each `SOQLBuilder` property setter returns a new SOQLBuilder object built from the calling object, which allows you to chain the method calls in a single logical statement. After you've specified all parts of the SOQL query, you call `build()` to create the final SOQL string.

```
private void syncDownContacts() {
    smartStore.registerSoup(ContactListLoader.CONTACT_SOUP, CONTACTS_INDEX_SPEC);
    try {
        final String soqlQuery =
SOQLBuilder.getInstanceWithFields(ContactObject.CONTACT_FIELDS)
      .from(Constants.CONTACT).limit(ContactListLoader.LIMIT).build();
      final SyncTarget target = SyncTarget.targetForSOQLSyncDown(soqlQuery);
      syncMgr.syncDown(target, ContactListLoader.CONTACT_SOUP,
                new SyncUpdateCallback() {
        @Override
        public void onUpdate(SyncState sync) {
            handleSyncUpdate(sync);
```

```
        }
     });
  } catch (JSONException e) {
            Log.e(TAG, "JSONException occurred while parsing", e);
    }
}
```

If the sync down operation succeeds—that is, if `SyncState.isDone()` equals true—the received data goes into the specified soup. The callback method then needs only a trivial implementation, as carried out in the `handleSyncUpdate()` method:

```
private void handleSyncUpdate(SyncState sync) {
    if (Looper.myLooper() == null) {
        Looper.prepare();
    }
    if (sync.isDone()) {
        switch(sync.getType()) {
            case syncDown:
                Toast.makeText(MainActivity.this,
                               Sync down successful!",
    Toast.LENGTH_LONG).show();
                break;
            case syncUp:
                Toast.makeText(MainActivity.this,
      "Sync up successful!",
      Toast.LENGTH_LONG).show();
                syncDownContacts();
                break;
            default:
                break;
         }
    }
}
```

iOS:

The native SmartSyncExplorer sample app demonstrates how to use SmartSync with Contact records. In iOS, this sample defines a `ContactSObjectData` class that represents a Salesforce Contact record as an Objective-C object. The sample also defines several classes that support the `ContactSObjectData` class:

- `ContactSObjectDataSpec`
- `SObjectData`
- `SObjectDataSpec`
- `SObjectDataFieldSpec`
- `SObjectDataManager`

To sync Contact data down to the SmartStore soup, the `refreshRemoteData` method of `SObjectDataManager` creates a `SFSyncTarget` object using a SOQL string. This query string is built with information from the Contact object. The `syncDownWithTarget:soupName:updateBlock:` method of `SFSmartSyncSyncManager` takes this target and the name of the soup that receives the returned data. This method also requires an update block that is called when the sync operation has either succeeded or failed.

```
- (void)refreshRemoteData {
    if (![self.store soupExists:self.dataSpec.soupName]) {
        [self registerSoup];
```

```
    }

    NSString *soqlQuery = [NSString stringWithFormat:@"SELECT %@ FROM %@ LIMIT %d",
        [self.dataSpec.fieldNames componentsJoinedByString:@","],
        self.dataSpec.objectType,
        kSyncLimit];
    SFSyncTarget *syncTarget = [SFSyncTarget newSyncTargetForSOQLSyncDown:soqlQuery];

    __weak SObjectDataManager *weakSelf = self;
    [self.syncMgr syncDownWithTarget:syncTarget
        soupName:self.dataSpec.soupName
        updateBlock:^(SFSyncState* sync) {
            if ([sync isDone] || [sync hasFailed]) {
                [weakSelf refreshLocalData];
            }
        }
    ];
}
```

If the sync down operation succeeds—that is, if the `isDone` method of `SFSyncState` returns YES—the specified soup receives the server data. The update block then passes control to the `refreshLocalData` method, which retrieves the data from the soup and updates the UI to reflect any changes.

```
- (void)refreshLocalData {
    if (![self.store soupExists:self.dataSpec.soupName]) {
        [self registerSoup];
    }

    SFQuerySpec *sobjectsQuerySpec =
        [SFQuerySpec newAllQuerySpec:self.dataSpec.soupName
         withPath:self.dataSpec.orderByFieldName
         withOrder:kSFSoupQuerySortOrderAscending withPageSize:kMaxQueryPageSize];
    NSError *queryError = nil;
    NSArray *queryResults =
        [self.store queryWithQuerySpec:sobjectsQuerySpec
         pageIndex:0
         error:&queryError];
    [self log:SFLogLevelDebug msg:@"Got local query results.  Populating data rows."];

    if (queryError) {
        [self log:SFLogLevelError
              format:@"Error retrieving '%@' data from SmartStore: %@",
                  self.dataSpec.objectType,
                  [queryError localizedDescription]];
        return;
    }

    self.fullDataRowList = [self populateDataRows:queryResults];
    [self log:SFLogLevelDebug
       format:@"Finished generating data rows.  Number of rows: %d.  Refreshing view.",

            [self.fullDataRowList count]];
```

```
    [self resetDataRows];
}
```

Incrementally Syncing Down

For certain target types, you can incrementally resync a previous sync down operation. Mobile SDK fetches only new or updated records if the sync down target supports resync. Otherwise, it reruns the entire sync operation.

Of the three built-in sync down targets (MRU, SOSL-based, and SOQL-based), only the SOQL-based sync down target supports resync. To support resync in custom sync targets, use the `maxTimeStamp` parameter passed during a fetch operation.

During sync down, Mobile SDK checks downloaded records for the modification date field specified by the target and determines the most recent timestamp. If you request a resync for that sync down, Mobile SDK passes the most recent timestamp, if available, to the sync down target. The sync down target then fetches only records created or updated since the given timestamp. The default modification date field is `lastModifiedDate`.

Limitation

After an incremental sync, the following unused records remain in the local soup:

- Deleted records
- Records that no longer satisfy the sync down target

If you choose to remove these orphaned records, you can:

- Run a full sync down operation, or
- Compare the IDs of local records against the IDs returned by a full sync down operation.

Invoking the Re-Sync Method

Android:

On a `SyncManager` instance, call:

```
SyncState reSync(long syncId, SyncUpdateCallback callback);
```

iOS:

On a `SFSmartSyncSyncManager` instance, call:

```
- (SFFSyncState*) reSync:(NSNumber *)syncId
             updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

Hybrid:

Call:

```
cordova.require("com.salesforce.plugin.SmartSync").reSync(syncId,successCB);
```

Sample Apps

Android

The SmartSyncExplorer sample app uses `reSync()` in the `ContactListLoader` class.

iOS

The SmartSyncExplorer sample app uses `reSync()` in the `SObjectDataManager` class.

Hybrid

The SimpleSync sample app uses `reSync()` in `SimpleSync.html`'s `app.views.SearchPage` class.

Syncing Up

To apply local changes on the server, use one of the "sync up" methods:

- **Android `SyncManager` method:**

```
public SyncState syncUp(SyncOptions options, String soupName,
    SyncUpdateCallback callback) throws JSONException
```

- **iOS `SFSmartSyncSyncManager` method:**

```
- (SFSyncState*) syncUpWithOptions:(SFSyncOptions*)options
                          soupName:(NSString*)soupName
                       updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

These methods update the server with data from the given SmartStore soup. They look for created, updated, or deleted records in the soup, and then replicate those changes on the server. The `options` argument specifies a list of fields to be updated.

Locally created objects must include an "attributes" field that contains a "type" field that specifies the sObject type. For example, for an account named Acme, use: `{Id:"local_x", Name: Acme, attributes: {type:"Account"}}`.

Merge Modes

For sync up operations, you can specify a mergeMode option. You can choose one of the following behaviors:

1. Overwrite server records even if they've changed since they were synced down to that client. When you call the `syncUp` method:
   - **Android:** Set the `options` parameter to `SSyncOptions.optionsForSyncUp(fieldlist, SyncState.MergeMode.OVERWRITE)`
   - **iOS:** Set the `options` parameter to `[SFSyncOptions newSyncOptionsForSyncUp:fieldlist mergeMode:SFSyncStateMergeModeOverwrite]`
   - **Hybrid:** Set the `syncOptions` parameter to `{mergeMode:"OVERWRITE"}`
2. If any server record has changed since it was synced down to that client, leave it in its current state. The corresponding client record also remains in its current state. When you call the `syncUp()` method:
   - **Android:** Set the `options` parameter to `SyncOptions.optionsForSyncUp(fieldlist, SyncState.MergeMode.LEAVE_IF_CHANGED)`
   - **iOS:** Set the `options` parameter to `[SFSyncOptions newSyncOptionsForSyncUp:fieldlist mergeMode:SFSyncStateMergeModeLeaveIfChanged]`
   - **Hybrid:** Set the `syncOptions` parameter to `{mergeMode:"LEAVE_IF_CHANGED"}`

If your local record includes the target's modification date field, Mobile SDK detects changes by comparing that field to the matching field in the server record. The default modification date field is `lastModifiedDate`. If your local records do not include the modification date field, the `LEAVE_IF_CHANGED` sync up operation reverts to an overwrite sync up.

Important: The `LEAVE_IF_CHANGED` merge requires extra round trips to the server. More importantly, the status check and the record save operations happen in two successive calls. In rare cases, a record that is updated between these calls can be prematurely modified on the server.

Example: **Android:**

When it's time to sync up to the server, you call `syncUp()` with the same arguments as `syncDown()`: list of fields, name of source SmartStore soup, and an update callback. The only coding difference is that you format the list of affected fields as an instance of `SyncOptions` instead of `SyncTarget`. Here's the way it's handled in the SmartSyncExplorer sample:

```
private void syncUpContacts() {
    final SyncOptions options =
        SyncOptions.optionsForSyncUp(Arrays.asList(ContactObject.CONTACT_FIELDS));
    try {
        syncMgr.syncUp(options, ContactListLoader.CONTACT_SOUP,
            new SyncUpdateCallback() {
            @Override
            public void onUpdate(SyncState sync) {
                handleSyncUpdate(sync);
            }
        });
    } catch (JSONException e) {
        Log.e(TAG, "JSONException occurred while parsing", e);
    }
}
```

In the update callback, the SmartSyncExplorer example takes the extra step of calling `syncDownContacts()` when sync up is done. This step guarantees that the SmartStore soup remains up-to-date with any recent changes made to Contacts on the server.

```
private void handleSyncUpdate(SyncState sync) {
    if (Looper.myLooper() == null) {
        Looper.prepare();
    }
    if (sync.isDone()) {
        switch(sync.getType()) {
            case syncDown:
                Toast.makeText(MainActivity.this,
                            Sync down successful!",
    Toast.LENGTH_LONG).show();
                break;
            case syncUp:
                Toast.makeText(MainActivity.this,
      "Sync up successful!",
      Toast.LENGTH_LONG).show();
                syncDownContacts();
                break;
            default:
                break;
         }
    }
}
```

iOS:

When it's time to sync up to the server, you send the `syncUp:withOptions:soupName:updateBlock:` message to `SFSmartSyncSyncManager` with the same arguments used for syncing down: list of fields, name of source SmartStore

soup, and an update block. The only coding difference is that you format the list of affected fields as an instance of `SFSyncOptions` instead of `SFSyncTarget`. Here's how the SmartSyncExplorer sample sends the sync up message:

```
- (void)updateRemoteData:(SFSyncSyncManagerUpdateBlock)completionBlock {
    SFSyncOptions *syncOptions =
        [SFSyncOptions newSyncOptionsForSyncUp:self.dataSpec.fieldNames];
    [self.syncMgr syncUpWithOptions:syncOptions
        soupName:self.dataSpec.soupName
        updateBlock:^(SFSyncState* sync) {
            if ([sync isDone] || [sync hasFailed]) {
                completionBlock(sync);
            }
        }
    ];
}
```

If the update block provided here determines that the sync operation has finished, it calls the completion block that's passed into `updateRemoteData`. A user initiates a syncing operation by tapping a button. Therefore, to see the definition of the completion block, look at the `syncUpDown` button handler in `ContactListViewController.m`. The handler calls `updateRemoteData` with the following block.

```
[self.dataMgr updateRemoteData:^(SFSyncState *syncProgressDetails) {
        dispatch_async(dispatch_get_main_queue(), ^{
            weakSelf.navigationItem.rightBarButtonItem.enabled = YES;
            if ([syncProgressDetails isDone]) {
                [weakSelf.dataMgr refreshLocalData];
                [weakSelf showToast:@"Sync complete!"];
                [weakSelf.dataMgr refreshRemoteData];
            } else if ([syncProgressDetails hasFailed]) {
                [weakSelf showToast:@"Sync failed."];
            } else {
                [weakSelf showToast:[NSString stringWithFormat:@"Unexpected status:
%@", [SFSyncState syncStatusToString:syncProgressDetails.status]]];
            }
        });
    }];
```

If the sync up operation succeeded, this block first refreshes the display on the device, along with a "Sync complete!" confirmation toast, and then sends the `refreshRemoteData` message to the `SObjectDataManager`. This final step guarantees that the SmartStore soup remains up-to-date with any recent changes made to Contacts on the server.

Using the Sync Manager with Global SmartStore

To use SmartSync with a global SmartStore instance, call a static factory method on the sync manager object to get a compatible sync manager instance.

Android:

Static Method	Description
`SyncManager getInstance(UserAccount account, String communityId, SmartStore smartStore);`	Returns a sync manager instance that talks to the server as the given community user and writes to or reads from the given SmartStore instance. Use this factory method for syncing data with the global SmartStore instance.

Static Method	Description
`SyncManager getInstance(UserAccount account, String communityId);`	Returns a sync manager instance that talks to the server as the given community user and writes to or reads from the user's default SmartStore instance.
`SyncManager getInstance(UserAccount account);`	Returns a sync manager instance that talks to the server as the given user and writes to or reads from the user's default SmartStore instance.
`SyncManager getInstance();`	Returns a sync manager instance that talks to the server as the current user and writes to or reads from the current user's default SmartStore instance.

iOS:

Static Method	Description
`+ (instancetype)sharedInstanceForUser:(SFUserAccount *)user storeName:(NSString *)storeName;`	Returns a sync manager instance that talks to the server as the given user and writes to or reads from the user's default SmartStore instance.
`+ (instancetype)sharedInstanceForStore:(SFSmartStore *)store;`	Returns a sync manager instance that talks to the server as the current user and writes to or reads from the given SmartStore instance. Use this factory method for syncing data with the global SmartStore instance.
`+ (instancetype)sharedInstance:(SFUserAccount *)user;`	Returns a sync manager instance that talks to the server as the given user and writes to or reads from the user's default SmartStore instance.

Hybrid:

In each of the following method, the optional `isGlobalStore` parameter tells the SmartSync plugin whether to use the global SmartStore instance. If `isGlobalStore` is true, SmartSync writes to and reads from the default global SmartStore instance. If `isGlobalStore` is false, or if the parameter is omitted, SmartSync writes to and reads from the current user's default SmartStore instance.

- `syncDown(isGlobalStore, target, soupName, options, successCB, errorCB);`
- `reSync(isGlobalStore, syncId, successCB, errorCB);`
- `syncUp(isGlobalStore, target, soupName, options, successCB, errorCB);`
- `getSyncStatus(isGlobalStore, syncId, successCB, errorCB);`

Custom Targets

Using Custom Sync Down Targets

During sync down operations, a sync down target controls the set of records to be downloaded and the request endpoint. You can use pre-formatted MRU, SOQL-based, and SOSL-based targets, or you can create custom sync down targets. Custom targets can access arbitrary REST endpoints both inside and outside of Salesforce.

Defining a Custom Sync Down Target

You define custom targets for sync down operations by subclassing your platform's abstract base class for sync down targets. To use custom targets in hybrid apps, implement a custom native target class for each platform you support. The base sync down target classes are:

- **Android:** `SyncDownTarget`
- **iOS:** `SFSyncDownTarget`

Every custom target class must implement the following required methods.

Start Fetch Method

Called by the sync manager to initiate the sync down operation. If `maxTimeStamp` is greater than 0, this operation becomes a "resync". It then returns only the records that have been created or updated since the specified time.

Android:

```
JSONArray startFetch(SyncManager syncManager, long maxTimeStamp);
```

iOS:

```
- (void) startFetch:(SFSmartSyncSyncManager*)syncManager
       maxTimeStamp:(long long)maxTimeStamp
         errorBlock:(SFSyncDownTargetFetchErrorBlock)errorBlock
      completeBlock:(SFSyncDownTargetFetchCompleteBlock)completeBlock;
```

Continue Fetching Method

Called by the sync manager repeatedly until the method returns null. This process retrieves all records that require syncing.

Android:

```
JSONArray continueFetch(SyncManager syncManager);
```

iOS:

```
- (void) continueFetch:(SFSmartSyncSyncManager*)syncManager
            errorBlock:(SFSyncDownTargetFetchErrorBlock)errorBlock
         completeBlock:(SFSyncDownTargetFetchCompleteBlock)completeBlock;
```

modificationDateFieldName Property (Optional)

Optionally, you can override the `modificationDateFieldName` property in your custom class. Mobile SDK uses the field with this name to compute the `maxTimestamp` value that `startFetch` uses to rerun the sync down operation. This operation is also known as *resync*. The default field is `lastModifiedDate`.

Android:

```
String getModificationDateFieldName();
```

iOS:

`modificationDateFieldName` property

idFieldName Property (Optional)

Optionally, you can override the `idFieldName` property in your custom class. Mobile SDK uses the field with this name to get the ID of the record. For example, during sync up, Mobile SDK obtains the ID that it passes to the `updateOnServer()` method from the field whose name matches `idFieldName` in the local record.

Android:

```
String getIdFieldName();
```

iOS:

`idFieldName` property

Invoking the Sync Down Method with a Custom Target

Android:

Pass an instance of your custom `SyncDownTarget` class to the `SyncManager` sync down method:

```
SyncState syncDown(SyncDownTarget target, SyncOptions options, String soupName,
SyncUpdateCallback callback);
```

iOS:

Pass an instance of your custom `SFSyncDownTarget` class to the `SFSmartSyncSyncManager` sync down method:

```
- (SFSyncState*) syncDownWithTarget:(SFSyncDownTarget*)target
                           soupName:(NSString*)soupName
                        updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock;
```

Hybrid:

1. Create a target object with the following property settings:
   - Set `type` to "custom".
   - Set at least one of the following properties:

     Android (if supported):
     Set `androidImpl` to the package-qualified name of your Android custom class.

     iOS (if supported):
     Set `iOSImpl` to the name of your iOS custom class.

   The following example supports both Android and iOS:

   ```
   var target = {type:"custom",
   androidImpl:"com.salesforce.samples.notesync.ContentSoqlSyncDownTarget",
       iOSImpl:"SFContentSoqlSyncDownTarget", … };
   ```

2. Pass this target to the hybrid sync down method:

```
cordova.require("com.salesforce.plugin.SmartSync").syncDown(target, …);
```

Sample Apps

Android

The NoteSync native Android sample app defines and uses the `com.salesforce.samples.notesync.ContentSoqlSyncDownTarget` sync down target.

iOS

The NoteSync native iOS sample app defines and uses the `SFContentSoqlSyncDownTarget` sync down target.

Using Custom Sync Up Targets

During sync up operations, a sync up target controls the set of records to be uploaded and the REST endpoint for updating records on the server. You can access arbitrary REST endpoints—both inside and outside of Salesforce—by creating custom sync up targets.

Defining a Custom Sync Up Target

You define custom targets for sync up operations by subclassing your platform's abstract base class for sync up targets. To use custom targets in hybrid apps, you're required to implement a custom native target class for each platform you support. The base sync up target classes are:

- **Android:** `SyncUpTarget`
- **iOS:** `SFSyncUpTarget`

Every custom target class must implement the following required methods.

Create On Server Method

Sync up a locally created record.

Android:

```
String createOnServer(SyncManager syncManager, String objectType, Map<String,
Object> fields);
```

iOS:

```
- (void) createOnServer:(NSString*)objectType
                 fields:(NSDictionary*)fields
        completionBlock:(SFSyncUpTargetCompleteBlock)completionBlock
              failBlock:(SFSyncUpTargetErrorBlock)failBlock;
```

Update On Server Method

Sync up a locally updated record. For the `objectId` parameter, SmartSync uses the field specified in the `getIdFieldName()` method (Android) or the `idFieldName` property (iOS) of the custom target.

Android:

```
updateOnServer(SyncManager syncManager, String objectType, String objectId,
Map<String, Object> fields);
```

iOS:

```
- (void) updateOnServer:(NSString*)objectType
               objectId:(NSString*)objectId
```

```
                fields:(NSDictionary*)fields
       completionBlock:(SFSyncUpTargetCompleteBlock)completionBlock
             failBlock:(SFSyncUpTargetErrorBlock)failBlock;
```

Delete On Server Method

Sync up a locally deleted record. For the `objectId` parameter, SmartSync uses the field specified in the `getIdFieldName()` method (Android) or the `idFieldName` property (iOS) of the custom target.

Android:

```
deleteOnServer(SyncManager syncManager, String objectType, String objectId);
```

iOS:

```
- (void) deleteOnServer:(NSString*)objectType
               objectId:(NSString*)objectId
        completionBlock:(SFSyncUpTargetCompleteBlock)completionBlock
              failBlock:(SFSyncUpTargetErrorBlock)failBlock;
```

Optional Configuration Changes

Optionally, you can override the following values in your custom class.

getIdsOfRecordsToSyncUp

List of record IDs returned for syncing up. By default, these methods return any record where `__local__` is true.

Android:

```
Set<String> getIdsOfRecordsToSyncUp(SyncManager syncManager, String soupName);
```

iOS:

```
- (NSArray*) getIdsOfRecordsToSyncUp:(SFSmartSyncSyncManager*)syncManager
                            soupName:(NSString*)soupName;
```

Modification Date Field Name

Field used during a `LEAVE_IF_CHANGED` sync up operation to determine whether a record was remotely modified. Default value is `lastModifiedDate`.

Android:

```
String getModificationDateFieldName();
```

iOS:

`modificationDateFieldName` property

Last Modification Date

The last modification date value returned for a record. By default, sync targets fetch the modification date field value for the record.

Android:

```
String fetchLastModifiedDate(SyncManager syncManager, String objectType, String
objectId);
```

iOS:

```
- (void)fetchRecordModificationDates:(NSDictionary *)record

modificationResultBlock:(SFSyncUpRecordModificationResultBlock)modificationResultBlock
```

ID Field Name

Field used to get the ID of the record. For example, during sync up, Mobile SDK obtains the ID that it passes to the `updateOnServer()` method from the field whose name matches `idFieldName` in the local record.

Android:

```
String getIdFieldName();
```

iOS:

`idFieldName` property

Invoking the Sync Up Method with a Custom Target

Android:

On a `SyncManager` instance, call:

```
SyncState syncUp(SyncUpTarget target, SyncOptions options, String soupName,
SyncUpdateCallback callback);
```

iOS:

On a `SFSyncManager` instance, call:

```
- (SFSyncState*) syncUpWithOptions:(SFSyncOptions*)options
                          soupName:(NSString*)soupName
                       updateBlock:(SFSyncSyncManagerUpdateBlock)updateBlock
```

Hybrid:

```
cordova.require("com.salesforce.plugin.smartsync").syncUp(isGlobalStore,
    target, soupName, options, successCB, errorCB);
```

Storing and Retrieving Cached Data

The cache manager provides methods for writing and reading sObject metadata to the SmartSync cache. Each method requires you to provide a key string that identifies the data in the cache. You can use any unique string that helps your app locate the correct cached data.

You also specify the type of cached data. Cache manager methods read and write each of the three categories of sObject data: metadata, MRU (most recently used) list, and layout. Since only your app uses the type identifiers you provide, you can use any unique strings that clearly distinguish these data types.

Cache Manager Classes

- **Android:** `com.salesforce.androidsdk.smartsync.manager.CacheManager`
- **iOS:** `SFSmartSyncCacheManager`

Read and Write Methods

Here are the `CacheManager` methods for reading and writing sObject metadata, MRU lists, and sObject layouts.

- **Android:**

sObjects Metadata

```
public List<SalesforceObject> readObjects(String cacheType, String cacheKey);
public void writeObjects(List<SalesforceObject> objects, String cacheKey, String
cacheType);
```

MRU List

```
public List<SalesforceObjectType> readObjectTypes(String cacheType, String cacheKey);
public void writeObjectTypes(List<SalesforceObjectType> objects,
    String cacheKey, String cacheType);
```

sObject Layouts

```
public List<SalesforceObjectTypeLayout> readObjectLayouts(String cacheType, String
cacheKey);
public void writeObjectLayouts(List<SalesforceObjectTypeLayout> objects,
    String cacheKey, String cacheType);
```

- **iOS:**

  Read Method

  ```
  - (NSArray *)readDataWithCacheType:(NSString *)cacheType
                            cacheKey:(NSString *)cacheKey
                         cachePolicy:(SFDataCachePolicy)cachePolicy
                         objectClass:(Class)objectClass
                          cachedTime:(out NSDate **)lastCachedTime;
  ```

  Write Method

  ```
  - (void)writeDataToCache:(id)data
                 cacheType:(NSString *)cacheType
                  cacheKey:(NSString *)cacheKey;
  ```

Clearing the Cache

When your app is ready to clear the cache, use the following cache manager methods:

- **Android:**

  ```
  public void removeCache(String cacheType, String cacheKey);
  ```

- **iOS:**

  ```
  - (void)removeCache:(NSString *)cacheType cacheKey:(NSString *)cacheKey;
  ```

These methods let you clear a selected portion of the cache. To clear the entire cache, call the method for each cache key and data type you've stored.

Hybrid

Using SmartSync in Hybrid Apps

The SmartSync Data Framework for hybrid apps 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.

Mobile SDK provides two options for using SmartSync in hybrid apps.

- `com.salesforce.plugin.smartsync`: The SmartSync plugin offers basic "sync up" and "sync down" functionality. This plugin exposes part of the native SmartSync library. For simple syncing tasks, you can use the plugin to sync records rapidly in a native thread, rather than in the web view.
- `smartsync.js`: The SmartSync JavaScript library provides a Force.SObject data framework for more complex syncing operations. This library is based on `backbone.js`, an open-source JavaScript framework that defines an extensible data modeling mechanism. To understand this technology, browse the examples and documentation at backbonejs.org.

A set of sample hybrid applications demonstrate how to use SmartSync. Sample apps in the `hybrid/SampleApps/AccountEditor/assets/www` folder demonstrate how to use the Force.SObject library in `smartsync.js`:

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

The sample app in the `hybrid/SampleApps/SimpleSync` folder demonstrates how to use the SmartSync plugin.

Should I Use Smartsync.js or the SmartSync Plugin?

`Smartsync.js`—the JavaScript version of SmartSync—and native SmartSync—available to hybrid apps through a Cordova plugin—share a name, but they offer different advantages.

`smartsync.js` is built on backbone.js and gives you easy-to-use model objects to represent single records or collections of records. It also provides convenient fetch, save, and delete methods. However, it doesn't give you true sync down and sync up functionality. Fetching records with an SObjectCollection is similar to the plugin's `syncDown` method, but it deposits all the retrieved objects in memory. For that reason, it's not the best choice for moving large datasets. Furthermore, you're required to implement the sync up functionality yourself. The AccountEditor sample app demonstrates a typical JavaScript `syncUp()` implementation.

Native SmartSync doesn't return model objects, but it provides robust `syncUp` and `syncDown` methods for moving large data sets to and from the server.

You can also use the two libraries together. For example, you can set up a `Force.StoreCache` with `smartsync.js`, sync data into it using the SmartSync plugin, and then call fetch or save using `smartsync.js`. You can then sync up from the same cache using the SmartSync plugin, and it all works.

Both libraries provide the means to define your own custom endpoints, so which do you choose? The following guidelines can help you decide:

- Use custom endpoints from `smartsync.js` if you want to talk to the server directly for saving or fetching data with JavaScript.
- If you talk only to SmartStore and get data into SmartStore using the SmartSync plugin, then you don't need the custom endpoints in `smartsync.js`. However, you must define native custom targets.

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 Plugin

The SmartSync plugin provides JavaScript access to the native SmartSync library's "sync up" and "sync down" functionality. As a result, performance-intensive operations—network negotiations, parsing, SmartStore management—run on native threads that do not affect web view operations.

Adding the SmartSync plugin to your hybrid project is a function of the Mobile SDK npm scripts:

- For forceios version 3.0, the plugin is automatically included.
- For forcedroid version 3.0, answer "yes" when asked if you want to use SmartStore.

If you're adding the SmartSync plugin to an existing hybrid app, it's best to re-create the app using version 3.0 of forcedroid or forceios. When the new app is ready, copy your custom HTML, CSS, and JavaScript files from your old project into the new project.

SmartSync Plugin Methods

The SmartSync plugin exposes two methods: `syncDown()` and `syncUp()`. When you use these methods, several important guidelines can make your life simpler:

- To create, update, or delete records locally for syncing with the plugin, use `Force.SObject` from `smartsync.js`. SmartSync expects some special fields on soup records that `smartsync.js` creates for you.
- Similarly, to create the soup that you'll use in your sync operations, use `Force.StoreCache` from `smartsync.js`.

- If you've changed objects in the soup, always call `syncUp()` before calling `syncDown()`.

syncDown() Method

Downloads the sObjects specified by `target` into the SmartStore soup specified by `soupName`. If sObjects in the soup have the same ID as objects specified in the target, SmartSync overwrites the duplicate objects in the soup.

Syntax

```
cordova.require("com.salesforce.plugin.smartsync").syncDown(
    [isGlobalStore, ]target, soupName, options, callback);
```

Parameters

isGlobalStore
(Optional) Pass true to sync the data into a global SmartStore soup.

`target`
Indicates which sObjects to download to the soup. Can be any of the following strings:

- `{type:"soql", query:"<soql query>"}`

  Downloads the sObjects returned by the given SOQL query.

- `{type:"sosl", query:"<sosl query>"}`

  Downloads the sObjects returned by the given SOSL query.

- `{type:"mru", sobjectType:"<sobject type>", fieldlist:"<fields to fetch>"}`

  Downloads the specified fields of the most recently used sObjects of the specified sObject type.

- `{type:"custom", androidImpl:"<name of native Android target class (if supported)>", iOSImpl:"<name of native iOS target class (if supported)>"}`

  Downloads the records specified by the given custom targets. If you use custom targets, provide either `androidImpl` or `iOSImpl`, or, preferably, both. See Using Custom Sync Down Targets.

`soupName`
Name of soup that receives the downloaded sObjects.

`options`
Use one of the following values:

- To overwrite local records that have been modified, pass `{mergeMode:Force.MERGE_MODE_DOWNLOAD.OVERWRITE}`.
- To preserve local records that have been modified, pass `{mergeMode:Force.MERGE_MODE_DOWNLOAD.LEAVE_IF_CHANGED}`. With this value, locally modified records are not overwritten.

`callback`
Function called once the sync has started. This function is called multiple times during a sync operation:

1. When the sync operation begins
2. When the internal REST request has completed
3. After each page of results is downloaded, until 100% of results have been received

Status updates on the sync operation arrive via browser events. To listen for these updates, use the following code:

```
document.addEventListener("sync",
    function(event) {
        // event.detail contains the status of the sync operation
    }
);
```

The `event.detail` member contains a map with the following fields:

- `syncId`: ID for this sync operation
- `type`: "syncDown"
- `target`: Targets you provided
- `soupName`: Soup name you provided
- `options`: "{}"
- `status`: Sync status, which can be "NEW", "RUNNING", "DONE" or "FAILED"
- `progress`: Percent of total records downloaded so far (integer, 0–100)
- `totalSize`: Number of records downloaded so far

syncUp() Method

Uploads created, deleted, or updated records in the SmartStore soup specified by `soupName` and updates, creates, or deletes the corresponding records on the Salesforce server. Updates are reported through browser events.

Syntax

```
cordova.require("com.salesforce.plugin.smartsync").syncUp(isGlobalStore, target, soupName,
 options, callback);
```

Parameters

`isGlobalStore`
Indicates whether you are using global SmartStore. Defaults to `false` if not specified

`target`
Name of one or more custom target native classes, if you define custom targets. See Using Custom Sync Up Targets.

`soupName`
Name of soup from which to upload sObjects.

`options`
A map with the following keys:

- `fieldlist`: List of fields sent to the server.
- `mergeMode`:
  - To overwrite remote records that have been modified, pass "OVERWRITE".
  - To preserve remote records that have been modified, pass "LEAVE_IF_CHANGED". With this value, modified records on the server are not overwritten.
  - Defaults to "OVERWRITE" if not specified.

`callback`
Function called multiple times after the sync has started. During the sync operation, this function is called for these events:

1. When the sync operation begins

2. When the internal REST request has completed
3. After each page of results is uploaded, until 100% of results have been received

Status updates on the sync operation arrive via browser events. To listen for these updates, use the following code:

```
document.addEventListener("sync",
    function(event) {
        // event.detail contains the status of the sync operation
    }
);
```

The `event.detail` member contains a map with the following fields:

- `syncId`: ID for this sync operation
- `type`: "syncUp"
- `target`: "{}" or a map or dictionary containing the class names of Android and iOS custom target classes you've implemented
- `soupName`: Soup name you provided
- `options`:
  - `fieldlist`: List of fields sent to the server
  - `mergeMode`: "OVERWRITE" or "LEAVE_IF_CHANGED"
- `status`: Sync status, which can be "NEW", "RUNNING", "DONE" or "FAILED"
- `progress`: Percent of total records downloaded so far (integer, 0–100)
- `totalSize`: Number of records downloaded so far

Using the SmartSync Data Framework in JavaScript

To use SmartSync in a hybrid app, import these files with <script> tags:

- `jquery-x.x.x.min.js` (use the version in the `dependencies/jquery/` directory of the SalesforceMobileSDK-Shared repository)
- `underscore-x.x.x.min.js` (use the version in the `dependencies/underscore/` directory of the SalesforceMobileSDK-Shared repository)
- `backbone-x.x.x.min.js` (use the version in the `dependencies/backbone/` directory of the SalesforceMobileSDK-Shared repository)
- `cordova.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.

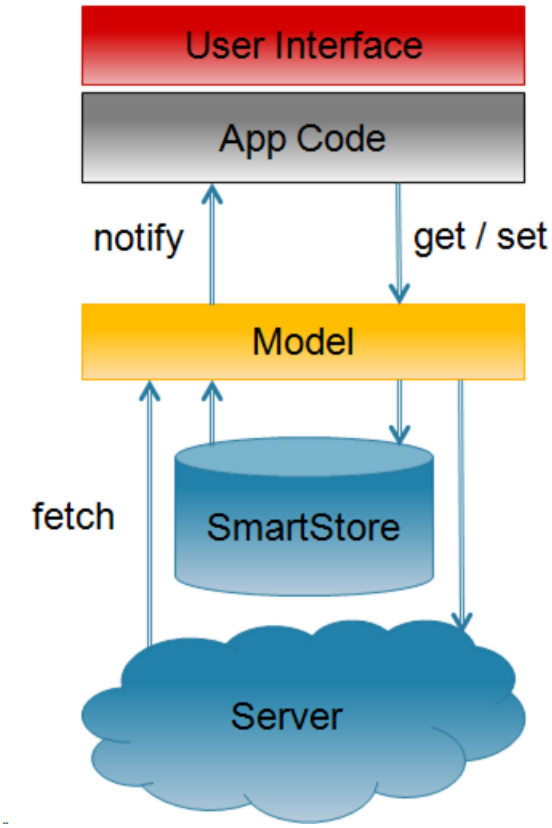
Default Cache and Custom Cache Implementations

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.

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.

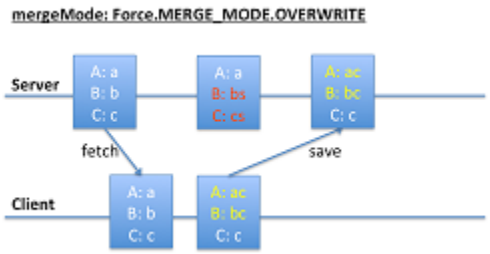
If a save or delete operation fails, you receive a report object with the following fields:

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.

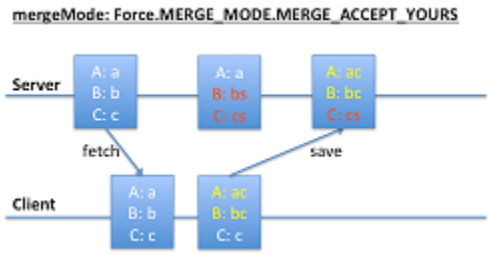
MERGE_MODE.OVERWRITE

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.



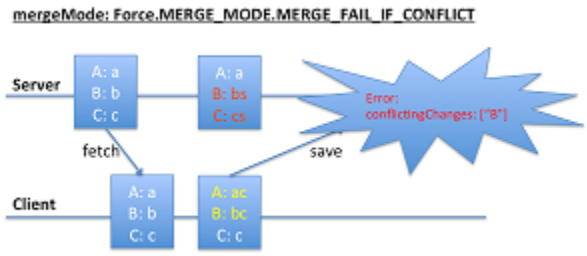
MERGE_ACCEPT_YOURS

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.



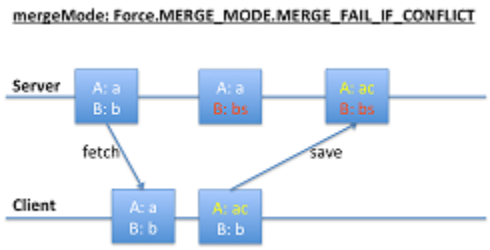
MERGE_FAIL_IF_CONFLICT (Fails)

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.

Accessing Custom API Endpoints

SmartSync enables connections to any REST API endpoint. You can perform basic operations on sObjects with the Force.com API. You can use Apex REST objects, Chatter Files, and any other Salesforce REST API. You can also access custom REST endpoints outside of Salesforce.

`Force.RemoteObject` Class

To support arbitrary REST calls, SmartSync introduces the `Force.RemoteObject` abstract class. `Force.RemoteObject` serves as a layer of abstraction between `Force.SObject` and `Backbone.Model`. Instead of directly subclassing `Backbone.Model`, `Force.SObject` now subclasses `Force.RemoteObject`, which in turn subclasses `Backbone.Model`. `Force.RemoteObject` does everything `Force.SObject` formerly did except communicate with the server.

Calling Custom Endpoints with `syncRemoteObjectWithServer()`

The `RemoteObject.syncRemoteObjectWithServer()` prototype method handles server interactions. `Force.SObject` implements `syncRemoteObjectWithServer()` to use the Force.com REST API. If you want to use other server end points, create a subclass of `Force.RemoteObject` and implement `syncRemoteObjectWithServer()`. This method is called when you call `fetch()` on an object of your subclass, if the object is currently configured to fetch from the server.

Example: **Example**

The HybridFileExplorer sample application is a SmartSync app that shows how to use `Force.RemoteObject`. HybridFileExplorer calls the Chatter REST API to manipulate files. It defines an `app.models.File` object that extends `Force.RemoteObject`. In its implementation of `syncRemoteObjectWithServer()`, `app.models.File` calls `forcetk.fileDetails()`, which wraps the `/chatter/files/fileId` REST API.

```
app.models.File = Force.RemoteObject.extend({
    syncRemoteObjectWithServer: function(method, id) {
        if (method != "read")
            throw "Method not supported " + method;
        return Force.forcetkClient.fileDetails(id, null);
    }
})
```

Force.RemoteObjectCollection Class

To support collections of fetched objects, SmartSync introduces the `Force.RemoteObjectCollection` abstract class. This class serves as a layer of abstraction between `Force.SObjectCollection` and Backbone.Collection. Instead of directly subclassing Backbone.Collection, Force.SObjectCollection now subclasses Force.RemoteObjectCollection, which in turn subclasses `Backbone.Collection`. `Force.RemoteObjectCollection` does everything `Force.SObjectCollection` formerly did except communicate with the server.

Implementing Custom Endpoints with fetchRemoteObjectFromServer()

The `RemoteObject.fetchRemoteObjectFromServer()` prototype method handles server interactions. This method uses the Force.com REST API to run SOQL/SOSL and MRU queries. If you want to use arbitrary server end points, create a subclass of `Force.RemoteObjectCollection` and implement `fetchRemoteObjectFromServer()`. This method is called when you call `fetch()` on an object of your subclass, if the object is currently configured to fetch from the server.

When the `app.models.FileCollection.fetchRemoteObjectsFromServer()` function returns, it promises an object containing valuable information and useful functions that use metadata from the response. This object includes:

- `totalSize`: The number of files in the returned collection
- `records`: The collection of returned files
- `hasMore`: A function that returns a boolean value that indicates whether you can retrieve another page of results
- `getMore`: A function that retrieves the next page of results (if `hasMore()` returns true)
- `closeCursor`: A function that indicates that you're finished iterating through the collection

These functions leverage information contained in the server response, including `Files.length` and `nextPageUrl`.

Example: **Example**

The HybridFileExplorer sample application also demonstrates how to use `Force.RemoteObjectCollection`. This example calls the Chatter REST API to iterate over a list of files. It supports three REST operations: `ownedFilesList`, `filesInUsersGroups`, and `filesSharedWithUser`.

You can write functions such as `hasMore()` and `getMore()`, shown in this example, to navigate through pages of results. However, since apps don't call `fetchRemoteObjectsFromServer()` directly, you capture the returned promise object when you call `fetch()` on your collection object.

```
app.models.FileCollection = Force.RemoteObjectCollection.extend({
    model: app.models.File,

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

    fetchRemoteObjectsFromServer: function(config) {
        var fetchPromise;
        switch(config.type) {
            case "ownedFilesList": fetchPromise =
                Force.forcetkClient.ownedFilesList("me", 0);
                break;
            case "filesInUsersGroups": fetchPromise =
                Force.forcetkClient.filesInUsersGroups("me", 0);
                break;
            case "filesSharedWithUser": fetchPromise =
                Force.forcetkClient.filesSharedWithUser("me", 0);
                break;
        };

        return fetchPromise
            .then(function(resp) {
                var nextPageUrl = resp.nextPageUrl;
                return {
                    totalSize: resp.files.length,
                    records: resp.files,
                    hasMore: function() {
                        return nextPageUrl != null; },
                    getMore: function() {
                        var that = this;
                        if (!nextPageUrl)
                            return null;
                        return
                            forcetkClient.queryMore(nextPageUrl)
                            .then(function(resp) {
                                nextPageUrl = resp.nextPageUrl;
                                that.records.
                                    pushObjects(resp.files);
                                return resp.files;
                            });
                    },
                    closeCursor: function() {
                        return $.when(function() {
                                        nextPageUrl = null;
                                    });
                    }
                };
            });
```

```
    }
});
```

Using Apex REST Resources

To support Apex REST resources, Mobile SDK provides two classes: `Force.ApexRestObject` and `Force.ApexRestObjectCollection`. These classes subclass `Force.RemoteObject` and `Force.RemoteObjectCollection`, respectively, and can talk to a REST API that you have created using Apex REST.

`Force.ApexRestObject`

`Force.ApexRestObject` is similar to `Force.SObject`. Instead of an `sobjectType`, `Force.ApexRestObject` requires the Apex REST resource path relative to `services/apexrest`. For example, if your full resource path is `services/apexrest/simpleAccount/*`, you specify only `/simpleAccount/*`. `Force.ApexRestObject` also expects you to specify the name of your ID field if it's different from "Id".

Example: **Example**

Let's assume you've created an Apex REST resource called "simple account," which is just an account with two fields: `accountId` and `accountName`.

```
@RestResource(urlMapping='/simpleAccount/*')
  global with sharing class SimpleAccountResource {
      static String getIdFromURI() {
          RestRequest req = RestContext.request;
          return req.requestURI.substring(req.requestURI.lastIndexOf('/')+1);
      }

      @HttpGet global static Map&lt;String, String&gt; doGet() {
          String id = getIdFromURI();
          Account acc = [select Id, Name from Account
                         where Id = :id];
          return new Map&lt;String, String&gt;{
              'accountId'=>acc.Id, 'accountName'=>acc.Name};
      }

      @HttpPost global static Map&lt;String, String&gt;
          doPost(String accountName) {
              Account acc = new Account(Name=accountName);
              insert acc;
              return new Map&lt;String, String&gt;{
                  'accountId'=>acc.Id, 'accountName'=>acc.Name};
      }

      @HttpPatch global static Map&lt;String, String&gt;
          doPatch(String accountName) {
              String id = getIdFromURI();
              Account acc = [select Id from Account
                             where Id = :id];
              acc.Name = accountName;
              update acc;
              return new Map&lt;String, String&gt;{
                  'accountId'=>acc.Id, 'accountName'=>acc.Name};
```

```
    }

    @HttpDelete global static void doDelete() {
        String id = getIdFromURI();
        Account acc = [select Id from Account where Id = :id];
        delete acc;
        RestContext.response.statusCode = 204;
    }
}
```

With SmartSync, you do the following to create a "simple account".

```
var SimpleAccount = Force.ApexRestObject.extend(
    {apexRestPath:"/simpleAccount",
      idAttribute:"accountId",
        fieldlist:["accountId", "accountName"]});
var acc = new SimpleAccount({accountName:"MyFirstAccount"});
acc.save();
```

You can update that "simple account".

```
acc.set("accountName", "MyFirstAccountUpdated");
acc.save(null, {fieldlist:["accountName"]);
// our apex patch endpoint only expects accountName
```

You can fetch another "simple account".

```
var acc2 = new SimpleAccount({accountId:"&lt;valid id&gt;"})
acc.fetch();
```

You can delete a "simple account".

```
acc.destroy();
```

Note: In SmartSync calls such as `fetch()`, `save()`, and `destroy()`, you typically pass an options parameter that defines success and error callback functions. For example:

```
acc.destroy({success:function(){alert("delete succeeded");}});
```

`Force.ApexRestObjectCollection`

`Force.ApexRestObjectCollection` is similar to `Force.SObjectCollection`. The config you specify for fetching doesn't support SOQL, SOSL, or MRU. Instead, it expects the Apex REST resource path, relative to `services/apexrest`. For example, if your full resource path is `services/apexrest/simpleAccount/*`, you specify only `/simpleAccount/*`.

You can also pass parameters for the query string if your endpoint supports them. The Apex REST endpoint is expected to return a response in this format:

```
{   totalSize: <number of records returned>
   records: <all fetched records>
   nextRecordsUrl: <url to get next records or null>
}
```

Example: **Example**

Let's assume you've created an Apex REST resource called "simple accounts". It returns "simple accounts" that match a given name.

```
@RestResource(urlMapping='/simpleAccounts/*')
global with sharing class SimpleAccountsResource {
    @HttpGet global static SimpleAccountsList doGet() {
        String namePattern =
            RestContext.request.params.get('namePattern');
        List<SimpleAccount> records = new List<SimpleAccount>();
        for (SObject sobj : Database.query(
            'select Id, Name from Account
             where Name like \'' + namePattern + '\'')) {
                 Account acc = (Account) sobj;
                records.add(new SimpleAccount(acc.Id, acc.Name));
        }
        return new SimpleAccountsList(records.size(), records);
    }

    global class SimpleAccountsList {
        global Integer totalSize;
        global List<SimpleAccount> records;

        global SimpleAccountsList(Integer totalSize,
            List<SimpleAccount> records) {
                this.totalSize = totalSize;
                this.records = records;
        }
    }

    global class SimpleAccount {
        global String accountId;
        global String accountName;

        global SimpleAccount(String accountId, String accountName)
        {
            this.accountId = accountId;
            this.accountName = accountName;
        }
    }
}
```

With SmartSync, you do the following to fetch a list of "simple account" records.

```
var SimpleAccountCollection =
    Force.ApexRestObjectCollection.extend(
        {model: SimpleAccount,
         config:{
                 apexRestPath:"/simpleAccounts",
                 params:{namePattern:"My%"}
             }
         }
     );
var accs = new SimpleAccountCollection();
accs.fetch();
```

Note: In SmartSync calls such as `fetch()`, you typically pass an options parameter that defines success and error callback functions. For example:

```
acc.fetch({success:function(){alert("fetched " +
    accs.models.length + " simple accounts");}});
```

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 iOS Installation. For Android instructions, see Android Installation.

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. Copy the files from `samples/usersearch` directory of the https://github.com/forcedotcom/SalesforceMobileSDK-Shared/ into the `www/` folder.
5. 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.js"></script>
<script src="forcetk.mobilesdk.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("com.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("com.salesforce.plugin.oauth").forcetkRefresh);
   app.router = new app.Router();
   Backbone.history.start();
}
```

Example: 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.js"></script>
    <script src="forcetk.mobilesdk.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("com.salesforce.plugin.oauth").
        getAuthCredentials(appStart);
  }

  function appStart(creds) {
      console.log(JSON.stringify(creds));
      Force.init(creds, null, null,
          cordova.require("com.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
});
```

Example: 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.

Example: 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;
}
```

Example: 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();
}
```

Example: 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.

Example: 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.js"></script>
    <script src="forcetk.mobilesdk.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("com.salesforce.plugin.oauth").
    getAuthCredentials(appStart);
}

function appStart(creds) {
    console.log(JSON.stringify(creds));
    Force.init(creds, null, null,
       cordova.require("com.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 Project Navigator, select the `Hybrid SDK/AccountEditor` project and click **Run**.

Running the Samples in Android

To run the sample in an Eclipse workspace, import the following projects from your clone of the SalesforceMobileSDK-Android repository:

- `libs/SalesforceSDK`
- `libs/SmartStore`
- `hybrid/SampleApps/AccountEditor`

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

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 build and run the sample, refer to the instructions at Build Hybrid Sample Apps on page 128.

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.js`—Required for all hybrid application used the SalesforceMobileSDK.
- `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 build and run the sample, refer to the instructions at Build Hybrid Sample Apps on page 128.

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.js`—Required for all hybrid application used the SalesforceMobileSDK.
- `forcetk.mobilesdk.js`—Force.com JavaScript library for making Rest API calls. Required by SmartSync.
- `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 original 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.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.
- `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

Event Type	Target Control	Handler function name
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 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 Files and Networking

In this chapter ...

Mobile SDK 2.1 introduces an API for files and networking. This API includes two levels of technology. For file management, the SDK provides a set of convenience methods that wraps the file requests in the Chatter REST API. Under the REST API wrapper level, a networking layer exposes objects that let the app control pending REST requests. Together, these two sides of the same coin give the SDK a more robust feature set as well as enhanced networking performance.

Architecture

Beginning with Mobile SDK 2.1, the Android REST request system uses Google Volley, an open-source external library, as its underlying architecture. This architecture allows you to access the Volley `QueueManager` object to manage requests. At runtime, you can use the `QueueManager` to cancel pending requests on asynchronous threads. You can learn about Volley at https://developers.google.com/events/io/sessions/325304728

In iOS, file management and networking rely on the `SalesforceNetwork` library. All REST API call—for files and any other REST requests—go through this library.

Note: If you directly accessed a third-party networking library in older versions of your app, update that code to use the `SalesforceNetwork` library.

Hybrid JavaScript functions use the architecture of the Mobile SDK for the device operating system (Android or iOS) to implement file operations. These functions are defined in `forcetk.mobilesdk.js`.

Downloading Files and Managing Sharing

Salesforce Mobile SDK provides convenience methods that build specialized REST requests for file download and sharing operations. You can use these requests to:

- Access the byte stream of a file.
- Download a page of a file.
- Preview a page of a file.
- Retrieve details of File records.
- Access file sharing information.
- Add and remove file shares.

Pages in Requests

The term "page" in REST requests can refer to either a specific item or a group of items in the result set, depending on the context. When you preview a page of a specific file, for example, the request retrieves the specified page from the rendered pages. For most other requests, a page refers to a section of the list of results. The maximum number of records or topics in a page defaults to 25.

The response includes a `NextPageUrl` field. If this value is defined, there is another page of results. If you want your app to scroll through pages of results, you can use this field to avoid sending unnecessary requests. You can also detect when you're at the end of the list by simply checking the response status. If nothing or an error is returned, there's nothing more to display and no need to issue another request.

Uploading Files

Native mobile platforms support a method for uploading a file. You provide a path to the local file to be uploaded, the name or title of the file, and a description. If you know the MIME type, you can specify that as well. The upload method returns a platform-specific request object that can upload the file to the server. When you send this request to the server, the server creates a file with version set to 1.

Use the following methods for the given app type:

App Type	Upload Method	Signature
Android native	`FileRequests.uploadFile()`	`public static RestRequest uploadFile( File theFile, String name, String description, String mimeType) throws UnsupportedEncodingException`
iOS native	`- requestForUploadFile: name:description:mimeType:`	`- (SFRestRequest *) requestForUploadFile:(NSData *)data name:(NSString *)name description:(NSString *)description mimeType:(NSString *)mimeType`
Hybrid (Android and iOS)	N/A	N/A

Encryption and Caching

Mobile SDK gives you access to the file's unencrypted byte stream but doesn't implement file caching or storage. You're free to devise your own solution if your app needs to store files on the device.

Using Files in Android Apps

The `FileRequests` class provides static methods for creating `RestRequest` objects that perform file operations. Each method returns the new `RestRequest` object. Applications then call the `ownedFilesList()` method to retrieve a `RestRequest` object. It passes this object as a parameter to a function that uses the `RestRequest` object to send requests to the server:

```
performRequest(FileRequests.ownedFilesList(null, null));
```

This example passes null to the first parameter (`userId`). This value tells the `ownedFilesList()` method to use the ID of the context, or logged-in, user. The second null, for the `pageNum` parameter, tells the method to fetch the first page of results.

For native Android apps, file management classes and methods live in the `com.salesforce.androidsdk.rest.files` package.

SEE ALSO:

FileRequests Methods (Android)

Managing the Request Queue

The `RestClient` class internally uses an instance of the Volley `RequestQueue` class to manage REST API requests. You can access the underlying `RequestQueue` object by calling `restClient.getRequestQueue()` on your `RestClient` instance. With the `RequestQueue` object you can directly cancel and otherwise manipulate pending requests.

Example: **Example: Canceling All Pending Requests**

The following code calls `getRequestQueue()` on an instance of `RestClient` (`client`). It then calls the `RequestQueue.cancelAll()` method to cancel all pending requests in the queue. The `cancelAll()` method accepts a `RequestFilter` parameter, so the code passes in an object of a custom class, `CountingFilter`, which implements the Volley `RequestFilter` interface.

```
  CountingFilter countingFilter = new CountingFilter();
  client.getRequestQueue().cancelAll(countingFilter);
  int count = countingFilter.getCancelCount();
  ...

 /**
  * Request filter that cancels all requests and also counts the number of requests
canceled
  *
  */
 class CountingFilter implements RequestFilter {

  private int count = 0;

  public int getCancelCount() {
   return count;
  }

  @Override
  public boolean apply(Request<?> request) {
   count++;
   return true;
  }
 }
```

`RequestQueue.cancelAll()` lets the `RequestFilter`-based object inspect each item in the queue before allowing the operation to continue. Internally, `cancelAll()` calls the filter's `apply()` method on each iteration. If `apply()` returns true, the cancel operation continues. If it returns false, `cancelAll()` does not cancel that request and continues to the next request in the queue.

In this code example, the `CountingFilter.apply()` merely increments an internal counter on each call. After the `cancelAll()` operation finishes, the sample code calls `CountingFilter.getCancelCount()` to report the number of canceled objects.

Using Files in iOS Native Apps

To handle files in native iOS apps, use convenience methods defined in the `SFRestAPI (Files)` category. These methods parallel the files API for Android native and hybrid apps. They send requests to the same list of REST APIs, but use different underpinnings.

iOS Project Settings

If your app is based on Mobile SDK 3.1 or earlier, adjust your project settings for all targets to include the `SalesforceNetwork` library.

1. Download the `SalesforceNetwork` library bundled with Mobile SDK 2.1. Get the binary libraries and their headers from compressed files at https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Distribution.
2. Link the following modules in your project:
   - `libMKNetworkKit-iOS.a`
   - `libSalesforceNetworkSDK.a`
   - `ImageIO.framework`

   The Files API also requires the following frameworks which are normally linked by default:
   - `CFNetwork.framework`
   - `SystemConfiguration.framework`
   - `Security.framework`

REST Responses and Multithreading

The networking library always dispatches REST responses to the thread where your `SFRestDelegate` currently runs. This design accommodates your app no matter how your delegate intends to handle the server response. When you receive the response, you can do whatever you like with the returned data. For example, you can cache it, store it in a database, or immediately blast it to UI controls. If you send the response directly to the UI, however, remember that your delegate must dispatch its messages to the main thread.

SEE ALSO:

SFRestAPI (Files) Category—Request Methods (iOS)

Managing Requests

The `SalesforceNetwork` library for iOS defines two primary objects, `SFNetworkEngine` and `SFNetworkOperation`. `SFRestRequest` internally uses a `SFNetworkOperation` object to make each server call.

If you'd like to access the `SFNetworkOperation` object for any request, you have two options.

- The following methods return `SFNetworkOperation*`:
  - `[SFRestRequest send:]`
  - `[SFRestAPI send:delegate:]`
- `SFRestRequest` objects include a `networkOperation` object of type `SFNetworkOperation*`.

To cancel pending REST requests, you also have two options.

- `SFRestRequest` provides a new method that cancels the request:

```
- (void) cancel;
```

- And `SFRestAPI` has a method that cancels all requests currently running:

```
- (void)cancelAllRequests;
```

Example: **Examples of Canceling Requests**

To cancel all requests:

```
[[SFRestAPI sharedInstance] cancelAllRequests];
```

To cancel a single request:

```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForOwnedFilesList:nil
page:0];
[[SFRestAPI sharedInstance] send:request delegate:self];
...
// User taps Cancel Request button while waiting for the response
-(void) cancelRequest:(SFRestRequest *) request {
   [request cancel];
}
```

Using Files in Hybrid Apps

Except for uploading, you can use the same file requests in hybrid apps as in native apps. Hybrid file request wrappers reside in the `forcetk.mobilesdk.js` JavaScript library. When using the hybrid functions, you pass in a callback function that receives and handles the server response. You also pass in a function to handle errors.

To simplify the code, you can leverage the `smartsync.js` and `forcetk.mobilesdk.js` libraries to build your HTML app. The HybridFileExplorer sample app demonstrates this.

Note: Mobile SDK does not support file uploads in hybrid apps.

SEE ALSO:

Files Methods For Hybrid Apps

CHAPTER 8 Push Notifications and Mobile SDK

In this chapter ...

Push notifications from Salesforce help your mobile users stay on top of important developments in their organizations. The Salesforce Mobile Push Notification Service, which becomes generally available in Summer '14, lets you configure and test mobile push notifications before you implement any code. To receive mobile notifications in a production environment, your Mobile SDK app implements the mobile OS provider's registration protocol and then handles the incoming notifications. Mobile SDK minimizes your coding effort by implementing most of the registration tasks internally.

About Push Notifications

With the Salesforce Mobile Push Notification Service, you can develop and test push notifications in native and hybrid mobile apps. Salesforce Mobile SDK provides APIs that you can implement to register devices with the push notification service. However, receiving and handling the notifications remain the responsibility of the developer.

Push notification setup occurs on several levels:

- Configuring push services from the device technology provider (Apple for iOS, Google for Android)
- Configuring your Salesforce connected app definition to enable push notifications
- Implementing Apex triggers

  OR

  Calling the push notification resource of the Chatter REST API
- Modifying code in your Mobile SDK app
- Registering the mobile device at runtime

You're responsible for Apple or Google service configuration, connected app configuration, Apex or Chatter REST API coding, and minor changes to your Mobile SDK app. Salesforce Mobile SDK handles runtime registration transparently.

For a full description of how to set up mobile push notifications for your organization, see the Salesforce Mobile Push Notifications Implementation Guide.

Using Push Notifications in Hybrid Apps

To use push notifications in a hybrid app, first be sure to

- Register for push notifications with the OS provider.
- Configure your connected app to support push notifications for your target device platform.

Salesforce Mobile SDK lets your hybrid app register itself to receive notifications, and then you define the behavior that handles incoming notifications.

SEE ALSO:

Using Push Notifications in Android

Using Push Notifications in iOS

Code Modifications (Hybrid)

1. In your callback for `cordova.require("com.salesforce.plugin.oauth").getAuthCredentials()`, add the following code:

```
cordova.require("com.salesforce.util.push").registerPushNotificationHandler(
    function(message) {
        // add code to handle notifications
    },
    function(error) {
        // add code to handle errors
    }
);
```

Example: This code demonstrates how you might handle messages. The server delivers the payload in `message["payload"]`.

```
function(message) {
    var payload = message["payload"];
    if (message["foreground"]) {
        // Notification is received while the app is in
        // the foreground
        // Do something appropriate with payload
    }
    if (!message["foreground"]) {
        // Notification was received while the app was in
        // the background, and the notification was clicked,
        // bringing the app to the foreground
        // Do something appropriate with payload
    }
}
```

Using Push Notifications in Android

Salesforce sends push notifications to Android apps through the Google Cloud Messaging for Android (GCM) framework. See http://developer.android.com/google/gcm/index.html for an overview of this framework.

When developing an Android app that supports push notifications, remember these key points:

- You must be a member of the Android Developer Program.
- You can test GCM push services only on an Android device with either the Android Market app or Google Play Services installed. Push notifications don't work on an Android emulator.
- You can also use the Send Test Notification link in your connected app detail view to perform a "dry run" test of your GCM setup without pinging any device.

To begin, create a Google API project for your app. Your project must have the GCM for Android feature enabled. See http://developer.android.com/google/gcm/gs.html for instructions on setting up your project.

The setup process for your Google API project creates a key for your app. Once you've finished the project configuration, you'll need to add the GCM key to your connected app settings.

Note: Push notification registration occurs at the end of the OAuth login flow. Therefore, an app does not receive push notifications unless and until the user logs into a Salesforce organization.

Configure a Connected App For GCM (Android)

To configure your Salesforce connected app to support push notifications:

1. In your Salesforce organization, go to **Setup** > **Create** > **Apps**.
2. In Connected Apps, click **Edit** next to an existing connected app, or **New** to create a new connected app.

   If you're creating a new connected app, see Create a Connected App.
3. Under Mobile App Settings, select **Push Messaging Enabled**.
4. For Supported Push Platform, select **Android GCM**.
5. For Key for Server Applications (API Key), enter the key you obtained during the developer registration with Google.

6. Click **Save**.

   Note: After saving a new connected app, you'll get a consumer key. Mobile apps use this key as their connection token.

Code Modifications (Android)

To configure your Mobile SDK app to support push notifications:

1. Add an entry for `androidPushNotificationClientId`.
   - In `res/values/bootconfig.xml` (for native apps):

     ```
     <string name="androidPushNotificationClientId">35123627573</string>
     ```

   - In `assets/www/bootconfig.json` (for hybrid apps):

     ```
     "androidPushNotificationClientId": "35123627573"
     ```

   This value represents the project number of the Google project that is authorized to send push notifications to an Android device.

   Behind the scenes, Mobile SDK automatically reads this value and uses it to register the device against the Salesforce connected app. This validation allows Salesforce to send notifications to the connected app. At logout, Mobile SDK also automatically unregisters the device for push notifications.

2. Create a class in your app that implements `PushNotificationInterface`. `PushNotificationInterface` is a Mobile SDK Android interface for handling push notifications. `PushNotificationInterface` has a single method, `onPushMessageReceived(Bundle message)`:

   ```
   public interface PushNotificationInterface {
       public void onPushMessageReceived(Bundle message);
   }
   ```

   In this method you implement your custom functionality for displaying, or otherwise disposing of, push notifications.

3. In the `onCreate()` method of your `Application` subclass, call the `SalesforceSDKManager.setPushNotificationReceiver()` method, passing in your implementation of `PushNotificationInterface`. Call this method immediately after the `SalesforceSDKManager.initNative()` call. For example:

   ```
   @Override
   public void onCreate() {
      super.onCreate();
      SalesforceSDKManager.initNative(getApplicationContext(),
         new KeyImpl(), MainActivity.class);
      SalesforceSDKManager.getInstance().
         setPushNotificationReceiver(myPushNotificationInterface);
   }
   ```

Using Push Notifications in iOS

When developing an iOS app that supports push notifications, remember these key points:

- You must be a member of the iOS Developer Program.
- You can test Apple push services only on an iOS physical device. Push notifications don't work in the iOS simulator.
- There are no guarantees that all push notifications will reach the target device, even if the notification is accepted by Apple.
- Apple Push Notification Services setup requires the use of the OpenSSL command line utility provided in Mac OS X.

Before you can complete registration on the Salesforce side, you need to register with Apple Push Notification Services. The following instructions provide a general outline for what's required. See http://www.raywenderlich.com/32960/ for complete instructions.

Configuration for Apple Push Notification Services

Registering with Apple Push Notification Services (APNS) requires the following items.

Certificate Signing Request (CSR) File
Generate this request using the Keychain Access feature in Mac OS X. You'll also use OpenSSL to export the CSR private key to a file for later use.

App ID from iOS Developer Program
In the iOS Developer Member Center, create an ID for your app, then use the CSR file to generate a certificate. Next, use OpenSSL to combine this certificate with the private key file to create a `.p12` file. You'll need this file later to configure your connected app.

iOS Provisioning Profile
From the iOS Developer Member Center, create a new provisioning profile using your iOS app ID and developer certificate. You then select the devices to include in the profile and download to create the provisioning profile. You can then add the profile to Xcode. Install the profile on your test device using Xcode's Organizer.

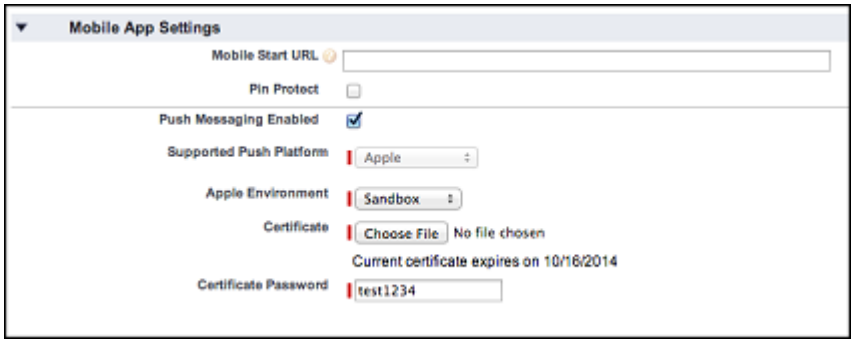
When you've completed the configuration, sign and build your app in Xcode. Check the build logs to verify that the app is using the correct provisioning profile. To view the content of your provisioning profile, run the following command at the Terminal window:

```
security cms -D -i <your profile>.mobileprovision
```

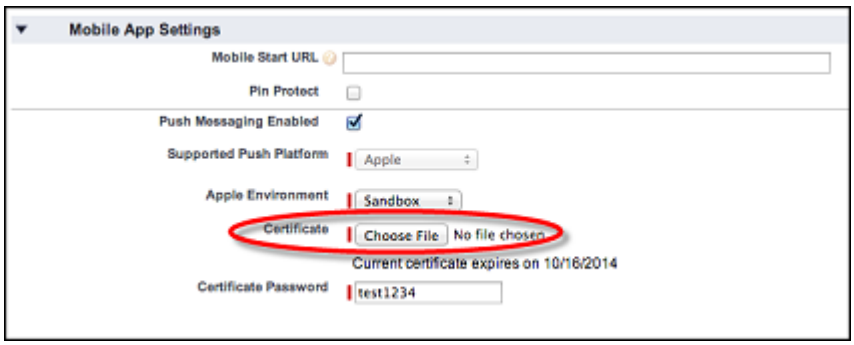
Configure a Connected App for APNS (iOS)

To configure your Salesforce connected app to support push notifications with Apple Push Notification Services (APNS):

1. In your Salesforce org, go to **Setup** > **Create** > **Apps**.
2. In Connected Apps, either click **Edit** next to an existing connected app, or **New** to create a new connected app. If you're creating a new connected app, see Create a Connected App.
3. Under Mobile App Settings, select **Push Messaging Enabled**.
4. For Supported Push Platform, select **Apple**.
   The page expands to show additional settings.



5. Select the Apple Environment that corresponds to your APNS certificate.
6. Add your `.p12` file and its password under **Mobile App Settings** > **Certificate** and **Mobile App Settings** > **Certificate Password**.



Note: You obtain the values for Apple Environment, Certificate, and Certificate Password when you configure your app with APNS.

7. Click **Save**.

Code Modifications (iOS)

Salesforce Mobile SDK for iOS provides the `SFPushNotificationManager` class to handle push registration. To use it, `import <SalesforceSDKCore/SFPushNotificationManager>`. The `SFPushNotificationManager` class is available as a runtime singleton:

```
[SFPushNotificationManager sharedInstance]
```

This class implements four registration methods:

```
- (void)registerForRemoteNotifications;
- (void)didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceTokenData;
- (BOOL)registerForSalesforceNotifications; // for internal use
- (BOOL)unregisterSalesforceNotifications; // for internal use
```

Mobile SDK calls `registerForSalesforceNotifications` after login and `unregisterSalesforceNotifications` at logout. You call the other two methods from your `AppDelegate` class.

Example: **SFPushNotificationManager Example**

To configure your `AppDelegate` class to support push notifications:

1. Register with Apple for push notifications by calling `registerForRemoteNotifications`. Place the call in the `application:didFinishLaunchingWithOptions:` method.

```
- (BOOL)application:(UIApplication *)application
        didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window =
        [[UIWindow alloc] initWithFrame:
                          [UIScreen mainScreen].bounds];
    [self initializeAppViewState];

    //
    // Register with APNS for push notifications. Note that,
    // to receive push notifications from Salesforce,
    // you also need to register for Salesforce notifications
    // in the application:
    // didRegisterForRemoteNotificationsWithDeviceToken:
    // method (as demonstrated below.)
    //
    [[SFPushNotificationManager sharedInstance]
           registerForRemoteNotifications];

    [[SFAuthenticationManager sharedManager]
        loginWithCompletion:self.initialLoginSuccessBlock
                    failure:self.initialLoginFailureBlock];

    return YES;
}
```

   If registration succeeds, Apple passes a device token to the `application:didRegisterForRemoteNotificationsWithDeviceToken:` method of your `AppDelegate` class.

2. Forward the device token from Apple to `SFPushNotificationManager` by calling `didRegisterForRemoteNotificationsWithDeviceToken` on the `SFPushNotificationManager` shared instance.

```
- (void)application:(UIApplication*)application
        didRegisterForRemoteNotificationsWithDeviceToken:(NSData*)deviceToken
{
    //
    // Register your device token with the push notification manager
    //
    [[SFPushNotificationManager sharedInstance]
        didRegisterForRemoteNotificationsWithDeviceToken:deviceToken];

    }}
```

3. Register to receive Salesforce notifications through the connected app by calling `registerForSalesforceNotifications`. Make this call only if the access token for the current session is valid.

```
- (void)application:(UIApplication*)application
    didRegisterForRemoteNotificationsWithDeviceToken:
        (NSData*)deviceToken
```

```
{
    //
    // Register your device token with the
    // push notification manager
    //
    [[SFPushNotificationManager sharedInstance]
        didRegisterForRemoteNotificationsWithDeviceToken:deviceToken];

    if ([SFAccountManager sharedInstance].
        credentials.accessToken != nil) {
        [[SFPushNotificationManager sharedInstance]
            registerForSalesforceNotifications];
    }}
```

4. Add the following method to log an error if registration with Apple fails.

```
- (void)application:(UIApplication*)application
        didFailToRegisterForRemoteNotificationsWithError:(NSError*)error
{
    NSLog(@"Failed to get token, error: %@", error);
}
```

CHAPTER 9 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. A remote access application is implemented as a "connected app" in the Salesforce Help. Remote access applications have been deprecated in favor of connected apps.

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.

3. App re-uses existing session ID.
4. App starts.

OAuth 2.0 User-Agent Flow

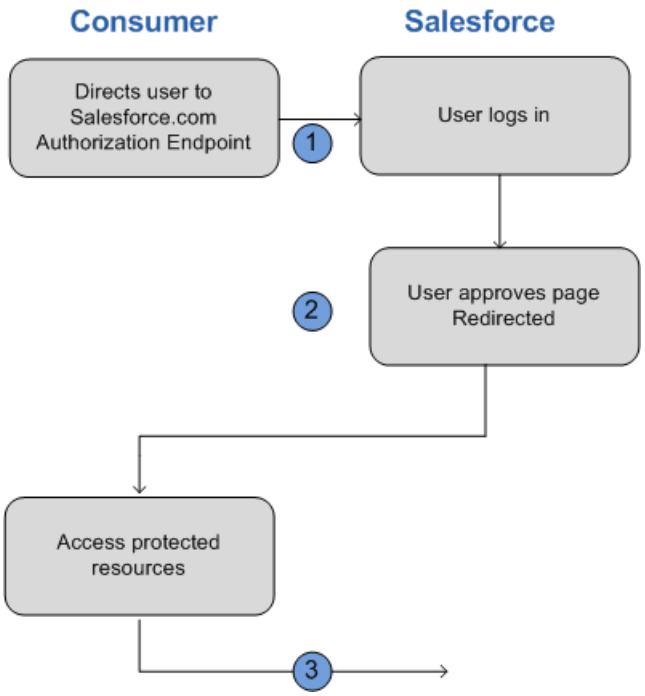
The user-agent authentication flow is used by client applications residing on the user's mobile device. The authentication is based on the user-agent's same-origin policy.

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

OAuth requires scope configuration both on server and on client. The agreement between the two sides defines the scope contract.

- **Server side**—Define scope permissions in a connected app on the Salesforce server. These settings determine which scopes client apps, such as Mobile SDK apps, can request. At a minimum, configure your connected app OAuth settings to match what's specified in your code. For hybrid apps and iOS native apps, `refresh_token`, `web`, and `api` are usually sufficient. For Android native apps, `refresh_token` and `api` are usually sufficient.
- **Client side**—Define scope requests in your Mobile SDK app. Client scope requests must be a subset of the connected app's scope permissions.

Server Side Configuration

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.
`custom_permissions`	Allows access to the custom permissions in an organization associated with the connected app, and shows whether the current user has each permission enabled.

Value	Description
`full`	Allows access to all data accessible by the logged-in user, and encompasses all other scopes. `full` does not return a refresh token. You must explicitly request the `refresh_token` scope to get a refresh token.
`id`	Allows access to the identity URL service. You can request `profile`, `email`, `address`, or `phone`, individually to get the same result as using `id`; they are all synonymous.
`openid`	Allows access to the current, logged in user's unique identifier for OpenID Connect apps. The `openid` scope can be used in the OAuth 2.0 user-agent flow and the OAuth 2.0 Web server authentication flow to get back a signed ID token conforming to the OpenID Connect specifications in addition to the access token.
`refresh_token`	Allows a refresh token to be returned if you are eligible to receive one. This lets the app interact with the user's data while the user is offline, and is synonymous with requesting `offline_access`.
`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.

Note: For Mobile SDK apps, you're always required to select `refresh_token` in server-side Connected App settings. Even if you select the `full` scope, you still must explicitly select `refresh_token`.

Client Side Configuration

The following rules govern scope configuration for Mobile SDK apps.

Scope	Mobile SDK App Configuration
`refresh_token`	Implicitly requested by Mobile SDK for your app; no need to include in your request.
`api`	Include in your request if you're making any Salesforce REST API calls (applies to most apps).
`web`	Include in your request if your app accesses pages defined in a Salesforce org (for hybrid apps, as well as native apps that load Salesforce-based Web pages.)
`full`	Include if you wish to request all permissions. (Mobile SDK implicitly requests `refresh_token` for you.)
`chatter_api`	Include in your request if your app calls Chatter REST APIs.
`id`	(Not needed)
`visualforce`	Use `web` instead.

Using Identity URLs

In addition to the access token, an identity URL is also returned as part of a token response, in the `id` scope 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. The access token can be used in an authorization request header or in a request with the `oauth_token` parameter.

Parameter	Description
Access token	See "Using the Access Token" in the Salesforce Help.
Format	This parameter is optional. Specify the format of the returned output. Valid values are: • `json` • `xml` Instead of using the `format` parameter, the client can also specify the returned format in an accept-request header using one of the following: • `Accept: application/json` • `Accept: application/xml` • `Accept: application/x-www-form-urlencoded` Note the following: • Wildcard accept headers are allowed. `*/*` is accepted and returns JSON. • A list of values is also accepted and is checked left-to-right. For example: `application/xml,application/json,application/html,*/*` returns XML. • 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.

Parameter	Description
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

A valid 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
- `email_verified`—Indicates whether the organization has email verification enabled (`true`), or not (`false`).
- `first_name`—The first name of the user
- `last_name`—The last name of the user
- `timezone`—The time zone in the user's settings
- `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`
- `addr_street`—The street specified in the address of the user's settings
- `addr_city`—The city specified in the address of the user's settings
- `addr_state`—The state specified in the address of the user's settings
- `addr_country`—The country specified in the address of the user's settings
- `addr_zip`—The zip or postal code specified in the address of the user's settings
- `mobile_phone`—The mobile phone number in the user's settings
- `mobile_phone_verified`—The user confirmed this is a valid mobile phone number. See the Mobile User field description.
- `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
- `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)
  - `rest` (REST)
  - `sobjects` (REST)
  - `search` (REST)
  - `query` (REST)
  - `recent` (REST)
  - `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
- `is_app_installed`—The value is `true` when the connected app is installed in the org of the current user and the access token for the user was created using an OAuth flow. If the connected app is not installed, the property does not exist (instead of being `false`). When parsing the response, check both for the existence and value of this property.
- `mobile_policy`—Specific values for managing mobile connected apps. These values are only available when the connected app is installed in the organization of the current user and the app has a defined session timeout value and a PIN (Personal Identification Number) length value.
  - `screen_lock`—The length of time to wait to lock the screen after inactivity
  - `pin_length`—The length of the identification number required to gain access to the mobile app
- `push_service_type`—This response value is set to `apple` if the connected app is registered with Apple Push Notification Service (APNS) for iOS push notifications or `androidGcm` if it's registered with Google Cloud Messaging (GCM) for Android push notifications. The response value type is an array.
- `custom_permissions`—When a request includes the `custom_permissions` scope parameter, the response includes a map containing custom permissions in an organization associated with the connected app. If the connected app is not installed in the organization, or has no associated custom permissions, the response does not contain a `custom_permissions` map. The following shows an example request.

```
http://login.salesforce.com/services/oauth2/authorize?response_type=token&client_
id=3MVG9lKcPoNINVBKV6EgVJiF.snSDwh6_2wSS7BrOhHGEJkC_&redirect_uri=http://www.example.org/qa/security/oauth
/useragent_flow_callback.jsp&scope=api%20id%20custom_permissions
```

The following shows the JSON block in the identity URL response.

```
"custom_permissions":
  {
    "Email.View":true,
    "Email.Create":false,
    "Email.Delete":false
  }
```

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:

Error Code	Request Problem
403 (forbidden) — HTTPS_Required	HTTP
403 (forbidden) — Missing_OAuth_Token	Missing access token
403 (forbidden) — Bad_OAuth_Token	Invalid access token
403 (forbidden) — Wrong_Org	Users in a different organization
404 (not found) — Bad_Id	Invalid or bad user or organization ID
404 (not found) — Inactive	Deactivated user or inactive organization
404 (not found) — No_Access	User lacks proper access to organization or information
404 (not found) — No_Site_Endpoint	Request to an invalid endpoint of a site
404 (not found) — Internal Error	No response from server
406 (not acceptable) — Invalid_Version	Invalid version
406 (not acceptable) — Invalid_Callback	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.

Android Configuration

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`.

Refresh Token Revocation in Android Native Apps

When a refresh token is revoked by an administrator, the default behavior 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.

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.

`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.

Connected Apps

A connected app integrates an application with Salesforce 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 corresponding applications.

A developer or administrator defines a connected app for Salesforce by providing the following information.

- Name, description, logo, and contact information
- A URL where Salesforce can locate the app for authorization or identification
- The authorization protocol: OAuth, SAML, or both
- Optional IP ranges where the connected app might be running
- Optional information about mobile policies the connected app can enforce

Salesforce Mobile SDK apps use connected apps to access Salesforce OAuth services and to call Salesforce REST APIs.

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.

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 OAuth 2.0 service recognizes your custom host name and redirects the user to your site login page if the user is not yet authenticated.

Example: 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 10 Using Communities With Mobile SDK Apps

In this chapter ...

Salesforce Communities is a social aggregation feature that supersedes the Portal feature of earlier releases. Communities can include up to five million users, with logical zones for sharing knowledge with Ideas, Answers, and Chatter Answers. With proper configuration, your community users can use their community login credentials to access your Mobile SDK app. Communities also leverage Site.com to enable you to brand your community site and login screen.

Communities and Mobile SDK Apps

To enable community members to log into your Mobile SDK app, set the appropriate permissions in Salesforce, and change your app's login server configuration to recognize your community URL.

With Communities, members that you designate can use your Mobile SDK app to access Salesforce. You define your own community login endpoint, and the Communities feature builds a branded community login page according to your specifications. It also lets you choose authentication providers and SAML identity providers from a list of popular choices.

Community membership is determined by profiles and permission sets. To enable community members to use your Mobile SDK app, configure the following:

- Make sure that each community member has the API Enabled permission. You can set this permission through profiles or permission sets.
- Configure your community to include your API-enabled profiles and permission sets.
- Configure your Mobile SDK app to use your community's login endpoint.

In addition to these high-level steps, you must take the necessary steps to configure your users properly. Example: Configure a Community For Mobile SDK App Access walks you through the community configuration process for Mobile SDK apps. For the full documentation of the Communities feature, see Getting Started With Communities.

Note: Community login is supported for native and hybrid local Mobile SDK apps on Android and iOS. It is not currently supported for hybrid remote apps using Visualforce.

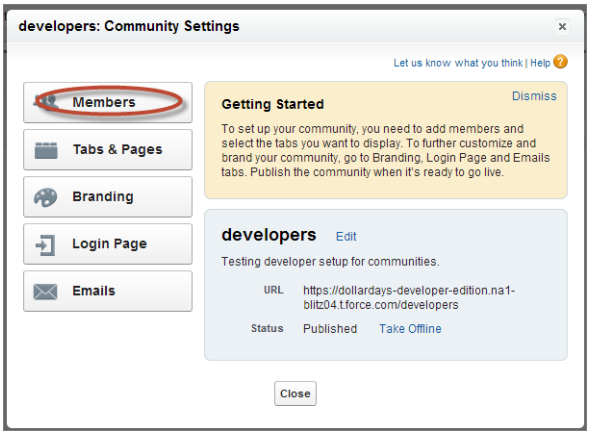
Set Up an API-Enabled Profile

If you're new to communities, start by enabling the community feature in your org. See Enabling Salesforce Communities in Salesforce Help. When you're asked to create a domain name, be sure that it doesn't use SSL (`https://`).

To set up your community, see Creating Communities in Salesforce Help. Note that you'll define a community URL based on the domain name you created when you enabled the community feature.

Next, configure one or more profiles with the API Enabled permissions. You can use these profiles to enable your Mobile SDK app for community members. For detailed instructions, follow the tutorial at Example: Configure a Community For Mobile SDK App Access.

1. Create a new profile or edit an existing one.
2. Edit the profile's details to select API Enabled under **Administrative Permissions**.
3. Save your changes, then edit your community at **Settings** > **Customize** > **Communities** > **Manage Communities**.
4. In *<your community>*: Community Settings, click **Members**.



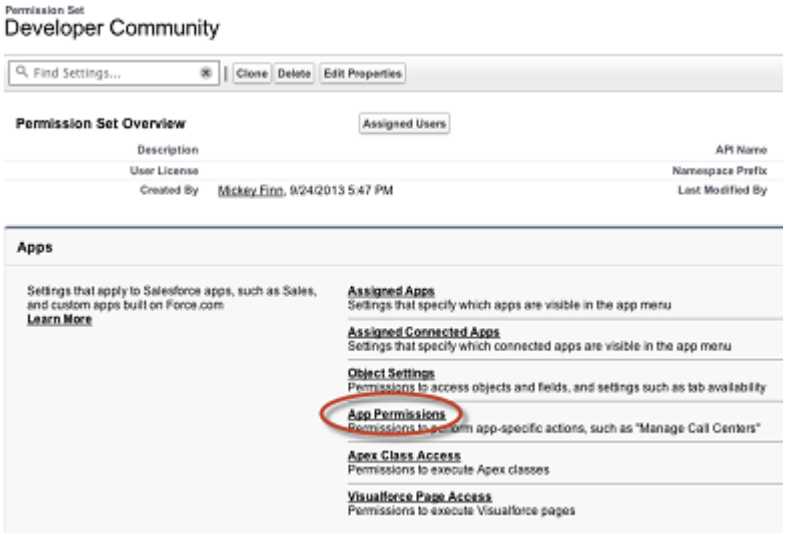
5. Add your API-enabled profile to **Selected Profiles**.

Users to whom these profiles are assigned now have API access. For an overview of profiles, see User Profiles Overview in Salesforce Help.

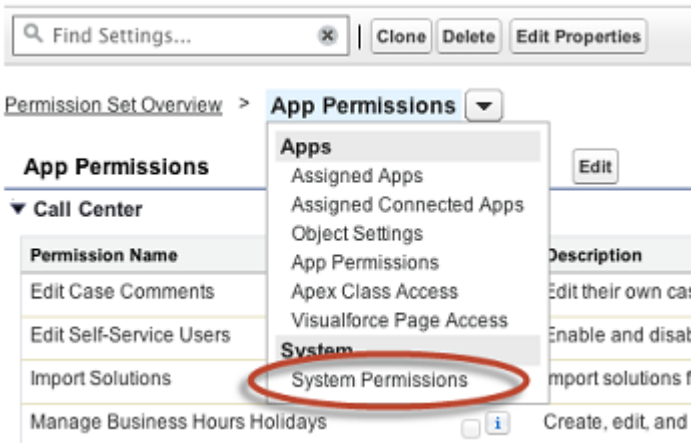
Set Up a Permission Set

Another way to enable mobile apps for your community is through a permission set.

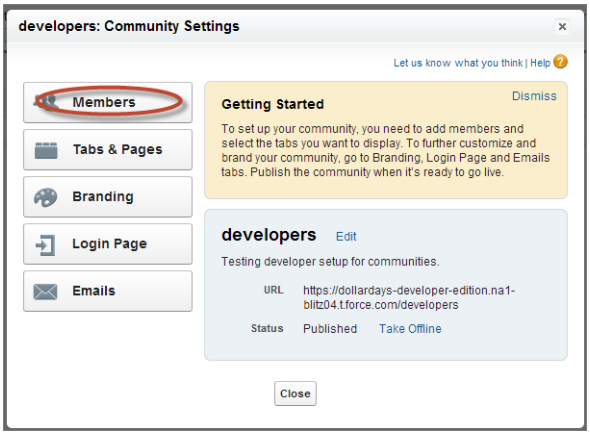
1. To add the API Enabled permission to an existing permission set, in Setup, click **Manage Users** > **Permissions Sets**, select the permission set, and skip to Step 6.
2. To create a permission set, in Setup, click **Administer** > **Manage Users** > **Permission Sets**.
3. Click **New**.
4. Give the Permission Set a label and press `Return` to automatically create the API Name.
5. Click **Next**.
6. Under the Apps section, click **App Permissions**.



7. Click **App Permissions** and select **System** > **System Permissions**.



8. On the System Permissions page, click **Edit** and select **API Enabled**.
9. Click **Save**.
10. Go to **Settings** > **Customize** > **Communities** > **Manage Communities** and click **Edit** next to your community name.
11. In *My Community*: Community Settings, click **Members**.



12. Under Select Permission Sets, add your API-enabled permission set to **Selected Permission Sets**.

Users in this permission set now have API access.

Grant API Access to Users

To extend API access to your community users, add them to a profile or a permission set that sets the API Enabled permission. If you haven't yet configured any profiles or permission sets to include this permission, see Set Up an API-Enabled Profile and Set Up a Permission Set.

Configure the Login Endpoint

Finally, configure the app to use your community login endpoint. The app's mobile platform determines how you configure this setting.

Android

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.)

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
  <server name="XYZ.com Login" url="https://<username>.cloudforce.com"/>
</servers>
```

iOS

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. Add the following key-value pair to your <*appname*>-Info.plist file:

```
<key>SFDCOAuthLoginHost</key>
<string>your_community_login_url_minus_the_https://_prefix</string>
```

It's important to remove the HTTP prefix from your URL. For example, if your community login URL is `https://mycommunity-developer-edition.na15.force.com/fineapps`, your key-value pair would be:

```
<key>SFDCOAuthLoginHost</key>
<string>mycommunity-developer-edition.na15.force.com/fineapps</string>
```

Optionally, you can remove `Settings.bundle` from your class if you don't want users to change the value.

Brand Your Community

If you are using the Salesforce Tabs + Visualforce template, you can customize the look and feel of your community in Community Management by adding your company logo, colors, and copyright. This ensures that your community matches your company's branding and is instantly recognizable to your community members.

Important: If you are using a self-service template or choose to use the Community Builder to create custom pages instead of using standard Salesforce tabs, you can use the Community Builder to design your community's branding too.

EDITIONS

Available in:
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To create, customize, or activate a community:
- "Create and Set Up Communities"

  AND

  Is a member of the community whose Community Management page they're trying to access.

1. Access Community Management in either of the following ways:
   - From the community, click in the global header.
   - From Setup, click **Customize** > **Communities** > **All Communities**, then click **Manage** next to the community name.
2. Click **Administration** > **Branding**.
3. Use the lookups to choose a header and footer for the community.

   The files you're choosing for header and footer must have been previously uploaded to the Documents tab and must be publicly available. The header can be .html, .gif, .jpg, or .png. The footer must be an .html file. The maximum file size for .html files is 100 KB combined. The maximum file size for .gif, .jpg, or .png files is 20 KB. So, if you have a header .html file that is 70 KB and you want to use an .html file for the footer as well, it can only be 30 KB.

   The header you choose replaces the Salesforce logo below the global header. The footer you choose replaces the standard Salesforce copyright and privacy footer.
4. Click **Select Color Scheme** to select from predefined color schemes or click the text box next to the page section fields to select a color from the color picker.

   Note that some of the selected colors impact your community login page and how your community looks in Salesforce1 as well.

Color Choice	Where it Appears
Header Background	Top of the page, under the black global header. If an HTML file is selected in the Header field, it overrides this color choice. Top of the login page. Login page in Salesforce1.
Page Background	Background color for all pages in your community, including the login page.
Primary	Tab that is selected.
Secondary	Top borders of lists and tables. Button on the login page.
Tertiary	Background color for section headers on edit and detail pages.

5. Click **Save**.

Customize Login, Logout, and Self-Registration in Your Community

Configure the login, logout, and self-registration options for your community that are available out-of-the-box, or customize the behavior with Apex and Visualforce or Community Builder pages.

By default, each community is associated with the default login (`CommunitiesLogin`) and self-registration (`CommunitiesSelfReg`) pages and their associated Apex controllers. You can customize the defaults in the following ways:

- Customize the branding of the default login page.
- Customize the login experience by modifying the default login page behavior, using a custom login page, and supporting other authentication providers.
- Redirect users to a different URL on logout.
- Set up self-registration for unlicensed guest users in your community.

EDITIONS

Available in:
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To create, customize, or activate a community:
- "Create and Set Up Communities"

  AND

  Is a member of the community whose Community Management page they're trying to access.

Using External Authentication With Communities

You can use an external authentication provider, such as Facebook©, to log community users into your Mobile SDK app.

Note: Although Salesforce supports Janrain as an authentication provider, it's primarily intended for internal use by Salesforce. We've included it here for the sake of completeness.

About External Authentication Providers

You can enable users to log into your Salesforce organization using their login credentials from an external service provider such as Facebook© or Janrain©.

Note: Social Sign-On (11:33 minutes)

Learn how to configure single sign-on and OAuth-based API access to Salesforce from other sources of user identity.

Do the following to successfully set up an authentication provider for single sign-on.

- Correctly configure the service provider website.
- Create a registration handler using Apex.
- Define the authentication provider in your organization.

When set up is complete, the authentication provider flow is as follows.

1. The user tries to login to Salesforce using a third party identity.
2. The login request is redirected to the third party authentication provider.
3. The user follows the third party login process and approves access.
4. The third party authentication provider redirects the user to Salesforce with credentials.
5. The user is signed into Salesforce.

EDITIONS

Available in:
- Professional
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To view the settings:
- "View Setup and Configuration"

To edit the settings:
- "Customize Application"

  AND

  "Manage Auth. Providers"

Note: If a user has an existing Salesforce session, after authentication with the third party they are automatically redirected to the page where they can approve the link to their Salesforce account.

Defining Your Authentication Provider

We support the following providers:

- Facebook
- Google
- Janrain
- LinkedIn
- Microsoft Access Control Service
- Salesforce
- Twitter
- Any service provider who implements the OpenID Connect protocol

Adding Functionality to Your Authentication Provider

You can add functionality to your authentication provider by using additional request parameters.

- Scope – Customizes the permissions requested from the third party
- Site – Enables the provider to be used with a site
- StartURL – Sends the user to a specified location after authentication
- Community – Sends the user to a specific community after authentication
- "Authorization Endpoint" in the Salesforce Help – Sends the user to a specific endpoint for authentication (Salesforce authentication providers, only)

Creating an Apex Registration Handler

A registration handler class is required to use Authentication Providers for the single sign-on flow. The Apex registration handler class must implement the `Auth.RegistrationHandler` interface, which defines two methods. Salesforce invokes the appropriate method on callback, depending on whether the user has used this provider before or not. When you create the authentication provider, you can automatically create an Apex template class for testing purposes. For more information, see RegistrationHandler in the *Force.com Apex Code Developer's Guide*.

Using the Community URL Parameter

Send your user to a specific Community after authenticating.

To direct your users to a specific community after authenticating, you need to specify a URL with the `community` request parameter. If you don't add the parameter, the user is sent to either `/home/home.jsp` (for a portal or standard application) or to the default sites page (for a site) after authentication completes.

Example: For example, with a `Single Sign-On Initialization URL`, the user is sent to this location after being logged in. For an `Existing User Linking URL`, the "Continue to Salesforce" link on the confirmation page leads to this page.

The following is an example of a `community` parameter added to the `Single Sign-On Initialization URL`, where:

- *`orgID`* is your Auth. Provider ID
- *`URLsuffix`* is the value you specified when you defined the authentication provider

`https://login.salesforce.com/services/auth/sso/orgID/URLsuffix?community=https://acme.force.com/support`

EDITIONS

Available in:
- Professional
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To view the settings:
- "View Setup and Configuration"

To edit the settings:
- "Customize Application"

  AND

  "Manage Auth. Providers"

Using the Scope Parameter

Customizes the permissions requested from the third party like Facebook or Janrain so that the returned access token has additional permissions.

You can customize requests to a third party to receive access tokens with additional permissions. Then you use `Auth.AuthToken` methods to retrieve the access token that was granted so you can use those permissions with the third party.

The default scopes vary depending on the third party, but usually do not allow access to much more than basic user information. Every provider type (Open ID Connect, Facebook, Salesforce, and others), has a set of default scopes it sends along with the request to the authorization endpoint. For example, Salesforce's default scope is `id`.

You can send scopes in a space-delimited string. The space-delimited string of requested scopes is sent as-is to the third party, and overrides the default permissions requested by authentication providers.

Janrain does not use this parameter; additional permissions must be configured within Janrain.

Example: The following is an example of a `scope` parameter requesting the Salesforce scopes `api` and `web`, added to the `Single Sign-On Initialization URL`, where:

- *`orgID`* is your Auth. Provider ID
- *`URLsuffix`* is the value you specified when you defined the authentication provider

`https://login.salesforce.com/services/auth/sso/orgID/URLsuffix?scope=id%20api%20web`

EDITIONS

Available in:
- Professional
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To view the settings:
- "View Setup and Configuration"

To edit the settings:
- "Customize Application"

  AND

  "Manage Auth. Providers"

Valid scopes vary depending on the third party; refer to your individual third-party documentation. For example, Salesforce scopes 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.
`custom_permissions`	Allows access to the custom permissions in an organization associated with the connected app, and shows whether the current user has each permission enabled.
`full`	Allows access to all data accessible by the logged-in user, and encompasses all other scopes. `full` does not return a refresh token. You must explicitly request the `refresh_token` scope to get a refresh token.
`id`	Allows access to the identity URL service. You can request `profile`, `email`, `address`, or `phone`, individually to get the same result as using `id`; they are all synonymous.
`openid`	Allows access to the current, logged in user's unique identifier for OpenID Connect apps. The `openid` scope can be used in the OAuth 2.0 user-agent flow and the OAuth 2.0 Web server authentication flow to get back a signed ID token conforming to the OpenID Connect specifications in addition to the access token.
`refresh_token`	Allows a refresh token to be returned if you are eligible to receive one. This lets the app interact with the user's data while the user is offline, and is synonymous with requesting `offline_access`.
`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.

Configuring a Facebook Authentication Provider

To use Facebook as an authentication provider:

1. Set up a Facebook application, making Salesforce the application domain.
2. Define a Facebook authentication provider in your Salesforce organization.
3. Update your Facebook application to use the `Callback URL` generated by Salesforce as the Facebook `Website Site URL`.
4. Test the connection.

EDITIONS

Available in:
- Professional
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To view the settings:
- "View Setup and Configuration"

To edit the settings:
- "Customize Application"

  AND

  "Manage Auth. Providers"

Setting up a Facebook Application

Before you can configure Facebook for your Salesforce organization, you must set up an application in Facebook:

Note: You can skip this step by allowing Salesforce to use its own default application. For more information, see Using Salesforce-Managed Values in Auth. Provider Setup.

1. Go to the Facebook website and create a new application.
2. Modify the application settings and set the Application Domain to Salesforce.
3. Note the Application ID and the Application Secret.

Defining a Facebook Provider in your Salesforce Organization

You need the Facebook Application ID and Application Secret to set up a Facebook provider in your Salesforce organization.

Note: You can skip specifying these key values in the provider setup by allowing Salesforce to manage the values for you. For more information, see Using Salesforce-Managed Values in Auth. Provider Setup.

1. From Setup, click **Security Controls** > **Auth. Providers**.
2. Click **New**.
3. Select Facebook for the `Provider Type`.
4. Enter a `Name` for the provider.
5. Enter the `URL Suffix`. This is used in the client configuration URLs. For example, if the URL suffix of your provider is "MyFacebookProvider", your single sign-on URL is similar to: `https://login.salesforce.com/auth/sso/00Dx00000000001/MyFacebookProvider`.
6. Use the Application ID from Facebook for the `Consumer Key` field.
7. Use the Application Secret from Facebook for the `Consumer Secret` field.
8. Optionally, set the following fields.
   a. Enter the base URL from Facebook for the `Authorize Endpoint URL`. For example, `https://www.facebook.com/v2.2/dialog/oauth`. If you leave this field blank, Salesforce uses the version of the Facebook API that your application uses.

      Tip: You can add query string parameters to the base URL, if necessary. For example, to get a refresh token from Google for offline access, use `https://accounts.google.com/o/oauth2/auth?access_type=offline&approval_prompt=force`. In this example, the additional `approval_prompt` parameter is necessary to ask the user to accept the refresh action, so that Google continues to provide refresh tokens after the first one.

b. Enter the `Token Endpoint URL` from Facebook. For example, `https://www.facebook.com/v2.2/dialog/oauth`. If you leave this field blank, Salesforce uses the version of the Facebook API that your application uses.

c. Enter the `User Info Endpoint URL` to change the values requested from Facebook's profile API. See https://developers.facebook.com/docs/facebook-login/permissions/v2.0#reference-public_profile for more information on fields. The requested fields must correspond to requested scopes. If you leave this field blank, Salesforce uses the version of the Facebook API that your application uses.

d. `Default Scopes` to send along with the request to the authorization endpoint. Otherwise, the hardcoded defaults for the provider type are used (see Facebook's developer documentation for these defaults).

For more information, see Using the Scope Parameter

e. `Custom Error URL` for the provider to use to report any errors.

f. `Custom Logout URL` to provide a specific destination for users after they log out, if they authenticated using the single sign-on flow. Use this field to direct users to a branded logout page or destination other than the default Salesforce logout page. The URL must be fully qualified with an `http` or `https` prefix, such as `https://acme.my.salesforce.com`.

g. Select an already existing Apex class as the `Registration Handler` class or click `Automatically create a registration handler template` to create an Apex class template for the registration handler. You must edit this class and modify the default content before using it.

Note: You must specify a registration handler class for Salesforce to generate the `Single Sign-On Initialization URL`.

h. Select the user that runs the Apex handler class for **Execute Registration As**. The user must have "Manage Users" permission. A user is required if you selected a registration handler class or are automatically creating one.

i. To use a portal with your provider, select the portal from the Portal drop-down list.

j. Use the `Icon URL` field to add a path to an icon to display as a button on the login page for a community. This icon applies to a community only, and does not appear on the login page for your Salesforce organization or custom domain created with My Domain. Users click the button to log in with the associated authentication provider for the community.

You can specify a path to your own image, or copy the URL for one of our sample icons into the field.

9. Click **Save**.

Be sure to note the generated `Auth. Provider Id` value. You must use it with the `Auth.AuthToken` Apex class.

Several client configuration URLs are generated after defining the authentication provider:

- `Test-Only Initialization URL`: Administrators use this URL to ensure the third-party provider is set up correctly. The administrator opens this URL in a browser, signs in to the third party, and is redirected back to Salesforce with a map of attributes.
- `Single Sign-On Initialization URL`: Use this URL to perform single sign-on into Salesforce from a third party (using third-party credentials). The end user opens this URL in a browser, and signs in to the third party. This then either creates a new user for them, or updates an existing user, and then signs them into Salesforce as that user.
- `Existing User Linking URL`: Use this URL to link existing Salesforce users to a third-party account. The end user opens this URL in a browser, signs in to the third party, signs in to Salesforce, and approves the link.
- `Oauth-Only Initialization URL`: Use this URL to obtain OAuth access tokens for a third party. Users must authenticate with Salesforce for the third-party service to get a token; this flow does not provide for future single sign-on functionality.
- `Callback URL`: Use the callback URL for the endpoint that the authentication provider calls back to for configuration. The authentication provider has to redirect to the `Callback URL` with information for each of the above client configuration URLs.

The client configuration URLs support additional request parameters that enable you to direct users to log into specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Updating Your Facebook Application

After defining the Facebook authentication provider in your Salesforce organization, go back to Facebook and update your application to use the `Callback URL` as the Facebook `Website Site URL`.

Testing the Single Sign-On Connection

In a browser, open the `Test-Only Initialization URL` on the Auth. Provider detail page. It should redirect you to Facebook and ask you to sign in. Upon doing so, you are asked to authorize your application. After you authorize, you are redirected back to Salesforce.

Configure a Salesforce Authentication Provider

You can use a connected app as an authentication provider. You must complete these steps:

1. Define a Connected App.
2. Define the Salesforce authentication provider in your organization.
3. Test the connection.

EDITIONS

Available in:
- Professional
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To view the settings:
- "View Setup and Configuration"

To edit the settings:
- "Customize Application"

  AND

  "Manage Auth. Providers"

Define a Connected App

Before you can configure a Salesforce provider for your Salesforce organization, you must define a connected app that uses single sign-on. Define connected apps under Setup, in **Create** > **Apps**.

After you finish defining a connected app, save the values from the `Consumer Key` and `Consumer Secret` fields.

Note: You can skip this step by allowing Salesforce to use its own default application. For more information, see Using Salesforce-Managed Values in Auth. Provider Setup.

Define the Salesforce Authentication Provider in your Organization

You need the values from the `Consumer Key` and `Consumer Secret` fields of the connected app definition to set up the authentication provider in your organization.

Note: You can skip specifying these key values in the provider setup by allowing Salesforce to manage the values for you. For more information, see Using Salesforce-Managed Values in Auth. Provider Setup.

1. From Setup, click **Security Controls** > **Auth. Providers**.
2. Click **New**.
3. Select Salesforce for the `Provider Type`.
4. Enter a `Name` for the provider.
5. Enter the `URL Suffix`. This is used in the client configuration URLs. For example, if the URL suffix of your provider is "MySFDCProvider", your single sign-on URL is similar to `https://login.salesforce.com/auth/sso/00Dx00000000001/MySFDCProvider`.

6. Paste the value of `Consumer Key` from the connected app definition into the `Consumer Key` field.
7. Paste the value of `Consumer Secret` from the connected app definition into the `Consumer Secret` field.
8. Optionally, set the following fields.
   a. `Authorize Endpoint URL` to specify an OAuth authorization URL.

      For the `Authorize Endpoint URL`, the host name can include a sandbox or custom domain name (created using My Domain), but the URL must end in `.salesforce.com`, and the path must end in `/services/oauth2/authorize`. For example, `https://test.salesforce.com/services/oauth2/authorize`.

   b. `Token Endpoint URL` to specify an OAuth token URL.

      For the `Token Endpoint URL`, the host name can include a sandbox or custom domain name (created using My Domain), but the URL must end in `.salesforce.com`, and the path must end in `/services/oauth2/token`. For example, `https://test.salesforce.com/services/oauth2/token`.

   c. `Default Scopes` to send along with the request to the authorization endpoint. Otherwise, the hardcoded default is used.

      For more information, see Using the Scope Parameter.

      Note: When editing the settings for an existing Salesforce authentication provider, you might have the option to select a checkbox to include the organization ID for third-party account links. For Salesforce authentication providers set up in the Summer '14 release and earlier, the user identity provided by an organization does not include the organization ID. So, the destination organization can't differentiate between users with the same user ID from two sources (such as two sandboxes). Select this checkbox if you have an existing organization with two users (one from each sandbox) mapped to the same user in the destination organization, and you want to keep the identities separate. Otherwise, leave this checkbox unselected. After enabling this feature, your users need to re-approve the linkage to all of their third party links. These links are listed in the Third-Party Account Links section of a user's detail page. Salesforce authentication providers created in the Winter '15 release and later have this setting enabled by default and do not display the checkbox.

   d. `Custom Error URL` for the provider to use to report any errors.
   e. `Custom Logout URL` to provide a specific destination for users after they log out, if they authenticated using the single sign-on flow. Use this field to direct users to a branded logout page or destination other than the default Salesforce logout page. The URL must be fully qualified with an `http` or `https` prefix, such as `https://acme.my.salesforce.com`.
9. Select an already existing Apex class as the `Registration Handler` class or click `Automatically create a registration handler template` to create the Apex class template for the registration handler. You must edit this template class to modify the default content before using it.

   Note: You must specify a registration handler class for Salesforce to generate the `Single Sign-On Initialization URL`.
10. Select the user that runs the Apex handler class for `Execute Registration As`. The user must have "Manage Users" permission. A user is required if you selected a registration handler class or are automatically creating one.
11. To use a portal with your provider, select the portal from the Portal drop-down list.
12. Use the `Icon URL` field to add a path to an icon to display as a button on the login page for a community. This icon applies to a community only, and does not appear on the login page for your Salesforce organization or custom domain created with My Domain. Users click the button to log in with the associated authentication provider for the community.

    You can specify a path to your own image, or copy the URL for one of our sample icons into the field.
13. Click `Save`.

Note the value of the Client Configuration URLs. You need the `Callback URL` to complete the last step, and you use the `Test-Only Initialization URL` to check your configuration. Also be sure to note the `Auth. Provider Id` value because you must use it with the `Auth.AuthToken` Apex class.

14. Return to the connected app definition you created above (under Setup, in **Create** > **Apps**, click on the connected app name) and paste the value of `Callback URL` from the authentication provider into the `Callback URL` field.

Several client configuration URLs are generated after defining the authentication provider:

- `Test-Only Initialization URL`: Administrators use this URL to ensure the third-party provider is set up correctly. The administrator opens this URL in a browser, signs in to the third party, and is redirected back to Salesforce with a map of attributes.
- `Single Sign-On Initialization URL`: Use this URL to perform single sign-on into Salesforce from a third party (using third-party credentials). The end user opens this URL in a browser, and signs in to the third party. This then either creates a new user for them, or updates an existing user, and then signs them into Salesforce as that user.
- `Existing User Linking URL`: Use this URL to link existing Salesforce users to a third-party account. The end user opens this URL in a browser, signs in to the third party, signs in to Salesforce, and approves the link.
- `Oauth-Only Initialization URL`: Use this URL to obtain OAuth access tokens for a third party. Users must authenticate with Salesforce for the third-party service to get a token; this flow does not provide for future single sign-on functionality.
- `Callback URL`: Use the callback URL for the endpoint that the authentication provider calls back to for configuration. The authentication provider has to redirect to the `Callback URL` with information for each of the above client configuration URLs.

The client configuration URLs support additional request parameters that enable you to direct users to log into specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Test the Single Sign-On Connection

In a browser, open the `Test-Only Initialization URL` on the Auth. Provider detail page. Both the authorizing organization and target organization must be in the same environment, such as production or sandbox.

Configure an OpenID Connect Authentication Provider

You can use any third-party Web application that implements the server side of the OpenID Connect protocol, such as Amazon, Google, and PayPal, as an authentication provider.

You must complete these steps to configure an OpenID authentication provider:

1. Register your application, making Salesforce the application domain.
2. Define an OpenID Connect authentication provider in your Salesforce organization.
3. Update your application to use the `Callback URL` generated by Salesforce as the callback URL.
4. Test the connection.

EDITIONS

Available in:
- Professional
- Enterprise
- Performance
- Unlimited
- Developer

USER PERMISSIONS

To view the settings:
- "View Setup and Configuration"

To edit the settings:
- "Customize Application"

  AND

  "Manage Auth. Providers"

Register an OpenID Connect Application

Before you can configure a Web application for your Salesforce organization, you must register it with your service provider. The process varies depending on the service provider. For example, to register a Google app, Create an OAuth 2.0 Client ID.

1. Register your application on your service provider's website.
2. Modify the application settings and set the application domain (or `Home Page URL`) to Salesforce.
3. Note the Client ID and Client Secret, as well as the Authorize Endpoint URL, Token Endpoint URL, and User Info Endpoint URL, which should be available in the provider's documentation. Here are some common OpenID Connect service providers:
   - Amazon
   - Google
   - PayPal

Define an OpenID Connect Provider in Your Salesforce Organization

You need some information from your provider (the Client ID and Client Secret, as well as the Authorize Endpoint URL, Token Endpoint URL, and User Info Endpoint URL) to configure your application in your Salesforce organization.

1. From Setup, click **Security Controls** > **Auth. Providers**.
2. Click **New**.
3. Select OpenID Connect for the `Provider Type`.
4. Enter a `Name` for the provider.
5. Enter the `URL Suffix`. This is used in the client configuration URLs. For example, if the URL suffix of your provider is "MyOpenIDConnectProvider," your single sign-on URL is similar to: `https://login.salesforce.com/auth/sso/00Dx00000000001/MyOpenIDConnectProvider`.
6. Use the Client ID from your provider for the `Consumer Key` field.
7. Use the Client Secret from your provider for the `Consumer Secret` field.
8. Enter the base URL from your provider for the `Authorize Endpoint URL`.

   Tip: You can add query string parameters to the base URL, if necessary. For example, to get a refresh token from Google for offline access, use

`https://accounts.google.com/o/oauth2/auth?access_type=offline&approval_prompt=force`. In this specific case, the additional `approval_prompt` parameter is necessary to ask the user to accept the refresh action, so Google will continue to provide refresh tokens after the first one.

9. Enter the `Token Endpoint URL` from your provider.
10. Optionally, set the following fields.
    a. `User Info Endpoint URL` from your provider.
    b. `Token Issuer`. This value identifies the source of the authentication token in the form *`https: URL`*. If this value is specified, the provider must include an `id_token` value in the response to a token request. The `id_token` value is not required for a refresh token flow (but will be validated by Salesforce if provided).
    c. `Default Scopes` to send along with the request to the authorization endpoint. Otherwise, the hardcoded defaults for the provider type are used (see the OpenID Connect developer documentation for these defaults).

       For more information, see Using the Scope Parameter.

11. You can select `Send access token in header` to have the token sent in a header instead of a query string.
12. Optionally, set the following fields.
    a. `Custom Error URL` for the provider to use to report any errors.
    b. `Custom Logout URL` to provide a specific destination for users after they log out, if they authenticated using the single sign-on flow. Use this field to direct users to a branded logout page or destination other than the default Salesforce logout page. The URL must be fully qualified with an `http` or `https` prefix, such as `https://acme.my.salesforce.com`.
    c. Select an existing Apex class as the `Registration Handler` class or click `Automatically create a registration handler template` to create an Apex class template for the registration handler. You must edit this class and modify the default content before using it.

       Note: You must specify a registration handler class for Salesforce to generate the `Single Sign-On Initialization URL`.

    d. Select the user that runs the Apex handler class for **Execute Registration As**. The user must have the "Manage Users" permission. A user is required if you selected a registration handler class or are automatically creating one.
    e. To use a portal with your provider, select the portal from the Portal drop-down list.
    f. Use the `Icon URL` field to add a path to an icon to display as a button on the login page for a community. This icon applies to a community only, and does not appear on the login page for your Salesforce organization or custom domain created with My Domain. Users click the button to log in with the associated authentication provider for the community.

       You can specify a path to your own image, or copy the URL for one of our sample icons into the field.

13. Click **Save**.

Be sure to note the generated `Auth. Provider Id` value. You must use it with the `Auth.AuthToken` Apex class.

Several client configuration URLs are generated after defining the authentication provider:

- `Test-Only Initialization URL`: Administrators use this URL to ensure the third-party provider is set up correctly. The administrator opens this URL in a browser, signs in to the third party, and is redirected back to Salesforce with a map of attributes.
- `Single Sign-On Initialization URL`: Use this URL to perform single sign-on into Salesforce from a third party (using third-party credentials). The end user opens this URL in a browser, and signs in to the third party. This then either creates a new user for them, or updates an existing user, and then signs them into Salesforce as that user.

- `Existing User Linking URL`: Use this URL to link existing Salesforce users to a third-party account. The end user opens this URL in a browser, signs in to the third party, signs in to Salesforce, and approves the link.
- `Oauth-Only Initialization URL`: Use this URL to obtain OAuth access tokens for a third party. Users must authenticate with Salesforce for the third-party service to get a token; this flow does not provide for future single sign-on functionality.
- `Callback URL`: Use the callback URL for the endpoint that the authentication provider calls back to for configuration. The authentication provider has to redirect to the `Callback URL` with information for each of the above client configuration URLs.

The client configuration URLs support additional request parameters that enable you to direct users to log into specific sites, obtain customized permissions from the third party, or go to a specific location after authenticating.

Update Your OpenID Connect Application

After defining the authentication provider in your Salesforce organization, go back to your provider and update your application's `Callback URL` (also called the `Authorized Redirect URI` for Google applications and `Return URL` for PayPal).

Test the Single Sign-On Connection

In a browser, open the `Test-Only Initialization URL` on the Auth. Provider detail page. It should redirect you to your provider's service and ask you to sign in. Upon doing so, you're asked to authorize your application. After you authorize, you're redirected back to Salesforce.

Example: Configure a Community For Mobile SDK App Access

Configuring your community to support logins from Mobile SDK apps can be tricky. This tutorial helps you see the details and correct sequence first-hand.

When you configure community users for mobile access, sequence and protocol affect your success. For example, if you create a user that's not associated with a contact, that user won't be able to log in on a mobile device. Here are some important guidelines to keep in mind:

- Create users only from contacts that belong to accounts. You can't create the user first and then associate it with a contact later.
- Be sure you've assigned a role to the owner of any account you use. Otherwise, the user gets an error when trying to log in.
- On iOS devices, when you create a Custom Host for your app in Settings, remove the `http[s]://` prefix. The iOS core appends the prefix at runtime, which could result in an invalid address if you explicitly include it.

1. Add Permissions to a Profile
2. Create a Community
3. Add the API User Profile To Your Community
4. Create a New Contact and User
5. Test Your New Community Login

Add Permissions to a Profile

Create a profile that has API Enabled and Enable Chatter permissions.

1. Go to **Setup** > **Manage Users** > **Profiles**.
2. Click **New Profile**.
3. For Existing Profile select **Customer Community User**.

4. For **Profile Name** type *FineApps API User*.
5. Click **Save**.
6. On the FineApps API User page, click **Edit**.
7. For **Administrative Permissions** select **API Enabled** and **Enable Chatter**.

   Note: A user who doesn't have the Enable Chatter permission gets an insufficient privileges error immediately after successfully logging into your community in Salesforce.

8. Click **Save**.

Note: In this tutorial we use a profile, but you can also use a permission set that includes the required permissions.

Create a Community

Create a community and a community login URL.

The following steps are fully documented at Enabling Salesforce Communities and Creating Communities in Salesforce Help.

1. In **Setup**, go to **Customize** > **Communities**.
2. If you don't see a **Manage Communities** options:
   a. Click **Settings**.
   b. Under Enable communities, select **Enable communities**.
   c. Under Select a domain name, enter a unique name, such as *fineapps.<your_name>.force.com* for **Domain name**.
   d. Click **Check Availability** to make sure the domain name isn't already being used.
   e. Click **Save**.
3. Go to **Setup** > **Customize** > **Communities** > **Manage Communities**.
4. Click **New Community**.
5. Name the new community *FineApps Users* and enter a description.
6. For **URL**, type *customers* in the suffix edit box.
   The full URL shown, including your suffix, becomes the new URL for your community.
7. Click **Create**, then click **Edit**.

Add the API User Profile To Your Community

Add the API User profile to your community setup on the Members page.

1. Click **Members**.
2. For Search, select **All**.
3. Select **FineApps API User** in the Available Profiles list, then click **Add**.
4. Click **Save**.
5. Click **Publish**.
6. Dismiss the confirmation dialog box and click **Close**.

Create a New Contact and User

Instead of creating users directly, create a contact on an account, then create the user from that contact.

If you don't currently have any accounts,

1. Click the **Accounts** tab.
2. If your org doesn't yet contain any accounts:
   a. In Quick Create, enter *My Test Account* for **Account Name**.
   b. Click **Save**
3. In Recent Accounts click **My Test Account** or any other account name. Note the Account Owner's name.
4. Go to **Manage Users** > **Users** and click **Edit** next to your Account Owner's name.
5. Make sure that **Role** is set to a management role, such as CEO.
6. Click **Save**.
7. Click the **Accounts** tab and again click the account's name.
8. In Contacts, click **New Contact**.
9. Fill in the following information: First Name: *Jim*, Last Name: *Parker*. Click **Save**.
10. On the Contact page for Jim Parker, click **Manage External User**, then select **Enable Customer User**.
11. For User License select **Customer Community**.
12. For Profile select the FineApps API User.
13. Use the following values for the other required fields:

Field	Value
Email	Enter your active valid email address.
Username	*jimparker@fineapps.com*
Nickname	*jimmyp*

You can remove any non-required information if it's automatically filled in by the browser.

14. Click **Save**.
15. Wait for an email to arrive in your inbox welcoming Jim Parker, then click the link in the email to create a password. Set the password to "mobile333".

Test Your New Community Login

Test your community setup by logging into your Mobile SDK native or hybrid local app as your new contact.

To log into your mobile app through your community, configure the settings in your Mobile SDK app to recognize your community login URL that ends with `/fineapps`.

1. For Android:
   a. Open your Android project in Eclipse.
   b. In the Project Explorer, go to the `res` folder and create a new (or select the existing) `xml` folder.

c. In the `xml` folder, create a new text file. You can do this using either the **File** menu or the *`CTRL-Click`* (or *`Right-Click`*) menu.

d. In the new text file, add the following XML. Replace the server URL with your community login URL:

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
 <server name="Community Login"
url="https://fineapps-developer-edition.<instance>.force.com/fineapps">
</servers>
```

e. Save the file as `servers.xml`.

2. For iOS:

a. Open your iOS project in Xcode.

b. Using the Project Navigator, open **Supporting Files** > **<*appname*>-Info.plist**.

c. Change the **SFDCOAuthLoginHost** value to your community login URL minus the `https://` prefix. For example:

```
fineapps-developer-edition.<instance>.force.com/fineapps
```

d. On your iOS simulator or device, go to **Settings** > **<*your_app_name*>**.

e. Click **Login Host** and select **Custom Host**.

f. Click **Back**.

g. Edit **Custom Host**, setting it to the `SFDCOAuthLoginHost` value you specified in the `<appname>-Info.plist` file.

3. Start your app on your device, simulator, or emulator, and log in with username *`jimparker@fineapps.com`* and password *`mobiletest1234`*.

Note: If you leave your mobile app at the login screen for an extended time without logging in, you might get an "insufficient privileges" error when you try to log in. If this happens, close and reopen the app, then log in immediately.

Example: Configure a Community For Facebook Authentication

You can extend the reach of your community by configuring an external authentication provider to handle community logins.

This example extends the previous example to use Facebook as an authentication front end. In this simple scenario, we configure the external authentication provider to accept any authenticated Facebook user into the community.

If your community is already configured for mobile app logins, you don't need to change your mobile app or your connected app to use external authentication. Instead, you define a Facebook app, a Salesforce Auth. Provider, and an Auth. Provider Apex class. You also make a minor change to your community setup.

Create a Facebook App

To enable community logins through Facebook, start by creating a Facebook app.

A Facebook app is comparable to a Salesforce connected app. It is a container for settings that govern the connectivity and authentication of your app on mobile devices.

1. Go to developers.facebook.com.
2. Log in with your Facebook developer account, or register if you're not a registered Facebook developer.

3. Go to **Apps** > **Create a New App**.
4. Set display name to "FineApps Community Test".
5. Add a Namespace, if you want. Per Facebook's requirements, a namespace label must be twenty characters or less, using only lowercase letters, dashes, and underscores. For example, "my_fb_goodapps".
6. For Category, choose **Utilities**.
7. Copy and store your App ID and App Secret for later use.

You can log in to the app using the following URL:

`https://developers.facebook.com/apps/<App ID>/dashboard/`

Define a Salesforce Auth. Provider

To enable external authentication in Salesforce, create an Auth. Provider.

External authentication through Facebook requires the App ID and App Secret from the Facebook app that you created in the previous step.

1. In Setup, go to **Security Controls** > **Auth. Providers**.
2. Click **New**.
3. Configure the Auth. Provider fields as shown in the following table.

Field	Value
`Provider Type`	Select **Facebook**.
`Name`	Enter *`FB Community Login`*.
`URL Suffix`	Accept the default. Note: You may also provide any other string that conforms to URL syntax, but for this example the default works best.
`Consumer Key`	Enter the App ID from your Facebook app.
`Consumer Secret`	Enter the App Secret from your Facebook app.
`Custom Error URL`	Leave blank.

4. For Registration Handler, click **Automatically create a registration handler template**.
5. For Execute Registration As:, click Search and choose a community member who has administrative privileges.
6. Leave Portal blank.
7. Click **Save**.

   Salesforce creates a new Apex class that extends `RegistrationHandler`. The class name takes the form *AutocreatedRegHandlerxxxxxx*....
8. Copy the Auth. Provider ID for later use.

9. In the detail page for your new Auth. Provider, under Client Configuration, copy the Callback URL for later use.

   The callback URL takes the form
   `https://login.salesforce.com/services/authcallback/<id>/<Auth.Provider_URL_Suffix>`.

Configure Your Facebook App

Next, you need to configure the community to use your Salesforce Auth. Provider for logins.

Now that you've defined a Salesforce Auth. Provider, complete the authentication protocol by linking your Facebook app to your Auth. Provider. You provide the Salesforce login URL and the callback URL, which contains your Auth. Provider ID and the Auth. Provider's URL suffix.

1. In your Facebook app, go to **Settings**.
2. In App Domains, enter *login.salesforce.com*.
3. Click **+Add Platform**.
4. Select **Website**.
5. For Site URL, enter your Auth. Provider's callback URL.
6. For **Contact Email**, enter your valid email address.
7. In the left panel, set Status & Review to **Yes**. With this setting, all Facebook users can use their Facebook logins to create user accounts in your community.
8. Click **Save**.
9. Click **Confirm**.

Customize the Auth. Provider Apex Class

Use the Apex class for your Auth. Provider to define filtering logic that controls who may enter your community.

1. In Setup, go to **Develop** > **Apex Classes**.
2. Click **Edit** next to your Auth. Provider class. The default class name starts with "AutocreatedRegHandlerxxxxxx…"
3. To implement the `canCreateUser()` method, simply return true.

   ```
   global boolean canCreateUser(Auth.UserData data) {
    return true;
   }
   ```

   This implementation allows anyone who logs in through Facebook to join your community.

   Note: If you want your community to be accessible only to existing community members, implement a filter to recognize every valid user in your community. Base your filter on any unique data in the Facebook packet, such as username or email address, and then validate that data against similar fields in your community members' records.

4. Change the `createUser()` code:
   a. Replace "Acme" with *FineApps* in the account name query.
   b. Replace the username suffix ("@acmecorp.com") with *@fineapps.com*.
   c. Change the profile name in the profile query ("Customer Portal User") to *API Enabled*.
5. In the `updateUser()` code, replace the suffix to the username ("myorg.com") with *@fineapps.com*.

6. Click **Save**.

Configure Your Salesforce Community

For the final step, configure the community to use your Salesforce Auth. Provider for logins.

1. In Setup, go to **Customize** > **Communities** > **Manage Communities**.
2. Click **Edit** next to your community name.
3. Click **Login Page**.
4. Under Options for External Users, select your new Auth. Provider.
5. Click **Save**.

You're done! Now, when you log into your mobile app using your community login URL, look for an additional button inviting you to log in using Facebook. Click the button and follow the on-screen instructions to see how the login works.

To test the external authentication setup in a browser, customize the Single Sign-On Initialization URL (from your Auth. Provider) as follows:

```
https://login.salesforce.com/services/auth/sso/orgID/
URLsuffix?community=<community_login_url>
```

For example:

```
https://login.salesforce.com/services/auth/sso/00Da0000000TPNEAA4/
FB_Community_Login?community=
https://mobilesdk-developer-edition.server_instance.force.com/fineapps
```

To form the Existing User Linking URL, replace `sso` with `link`:

```
https://login.salesforce.com/services/auth/link/00Da0000000TPNEAA4/
FB_Community_Login?community=
https://mobilesdk-developer-edition.server_instance.force.com/fineapps
```

CHAPTER 11 Multi-User Support in Mobile SDK

In this chapter ...

If you need to enable simultaneous logins for multiple users, Mobile SDK provides a basic implementation for user switching, plus APIs for Android, iOS, and hybrid apps.

Mobile SDK provides a default dialog box that lets the user select from authenticated accounts. Your app implements some means of launching the dialog box and calls the APIs that initiate the user switching workflow.

About Multi-User Support

Beginning in version 2.2, Mobile SDK supports simultaneous logins from multiple user accounts. These accounts can represent different users from the same organization, or different users on different organizations (such as production and sandbox, for instance.)

Once a user signs in, that user's credentials are saved to allow seamless switching between accounts, without the need to re-authenticate against the server. If you don't wish to support multiple logins, you don't have to change your app. Existing Mobile SDK APIs work as before in the single-user scenario.

Mobile SDK assumes that each user account is unrelated to any other authenticated user account. Accordingly, Mobile SDK isolates data associated with each account from that of all others, thus preventing the mixing of data between accounts. Data isolation protects `SharedPreferences` files, SmartStore databases, `AccountManager` data, and any other flat files associated with an account.

Example: The following Mobile SDK sample apps demonstrate multi-user switching:

- **Android native (without SmartStore):** `RestExplorer`
- **Android native (with SmartStore):** `NativeSqlAggregator`
- **iOS native:** `RestAPIExplorer`
- **iOS hybrid:** `ContactExplorer`
- **Hybrid (without SmartStore):** `ContactExplorer`
- **Hybrid (with SmartStore):** `AccountEditor`

Implementing Multi-User Support

Mobile SDK provides APIs for enabling multi-user support in native Android, native iOS, and hybrid apps.

Although Mobile SDK implements the underlying functionality, multi-user switching isn't initialized at runtime unless and until your app calls an API that switches to a different user. APIs that switch users are:

Android native (`UserAccountManager` class methods)

```
public void switchToUser(UserAccount user)
public void switchToNewUser()
```

iOS native (`SFUserAccountManager` class methods)

```
- (void)switchToUser:(SFUserAccount *)newCurrentUser
- (void)switchToNewUser
```

Hybrid (JavaScript method)

```
switchToUser
```

To let the user switch to a different account, launch a selection screen from a button, menu, or some other control in your user interface. Mobile SDK provides a standard multi-user switching screen that displays all currently authenticated accounts in a radio button list. You can choose whether to customize this screen or just show the default version. When the user makes a selection, call the Mobile SDK method that launches the multi-user flow.

Before you begin to use the APIs, it's important that you understand the division of labor between Mobile SDK and your app. The following lists show tasks that Mobile SDK performs versus tasks that your app is required to perform in multi-user contexts. In particular, consider how to manage:

- Push Notifications (if your app supports them)
- SmartStore Soups (if your app uses SmartStore)
- Account Management

Push Notifications Tasks

Mobile SDK (for all accounts):

- Registers push notifications at login
- Unregisters push notifications at logout
- Delivers push notifications

Your app:

- Differentiates notifications according to the target user account
- Launches the correct user context to display each notification

SmartStore Tasks

Mobile SDK (for all accounts):

- Creates a separate SmartStore database for each authenticated user account
- Switches to the correct backing database each time a user switch occurs

Your app:

- Refreshes its cached credentials, such as instances of SmartStore held in memory, after every user switch or logout

Account Management Tasks

Mobile SDK (for all accounts):

- Loads the correct account credentials every time a user switch occurs

Your app:

- Refreshes its cached credentials, such as instances of `RestClient` held in memory, after every user switch or logout

Android Native APIs

Native classes in Mobile SDK for Android do most of the work for multi-user support. Your app makes a few simple calls and handles any data cached in memory. You also have the option of customizing the user switching activity.

To support user switching, Mobile SDK for Android defines native classes in the `com.salesforce.androidsdk.accounts`, `com.salesforce.androidsdk.ui`, and `com.salesforce.androidsdk.util` packages. Classes in the `com.salesforce.androidsdk.accounts` package include:

- `UserAccount`
- `UserAccountManager`

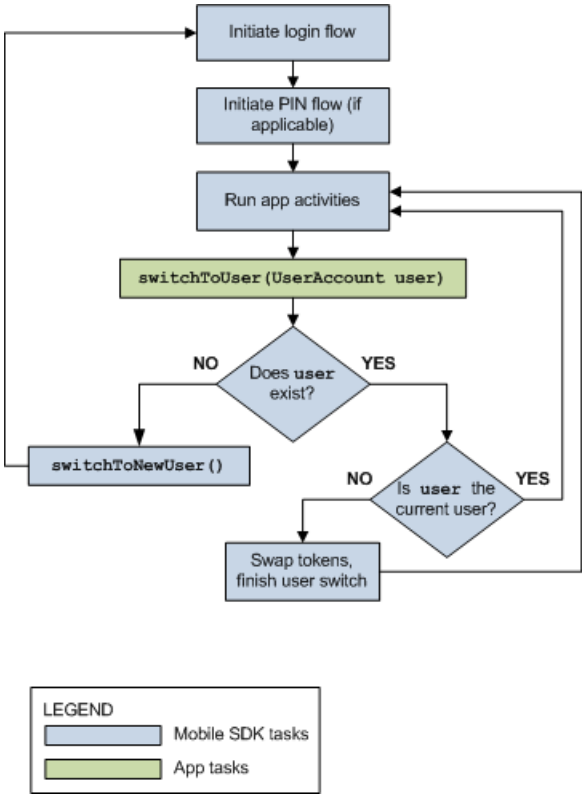
The `com.salesforce.androidsdk.ui` package contains the `AccountSwitcherActivity` class. You can extend this class to add advanced customizations to the account switcher activity.

The `com.salesforce.androidsdk.util` package contains the `UserSwitchReceiver` abstract class. You must implement this class if your app caches data other than tokens.

The following sections briefly describe these classes. For full API reference documentation, see http://forcedotcom.github.io/SalesforceMobileSDK-Android/index.html.

Multi-User Flow

For native Android apps, the `UserAccountManager.switchToUser()` Mobile SDK method launches the multi-user flow. Once your app calls this method, the Mobile SDK core handles the execution flow through all possible paths. The following diagram illustrates this flow.



IN THIS SECTION:

UserAccount Class

The `UserAccount` class represents a single user account that is currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

UserAccountManager Class

The `UserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out existing accounts, and switch between existing accounts.

AccountSwitcherActivity Class

Use or extend the `AccountSwitcherActivity` class to display the user switching interface.

UserSwitchReceiver Class

If your native Android app caches data other than tokens, implement the `UserSwitchReceiver` abstract class to receive notifications of user switching events.

UserAccount Class

The `UserAccount` class represents a single user account that is currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

Constructors

You can create `UserAccount` objects directly, from a JSON object, or from a bundle.

Constructor	Description
`public UserAccount(String authToken, String refreshToken, String loginServer, String idUrl, String instanceServer, String orgId, String userId, String username, String accountName, String clientId, String communityId, String communityUrl )`	Creates a `UserAccount` object using values you specify.
`public UserAccount(JSONObject object)`	Creates a `UserAccount` object from a JSON string.
`public UserAccount(Bundle bundle)`	Creates a `UserAccount` object from an Android application bundle.

Methods

Method	Description
`public String getOrgLevelStoragePath()`	Returns the organization level storage path for this user account, relative to the higher level directory of app data. The higher level directory could be `files`. The output is in the format `/{orgID}/`. This storage path is meant for data that can be shared across multiple users of the same organization.
`public String getUserLevelStoragePath()`	Returns the user level storage path for this user account, relative to the higher level directory of app data. The higher level directory could be `files`. The output is in the format `/{orgID}/{userID}/`. This storage path is meant for data that is unique to a particular user in an organization, but common across all the communities that the user is a member of within that organization.
`public String getCommunityLevelStoragePath(String communityId)`	Returns the community level storage path for this user account, relative to the higher level directory of app data. The higher level directory could be `files`. The output is in the format `/{orgID}/{userID}/{communityID}/`. If `communityID` is null, then the output would be `/{orgID}/{userID}/internal/`. This storage path is

Method	Description
	meant for data that is unique to a particular user in a specific community.
`public String getOrgLevelFilenameSuffix()`	Returns a unique suffix for this user account, that can be appended to a file to uniquely identify this account, at an organization level. The output is in the format `_{orgID}`. This suffix is meant for data that can be shared across multiple users of the same organization.
`public String getUserLevelFilenameSuffix()`	Returns a unique suffix for this user account, that can be appended to a file to uniquely identify this account, at a user level. The output is in the format `_{orgID}_{userID}`. This suffix is meant for data that is unique to a particular user in an organization, but common across all the communities that the user is a member of within that organization.
`public String getCommunityLevelFilenameSuffix(String communityId)`	Returns a unique suffix for this user account, that can be appended to a file to uniquely identify this account, at a community level. The output is in the format `_{orgID}_{userID}_{communityID}`. If `communityID` is null, then the output would be `_{orgID}_{userID}_internal`. This suffix is meant for data that is unique to a particular user in a specific community.

UserAccountManager Class

The `UserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out existing accounts, and switch between existing accounts.

You don't directly create instances of `UserAccountManager`. Instead, obtain an instance using the following call:

```
SalesforceSDKManager.getInstance().getUserAccountManager();
```

Methods

Method	Description
`public UserAccount getCurrentUser()`	Returns the currently active user account.
`public List<UserAccount> getAuthenticatedUsers()`	Returns the list of authenticated user accounts.
`public boolean doesUserAccountExist(UserAccount account)`	Checks whether the specified user account is already authenticated.
`public void switchToUser(UserAccount user)`	Switches the application context to the specified user account. If the specified user account is invalid or null, this method launches the login flow.
`public void switchToNewUser()`	Launches the login flow for a new user to log in.

Method	Description
`public void signoutUser(UserAccount userAccount, Activity frontActivity)`	Logs the specified user out of the application and wipes the specified user's credentials.

AccountSwitcherActivity Class

Use or extend the `AccountSwitcherActivity` class to display the user switching interface.

The `AccountSwitcherActivity` class provides the screen that handles multi-user logins. It displays a list of existing user accounts and lets the user switch between existing accounts or sign into a new account. To enable multi-user logins, launch the activity from somewhere in your app using the following code:

```
final Intent i = new Intent(this,
SalesforceSDKManager.getInstance().getAccountSwitcherActivityClass());
i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
this.startActivity(i);
```

For instance, you might launch this activity from a "Switch User" button in your user interface. See `SampleApps/RestExplorer` for an example.

If you like, you can customize and stylize `AccountSwitcherActivity` through XML.

For more control, you can extend `AccountSwitcherActivity` and replace it with your own custom sub-class. To replace the default class, call `SalesforceSDKManager.setAccountSwitcherActivityClass()`. Pass in a reference to the class file of your replacement activity class, such as `AccountSwitcherActivity.class`.

UserSwitchReceiver Class

If your native Android app caches data other than tokens, implement the `UserSwitchReceiver` abstract class to receive notifications of user switching events.

Every time a user switch occurs, Mobile SDK broadcasts an intent. The intent action is declared in the `UserAccountManager` class as:

```
public static final String USER_SWITCH_INTENT_ACTION = "com.salesforce.USERSWITCHED";
```

This broadcast event gives applications a chance to properly refresh their cached resources to accommodate user switching. To help apps listen for this event, Mobile SDK provides the `UserSwitchReceiver` abstract class. This class is implemented in the following Salesforce activity classes:

- `SalesforceActivity`
- `SalesforceListActivity`
- `SalesforceExpandableListActivity`

If your main activity extends one of the Salesforce activity classes, you don't need to implement `UserSwitchReceiver`.

If you've cached only tokens in memory, you don't need to do anything—Mobile SDK automatically refreshes tokens.

If you've cached user data other than tokens, override your activity's `refreshIfUserSwitched()` method with your custom refresh actions.

If your main activity does not extend one of the Salesforce activity classes, implement `UserSwitchReceiver` to handle cached data during user switching.

To set up the broadcast receiver:

1. Implement a subclass of `UserSwitchReceiver`.

2. Register your subclass as a receiver in your activity's `onCreate()` method.
3. Unregister your receiver in your activity's `onDestroy()` method.

For an example, see the `ExplorerActivity` class in the `RestExplorer` sample application.

If your application is a hybrid application, no action is required.

The `SalesforceDroidGapActivity` class refreshes the cache as needed when a user switch occurs.

Methods

A single method requires implementation.

Method Name	Description
`protected abstract void onUserSwitch();`	Implement this method to handle cached user data (other than tokens) when user switching occurs.

iOS Native APIs

Native classes in Mobile SDK for iOS do most of the work for multi-user support. Your app makes a few simple calls and handles any data cached in memory. You also have the option of customizing the user switching activity.

To support user switching, Mobile SDK for iOS defines native classes in the `Security` folder of the `SalesforceSDKCore` library. Classes include:

- `SFUserAccount`
- `SFUserAccountManager`

The following sections briefly describe these classes. For full API reference documentation, see http://forcedotcom.github.io/SalesforceMobileSDK-iOS/Documentation/SalesforceSDKCore/html/index.html.

IN THIS SECTION:

SFUserAccount Class

The `SFUserAccount` class represents a single user account that's currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

SFUserAccountManager Class

The `SFUserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out accounts, and switch between accounts.

SFUserAccount Class

The `SFUserAccount` class represents a single user account that's currently authenticated. It encapsulates data that can be used to uniquely identify a user account.

Properties

You can create `SFUserAccount` objects directly, from a JSON object, or from a bundle.

Property	Description
`@property (nonatomic, copy) NSSet *accessScopes`	The access scopes for this user.
`@property (nonatomic, strong) SFOAuthCredentials *credentials;`	The credentials that are associated with this user.
`@property (nonatomic, strong) SFIdentityData *idData;`	The identity data that's associated with this user.
`@property (nonatomic, copy, readonly) NSURL *apiUrl;`	The URL that can be used to invoke any API on the server side. This URL takes into account the current community if available.
`@property (nonatomic, copy) NSString *email;`	The user's email address.
`@property (nonatomic, copy) NSString *organizationName;`	The name of the user's organization.
`@property (nonatomic, copy) NSString *fullName;`	The user's first and last names.
`@property (nonatomic, copy) NSString *userName;`	The user's username.
`@property (nonatomic, strong) UIImage *photo;`	The user's photo, typically a thumbnail of the user. The consumer of this class must set this property at least once in order to use the photo. This class doesn't fetch the photo from the server; it stores and retrieves the photo locally.
`@property (nonatomic) SFUserAccountAccessRestriction accessRestrictions;`	The access restrictions that are associated with this user.
`@property (nonatomic, copy) NSString *communityId;`	The current community ID, if the user is logged into a community. Otherwise, this property is nil.
`@property (nonatomic, readonly, getter = isSessionValid) BOOL sessionValid;`	Returns YES if the user has an access token and, presumably, a valid session.
`@property (nonatomic, copy) NSDictionary *customData;`	The custom data for the user. Because this data can be serialized, the objects that are contained in `customData` must follow the `NSCoding` protocol.

Global Function

Function Name	Description
`NSString *SFKeyForUserAndScope (SFUserAccount *user, SFUserAccountScope scope);`	Returns a key that uniquely identifies this user account for the given scope. If you set `scope` to `SFUserAccountScopeGlobal`, the same key will be returned regardless of the user account.

SFUserAccountManager Class

The `SFUserAccountManager` class provides methods to access authenticated accounts, add new accounts, log out accounts, and switch between accounts.

To access the singleton `SFUserAccountManager` instance, send the following message:

```
[SFUserAccountManager sharedInstance]
```

Properties

Property	Description
`@property (nonatomic, strong) SFUserAccount *currentUser`	The current user account. If the user has never logged in, this property may be nil.
`@property (nonatomic, readonly) NSString *currentUserId`	A convenience property to retrieve the current user's ID. This property is an alias for `currentUser.credentials.userId`.
`@property (nonatomic, readonly) NSString *currentCommunityId`	A convenience property to retrieve the current user's community ID. This property is an alias for `currentUser.communityId`.
`@property (nonatomic, readonly) NSArray *allUserAccounts`	An `NSArray` of all the `SFUserAccount` instances for the app.
`@property (nonatomic, readonly) NSArray *allUserIds`	Returns an array that contains all user IDs.
`@property (nonatomic, copy) NSString *activeUserId`	The most recently active user ID. If the user that's specified by `activeUserId` is removed from the accounts list, this user may be temporarily different from the current user.
`@property (nonatomic, strong) NSString *loginHost`	The host to be used for login.
`@property (nonatomic, assign) BOOL retryLoginAfterFailure`	A flag that controls whether the login process restarts after it fails. The default value is YES.
`@property (nonatomic, copy) NSString *oauthClientId`	The OAuth client ID to use for login. Apps may customize this property before login. Otherwise, this value is determined by the `SFDCOAuthClientIdPreference` property that's configured in the settings bundle.

Property	Description
`@property (nonatomic, copy) NSString *oauthCompletionUrl`	The OAuth callback URL to use for the OAuth login process. Apps may customize this property before login. By default, the property's value is copied from the `SFDCOAuthRedirectUri` property in the main bundle. The default value is `@"sfdc:///axm/detect/oauth/done"`.
`@property (nonatomic, copy) NSSet *scopes`	The OAuth scopes that are associated with the app.

Methods

Method	Description
`- (NSString*)userAccountPlistFileForUser:(SFUserAccount*)user`	Returns the path of the `.plist` file for the specified user account.
`- (void)addDelegate:(id<SFUserAccountManagerDelegate>)delegate`	Adds a delegate to this user account manager.
`- (void)removeDelegate:(id<SFUserAccountManagerDelegate>)delegate`	Removes a delegate from this user account manager.
`- (SFLoginHostUpdateResult *)updateLoginHost`	Sets the app-level login host to the value in app settings.
`- (BOOL)loadAccounts:(NSError**)error`	Loads all accounts.
`- (BOOL)saveAccounts:(NSError**)error`	Saves all accounts.
`- (SFUserAccount*)createUserAccount`	Can be used to create an empty user account if you want to configure all of the account information yourself. Otherwise, use `[SFAuthenticationManager loginWithCompletion:failure:]` to automatically create an account when necessary.
`- (SFUserAccount*)userAccountForUserId:(NSString*)userId`	Returns the user account that's associated with a given user ID.
`- (NSArray *)accountsForOrgId:(NSString *)orgId`	Returns all accounts that have access to a particular organization.
`- (NSArray *)accountsForInstanceURL:(NSString *)instanceURL`	Returns all accounts that match a particular instance URL.
`- (void)addAccount:(SFUserAccount *)acct`	Adds a user account.
`- (BOOL)deleteAccountForUserId:(NSString*)userId error:(NSError **)error`	Removes the user account that's associated with the given user ID.

Method	Description
`- (void)clearAllAccountState`	Clears the account's state in memory (but doesn't change anything on the disk).
`- (void)applyCredentials:(SFOAuthCredentials*)credentials`	Applies the specified credentials to the current user. If no user exists, a user is created.
`- (void)applyCustomDataToCurrentUser:(NSDictionary *)customData`	Applies custom data to the `SFUserAccount` that can be accessed outside that user's sandbox. This data persists between app launches. Because this data will be serialized, make sure that objects that are contained in `customData` follow the `NSCoding` protocol. **Important:** Use this method only for nonsensitive information.
`- (void)switchToNewUser`	Switches from the current user to a new user context.
`- (void)switchToUser:(SFUserAccount *)newCurrentUser`	Switches from the current user to the specified user account.
`- (void)userChanged:(SFUserAccountChange)change`	Informs the `SFUserAccountManager` object that something has changed for the current user.

Hybrid APIs

Hybrid apps can enable multi-user support through Mobile SDK JavaScript APIs. These APIs reside in the `SFAccountManagerPlugin` Cordova-based module.

SFAccountManagerPlugin Methods

Before you call any of these methods, you need to load the `sfaccountmanager` plugin. For example:

```
cordova.require("com.salesforce.plugin.sfaccountmanager").logout();
```

Method Name	Description
`getUsers`	Returns the list of users already logged in.
`getCurrentUser`	Returns the current active user.

Method Name	Description
`logout`	Logs out the specified user if a user is passed in, or the current user if called with no arguments.
`switchToUser`	Switches the application context to the specified user, or launches the account switching screen if no user is specified.

Hybrid apps don't need to implement a receiver for the multi-user switching broadcast event. This handler is implemented by the `SalesforceDroidGapActivity` class.

CHAPTER 12 Migrating from the Previous Release

In this chapter ...

If you're upgrading an app built with Salesforce Mobile SDK 2.2, follow these instructions to update your app to version 2.3.

If you're upgrading an app that's built with a version earlier than Salesforce Mobile SDK, start upgrading with Migrating from Earlier Releases.

Migrate Android Native Apps from 3.1 to 3.2

Perform these steps to upgrade your Android native applications from Salesforce Mobile SDK 3.1 to version 3.2.

1. Open your Mobile SDK project workspace in Eclipse.
2. Replace the existing Cordova project with the Mobile SDK 3.1 Cordova project.
3. Replace the existing SalesforceSDK project with the new SalesforceSDK project.
4. If your app uses SmartStore, replace the existing SmartStore project with the new SmartStore project.
5. If your app uses SmartSync, replace the existing SmartSync project with the new SmartSync project.
6. In Project Explorer, RIGHT-CLICK your project and select **Properties**.
7. In the left panel, select **Android**.
8. In the Library section, replace the existing SalesforceSDK entry with the new SalesforceSDK project in your workspace.
9. If your app uses SmartStore, repeat step 8 for the SmartStore project.
10. If your app uses SmartSync, repeat step 8 for the SmartSync project.

Migrate Hybrid Apps from 3.1 to 3.2

The easiest way to upgrade native and hybrid iOS apps is to build a new app with the latest version of forceios. When the new app is ready, migrate your app's code into the new template. Doing this for hybrid apps ensures that your app's Cordova underpinnings match the current Mobile SDK Cordova plugin.

You don't need to modify your existing Web app code to upgrade from Mobile SDK 3.1 to Mobile SDK 3.2. You simply upgrade the Salesforce Cordova plugin. Mobile SDK 3.2 supports Cordova 3.6.x (3.6.3 for iOS, 3.6.4 for Android) or later, and is expected to work with Cordova 3.7.

To upgrade the Salesforce Cordova plugin, use the Cordova command-line tool to remove and then readd the plugin, as shown here:

Example:

```
$ cd <your_Cordova_app_folder>
$ cordova plugin rm com.salesforce
$ cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
$ cordova prepare
```

Migrate iOS Native Apps from 3.1 to 3.2

Migrating to Mobile SDK 3.2 requires minor effort. As in the previous release, the minimum supported iOS version is 7.0, and the minimum supported Xcode version is 6.0. We do not guarantee backwards compatibility for earlier versions of iOS or Xcode.

In Mobile SDK 3.2, we've replaced the MKNetworkKit and SalesforceNetworkSDK networking libraries with the SalesforceNetwork library. If your app calls MKNetworkKit APIs directly, replace those calls with calls to equivalent API in the SalesforceNetwork library.

Perform the steps in Update Mobile SDK Library Packages to upgrade a Mobile SDK 3.1 app to Mobile SDK 3.2.

Update Mobile SDK Library Packages

The easiest way to upgrade native and hybrid iOS apps is to build a new app with the latest version of forceios. When the new app is ready, migrate your app's code into the new template. If you are instead manually updating the Mobile SDK artifacts in your existing app, use the following instructions. If you're managing your app's Mobile SDK dependencies with CocoaPods, you don't need to follow these instructions.

To update Mobile SDK library packages, delete the existing Dependencies folder of your app's Xcode project, and then add the new libraries in a re-created Dependencies folder.

1. Download the following binary packages from the SalesforceMobileSDK-iOS-Distribution repo (https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Distribution):
   - `SalesforceRestAPI-Release.zip`
   - `SalesforceNetwork-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCore-Release.zip`
   - `SalesforceSecurity-Release.zip`
   - `SmartSync-Release.zip`
   - `SalesforceSDKCommon-Release.zip`
2. Download the following folders from the ThirdParty folder link in the distribution repo:
   - `SalesforceCommonUtils`
   - `openssl`
   - `sqlcipher`
3. Open your Mobile SDK project in Xcode.
4. In Project Navigator, locate the Dependencies folder.
5. CONTROL+CLICK the folder. Choose **Delete**, and select **Move to Trash**.
6. Re-create the Dependencies folder in your Xcode project, under your app folder.
7. Unzip the new packages from step 1, and copy the folders from step 2, into the Dependencies folder.
8. In Project Navigator, CONTROL+CLICK your app folder and select **Add Files to "<*App Name*>"...**.
9. Select the Dependencies folder, making sure that **Create groups** is selected for Added Folders.
10. Click **Add**.

Migrating from Earlier Releases

To migrate from versions older than the previous release, perform the code upgrade steps for each intervening release, starting at your current version.

Migrate Hybrid Apps from 3.0 to 3.1

Existing Mobile SDK 3.0 hybrid apps work without code modifications in Mobile SDK 3.1. You simply upgrade the Salesforce Cordova plugin. Mobile SDK 3.1 supports Cordova 3.5 or later, has been tested through Cordova 3.6.3, and is expected to work with Cordova 3.7.

To upgrade the Salesforce Cordova plugin, use the Cordova command-line tool to remove and then readd the plugin, as shown here:

Example:

```
$ cd <your_Cordova_app_folder>
$ cordova plugin rm com.salesforce
$ cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
$ cordova prepare
```

Migrate Android Native Apps from 3.0 to 3.1

Perform these steps to upgrade your Android native applications from Salesforce Mobile SDK 3.0 to version 3.1.

1. Open your Mobile SDK project workspace in Eclipse.
2. Replace the existing Cordova project with the Mobile SDK 3.1 Cordova project.
3. Replace the existing SalesforceSDK project with the new SalesforceSDK project.
4. If your app uses SmartStore, replace the existing SmartStore project with the new SmartStore project.
5. If your app uses SmartSync, replace the existing SmartSync project with the new SmartSync project.
6. In Project Explorer, RIGHT-CLICK your project and select **Properties**.
7. In the left panel, select **Android**.
8. In the Library section, replace the existing SalesforceSDK entry with the new SalesforceSDK project in your workspace.
9. If your app uses SmartStore, repeat step 8 for the SmartStore project.
10. If your app uses SmartSync, repeat step 8 for the SmartSync project.

Migrate iOS Native Apps from 3.0 to 3.1

Migrating to Mobile SDK 3.1 requires little effort. The minimum supported Xcode version is now 6.0. Also, in addition to updating the existing binary packages, we've added a new one—SalesforceSDKCommon (`SalesforceSDKCommon-[Debug/Release].zip`). This package contains low-level network and security utilities.

Perform the steps in Update Mobile SDK Library Packages from 3.0 to 3.1 to upgrade a Mobile SDK 3.0 app to Mobile SDK 3.1.

Update Mobile SDK Library Packages from 3.0 to 3.1

To update the library packages, delete and re-create the Dependencies folder of your app's Xcode project, and then add the new libraries to it.

1. Download the following binary packages from the SalesforceMobileSDK-iOS-Distribution repo (https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Distribution):
   - `MKNetworkKit-iOS-Release.zip`
   - `SalesforceRestAPI-Release.zip`
   - `SalesforceNetworkSDK-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCommon-Release.zip`
   - `SalesforceSDKCore-Release.zip`
   - `SalesforceSecurity-Release.zip`
   - `SmartSync-Release.zip`

2. Download the following folders from the ThirdParty folder link in the distribution repo:
   - `SalesforceCommonUtils`
   - `openssl`
   - `sqlcipher`
3. Open your Mobile SDKproject in Xcode.
4. In Project Navigator, locate the Dependencies folder.
5. CONTROL+CLICK the folder. Choose **Delete**, and select **Move to Trash**.
6. Re-create the Dependencies folder in your Xcode project, under your app folder.
7. Unzip the new packages from step 1, and copy the folders from step 2, into the Dependencies folder.
8. In Project Navigator, CONTROL+CLICK your app folder and select **Add Files to "<*App Name*>"...**.
9. Select the Dependencies folder, making sure that **Create groups** is selected for Added Folders.
10. Click **Add**.

Migrate Hybrid Applications from 2.3 to 3.0

Existing hybrid apps should continue to work without modification in Mobile SDK 3.0. Mobile SDK 3.0 supports Cordova 3.5 or later, has been tested through Cordova 3.6.3, and is expected to work with Cordova 3.7.

Upgrading your hybrid app from 2.3 to 3.0 is as simple as upgrading the Salesforce Cordova plugins. To do this, use the Cordova command-line tool to remove, then re-add the plugin as shown here:

```
$ cd <your_Cordova_app_folder>
$ cordova plugin rm com.salesforce
$ cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
$ cordova prepare
```

Migrate Android Native Apps from 2.3 to 3.0

Perform these steps to upgrade your Android native applications from Salesforce Mobile SDK 2.3 to version 3.0.

1. Open your Mobile SDK project workspace in Eclipse.
2. Replace the existing Cordova project with the Mobile SDK 3.0 Cordova project.
3. Replace the existing SalesforceSDK project with the new SalesforceSDK project.
4. If your app uses SmartStore, replace the existing SmartStore project with the new SmartStore project.
5. In Project Explorer, RIGHT-CLICK your project and select **Properties**.
6. In the left panel, select **Android**.
7. In the Library section, replace the existing SalesforceSDK entry with the new SalesforceSDK project in your workspace.
8. If your app uses SmartStore, repeat the previous step for the SmartStore project.
9. Ensure that the minimum Android SDK version used by your app is level 17 or higher.

Migrate iOS Native Apps from 2.3 to 3.0

To migrate iOS native apps from Mobile SDK 2.3 to version 3.0, follow this two-step process:

1. Update Mobile SDK Library Packages
2. Update App Bootstrap Flow to Use SalesforceSDKManager

Update Mobile SDK Library Packages

The easiest way to update the library packages is to delete the existing Dependencies folder of your app's Xcode project, and then add the new libraries in a re-created Dependencies folder.

1. Download the following binary packages from the SalesforceMobileSDK-iOS-Distribution repo (https://github.com/forcedotcom/SalesforceMobileSDK-iOS-Distribution):
   - `MKNetworkKit-iOS-Release.zip`
   - `SalesforceRestAPI-Release.zip`
   - `SalesforceNetworkSDK-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCore-Release.zip`
   - `SalesforceSecurity-Release.zip`
   - `SmartSync-Release.zip`
2. Download the following folders from the ThirdParty folder link in the distribution repo:
   - `SalesforceCommonUtils`
   - `openssl`
   - `sqlcipher`
3. Open your Mobile SDKproject in Xcode.
4. In Project Navigator, locate the Dependencies folder.
5. CONTROL+CLICK the folder. Choose **Delete**, and select **Move to Trash**.
6. Re-create the Dependencies folder in your Xcode project, under your app folder.
7. Unzip the new packages from step 1, and copy the folders from step 2, into the Dependencies folder.
8. In Project Navigator, CONTROL+CLICK your app folder and select **Add Files to "<*App Name*>"...**.
9. Select the Dependencies folder, making sure that **Create groups** is selected for Added Folders.
10. Click **Add**.

Update App Bootstrap Flow to Use SalesforceSDKManager

Starting with the Mobile SDK 3.0, much of the Mobile SDK bootstrapping process moves into the `SalesforceSDKManager` singleton class. While you can reuse most custom code that handles launch events, you'll have to move it to slightly different contexts. The following list describes important moves and changes to bootstrapping and configuration code.

1. Update the code that configures your app's Connected App settings and OAuth scopes.
   a. Replace `[SFUserAccountManager sharedInstance].oauthClientId` with `[SalesforceSDKManager sharedManager].connectedAppId`.
   b. Replace `[SFUserAccountManager sharedInstance].oauthCompletionUrl` with `[SalesforceSDKManager sharedManager].connectedAppCallbackUri`.

   c. Replace `[SFUserAccountManager sharedInstance].scopes` with `[SalesforceSDKManager sharedManager].authScopes`.

2. If your app authenticates at the beginning of your app launch process (the default behavior), replace your call to `[[SFAuthenticationManager sharedManager] loginWithCompletion:failure:]` as follows:
   a. Replace your completion block by setting `[SalesforceSDKManager sharedManager].postLaunchAction`.
   b. Replace your failure block by setting `[SalesforceSDKManager sharedManager].launchErrorAction`.
   c. Replace your call to `loginWithCompletion:failure:` with `[[SalesforceSDKManager sharedManager] launch]`.
3. If your app does not authenticate as part of your app's launch process, do the the following:
   a. Set `[SalesforceSDKManager sharedManager].authenticateAtLaunch` to `NO` somewhere before calling launch.
   b. Continue to call `[[SFAuthenticationManager sharedManager] loginWithCompletion:failure:]` at the appropriate time in your app's lifecycle.
4. Regardless of whether your app authenticates at app startup or not, your `AppDelegate` implementation must call `[[SalesforceSDKManager sharedManager] launch]` in `application:didFinishLaunchingWithOptions:`.
5. Your `AppDelegate` class no longer needs to implement the `SFAuthenticationManagerDelegate` or `SFUserAccountManagerDelegate` protocols for bootstrapping events.
   - The `[SalesforceSDKManager sharedManager].postLogoutAction` block replaces `[SFAuthenticationManagerDelegate authManagerDidLogout:]`.
   - The `[SalesforceSDKManager sharedManager].switchUserAction` block replaces `[SFUserAccountManagerDelegate userAccountManager:didSwitchFromUser:toUser:]`.
6. If you subscribed to the following `SFAuthenticationManagerDelegate` methods for app event boundaries, you must now subscribe to the equivalent delegate methods of `SalesforceSDKManagerDelegate`:
   - `authManagerWillResignActive:`
   - `authManagerWillEnterForeground:`
   - `authManagerDidBecomeActive:`
   - `authManagerDidEnterBackground:`
7. If you customized the snapshot view functionality of `SFAuthenticationManager` (`useSnapshotView`, `snapshotView`), move those customizations to the equivalent functionality in `SalesforceSDKManager`.
8. (Uncommon) If you provided an override for the default `preferredPasscodeProvider` value in `SFAuthenticationManager`, move your customizations to `SalesforceSDKManager`.

See Developing a Native iOS App on page 24 for a detailed look at how `SalesforceSDKManager` impacts the Mobile SDK bootstrapping process of your app.

Migrating Hybrid Applications from 2.2 to 2.3

Mobile SDK 2.3 focuses on upgrading the hybrid container to Cordova 3.5. All migration requirements for this version of Mobile SDK pertain to hybrid apps. Native iOS and Android apps built with version 2.2 require no changes to run in 2.3.

Beginning with version 2.3, Mobile SDK uses Cordova 3.5 or higher as a completely independent module for creating hybrid containers. This new architecture requires you to use the Cordova command line directly. To migrate an existing hybrid application to Mobile SDK 2.3, you create a new Cordova app, then move your existing Web assets into it. You can do this two ways:

- Create a new forceios or forcedroid app, move your Web assets into it, then use Cordova to finish configuring the app. See Create Hybrid Apps on page 123.
- Create a Cordova app, then add the Salesforce Mobile SDK Cordova plugin to it along with your other plugins and Web assets. This slightly simpler method is described in the following sections

Create a Cordova App

Run the following commands in a Mac OS X Terminal window or Windows Command Prompt.

1. Install Cordova 3.5 (or higher):

   ```
   npm -g install cordova
   ```

   Or, if you've already installed Cordova, make sure it's version 3.5 or higher:

   ```
   cordova --version
   ```

2. Create a Cordova-friendly application directory:

   ```
   cordova create MyNewApplication
   ```

   `cordova create` takes three arguments, but only the first one (<*directory*>) is required. You can also specify a package name, such as `com.acme.hello`, and an application display title. In the single-parameter example shown here, Cordova creates a package named `io.cordova.hellocordova` and uses "HelloCordova" for the display name. See "The Command Line Interface" at http://cordova.apache.org/docs/en/3.5.0.

3. Go into the new application's directory:

   ```
   cd MyNewApplication
   ```

4. If your app supports iOS:

   ```
   cordova platform add ios
   ```

5. If your app supports Android:

   ```
   cordova platform add android
   ```

6. Add the Salesforce Mobile SDK plugins:

   ```
   cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
   ```

7. Plugins are no longer bundled with Cordova. If your app depends on any of the following plugins, call the `cordova plugin add` command for each dependency:
   - org.apache.cordova.battery-status
   - org.apache.cordova.camera
   - org.apache.cordova.console
   - org.apache.cordova.contacts
   - org.apache.cordova.device-motion
   - org.apache.cordova.device-orientation

- org.apache.cordova.dialogs
- org.apache.cordova.file
- org.apache.cordova.file-transfer
- org.apache.cordova.geolocation
- org.apache.cordova.globalization
- org.apache.cordova.inappbrowser
- org.apache.cordova.media
- org.apache.cordova.media-capture
- org.apache.cordova.network-information
- org.apache.cordova.splashscreen
- org.apache.cordova.statusbar
- org.apache.cordova.vibration

For example:

```
cordova plugin add org.apache.cordova.camera
```

8. Move your app's HTML, JavaScript, and CSS web assets, as well as the `bootconfig.json` file, to your new app's `assets/www/` directory.
9. Deploy the web assets to the platforms specific folders:

```
cordova prepare
```

HTML and JavaScript Code Changes

When you create a project with `cordova create MyProject`, Cordova creates the following directories:

- `MyProject/www`--Contains your app's HTML, JavaScript, and CSS code
- `MyProject/platforms`--Contains platform-specific projects in subdirectories
- `MyProject/plugins`--Contains plugins

Be careful to avoid putting `cordova.js`, `cordova.force.js`, or any other Cordova plugins in your app's `www/` directory. Cordova automatically deposits these plugins into your platform-specific project directories when you run `cordova prepare`. You do app development work in the `www/` directory.

Upgrade Steps

1. In your HTML code, include `cordova.js` in a `<script>` tag. Be sure to include vanilla `cordova.js`, rather than any flavored "cordova-*xyz*.js" file:

```
<script src="cordova.js"></script>
```

2. At runtime, Cordova automatically injects any code that you import through `cordova.require()`. The Cordova tool generates a `cordova_plugins.js` file that maps required modules to files and injects the necessary `<script>` inclusion tags. To synchronize with the runtime injection of JavaScript plugins, do not call `cordova.require()` until you have received the `deviceready` event notification.
3. We've changed the naming convention for Salesforce plugins. Update your `cordova.require()` calls to reflect the following new paths:

Old Convention	New Convention
salesforce/util/<util_class>	com.salesforce.util.<util_class>
com.salesforce.plugin.<plugin_name>	com.salesforce.plugin.<plugin_name>

Example: Replace the old statements with the new, as shown in the following table:

Old Syntax	New Syntax
`cordova.require("salesforce/util/logger")`	`cordova.require("com.salesforce.util.logger")`
`cordova.require("salesforce/util/bootstrap")`	`cordova.require("com.salesforce.util.bootstrap")`
`cordova.require("salesforce/util/event")`	`cordova.require("com.salesforce.util.event")`
`cordova.require("salesforce/util/exec")`	`cordova.require("com.salesforce.util.exec")`
`cordova.require("salesforce/util/push")`	`cordova.require("com.salesforce.util.push")`
`cordova.require("salesforce/plugin/oauth")`	`cordova.require("com.salesforce.plugin.oauth")`
`cordova.require("salesforce/plugin/smartstore")`	`cordova.require("com.salesforce.plugin.smartstore")`
`cordova.require("salesforce/plugin/sdkinfo")`	`cordova.require("com.salesforce.plugin.sdkinfo")`
`cordova.require("salesforce/plugin/sfdcaccountmanager")`	`cordova.require("com.salesforce.plugin.sfdcaccountmanager")`

Migrate Android Native Apps from 2.2 to 2.3

Perform these tasks to upgrade your Android native applications from Salesforce Mobile SDK 2.2 to version 2.3. In Mobile SDK 2.3, we've upgraded the Cordova module and changed how the SDK references it. Cordova is now a project that you must include in all Android projects, whether they're native or hybrid.

1. Pull the latest version from the master branch of the SalesforceMobileSDK-Android GitHub repository.
2. In Eclipse, open your Mobile SDK 2.2 project workspace.
3. Import the Cordova library project in Eclipse:
   a. Click **File** > **Import**.
   b. Expand the General tab and select **Existing Projects into Workspace**, then click **Next**.

c. Select the `SalesforceMobileSDK-Android` repository directory as your root.

d. Click **Deselect All**.

e. Select the Cordova project.

f. Click **Finish**.

4. Replace the existing SalesforceSDK project in Eclipse with the Mobile SDK 2.3 SalesforceSDK project.
5. If your app uses SmartStore, replace the existing SmartStore project in Eclipse with the Mobile SDK 2.3 SmartStore project.
6. Right-click your project and select **Properties**.
7. Select **Android**.
8. In the Library section, replace the existing SalesforceSDK entry with the Mobile SDK 2.3 SalesforceSDK project in your workspace.
9. If your app uses SmartStore, replace the existing SmartStore entry in the Library section with the Mobile SDK 2.3 SmartStore project in your workspace.

Migrate iOS Native Apps from 2.2 to 2.3

To migrate iOS native apps to Mobile SDK 2.3, you update the Mobile SDK library packages, and, if you use SmartStore, update the code that registers your soups.

The easiest way to update your libraries is to delete everything in the `Dependencies` folder of your app's Xcode project, and then add the new libraries.

1. Open your project in Xcode.
2. In Project Navigator, control-click the `Dependencies` folder.
3. Select **Delete**, and then select **Move to Trash**.
4. Locate your project folder in Finder and delete the `Dependencies` folder if it still exists.
5. Download the following binary packages from the forcedotcom/SalesforceMobileSDK-iOS-Distribution GitHub repo:
   - `MKNetworkKit-iOS-Release.zip`
   - `SalesforceNativeSDK-Release.zip`
   - `SalesforceNetworkSDK-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCore-Release.zip`
   - `SalesforceSecurity-Release.zip`
6. Download the following folders from the `ThirdParty` folder link in the distribution repo:
   - `SalesforceCommonUtils`
   - `openssl`
   - `sqlcipher`
7. In Finder, re-create the `Dependencies` folder under your app folder.
8. Unzip the new packages from step 2, and copy the folders from step 3, into the `Dependencies` folder.
9. In Project Navigator, control-click your app folder and select **Add Files to "<App Name>"...**.
10. Select the Dependencies folder, making sure that **Create groups for any added folder** is selected.
11. Click **Add**.

Now that the libraries are up-to-date, update your SmartStore code as described in Update SmartStore Index Specs Object.

Update SmartStore Index Specs Object

If your iOS native app uses SmartStore, change the object that contains your index specifications when you register a soup. Prior to Mobile SDK 2.3, index specs were collected in an `NSArray` of `NSDictionary` objects. As of Mobile SDK 2.3, the `registerSoup:withIndexSpecs:` method expects an array of `SFSoupIndex` objects.

Example: For code examples, see Registering a Soup or the NativeSqlAggregator sample app.

Migrate Mobile SDK Android Applications from 2.1 to 2.2

Perform these tasks to upgrade your Android applications (native or hybrid) from Salesforce Mobile SDK 2.1 to version 2.2.

1. Replace the existing SalesforceSDK project in Eclipse with the Mobile SDK 2.2 SalesforceSDK project.
2. If your app uses SmartStore, replace the existing SmartStore project in Eclipse with the Mobile SDK 2.2 SmartStore project.
3. Right-click your project and select **Properties**.
4. Select **Android**.
5. In the Library section, replace the existing SalesforceSDK entry with the Mobile SDK 2.2 SalesforceSDK project in your workspace.
6. If your app uses SmartStore, replace the existing SmartStore entry in the Library section with the Mobile SDK 2.2 SmartStore project in your workspace.

Migrate Mobile SDK iOS Applications From 2.1 to 2.2

Important: To upgrade native and hybrid apps, we strongly recommend you create a new app using the updated forceios npm package, then migrate your existing code and resources into the new app.

Perform the following manual steps only if you prefer to update the Mobile SDK artifacts in your existing app.

iOS Hybrid Applications

Update Mobile SDK Library Packages

The easiest way to upgrade Mobile SDK library packages is to delete the `Dependencies` folder of your app's Xcode project, and then add the new libraries.

1. In your Xcode project, in Project Navigator, locate the `Dependencies` folder. Control-click the folder, choose **Delete**, and select **Move to Trash**.
2. Download the following binary packages from the distribution repo:
   - `Cordova/Cordova-Release.zip`
   - `SalesforceHybridSDK-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCore-Release.zip`
   - `SalesforceSecurity-Release.zip`
3. Also, download the following folders from the `ThirdParty` folder link in the distribution repo:

- `SalesforceCommonUtils`
- `openssl`
- `sqlcipher`

4. Recreate the `Dependencies` folder under your app folder.
5. Unzip the new packages from step 2, and copy the folders from step 3, into the `Dependencies` folder.
6. In Project Navigator, control-click your app folder and select **Add Files to "<*app_name*>"**.
7. Select the `Dependencies` folder, making sure that **Create groups for any added folder** is selected.
8. Click **Add**.

Add a Search Path for the SalesforceSecurity Header File

Update the header file search paths of your Xcode project.

1. Select your project in Project Navigator.
2. Select the **Build Settings** tab of your main target.
3. Scroll down to (or search/filter for) **Header Search Paths**.
4. Add the following search path:
   - `$(SRCROOT)/[App Name]/Dependencies/SalesforceSecurity/Headers`

Update Hybrid Local Artifacts

1. For your hybrid local apps, replace the following files in the `www/` folder of your app with the new versions from the `libs` folder of the SalesforceMobileSDK-Shared repo:
   - `cordova.force.js`
   - `forcetk.mobilesdk.js`
   - `smartsync.js`

Update AppDelegate Implementation

Some user management APIs have changed, as well as the patterns for handling logout and login host change events. To see the changes, we recommend that you consult the `AppDelegate` code from the 2.2 version of a forceios hybrid app. Here's a high-level overview of what's changed.

- Notifications for logout events and login host changes have moved to delegate methods. Update your `AppDelegate` class to implement the `SFAuthenticationManagerDelegate` and `SFUserAccountManagerDelegate` delegates.
  - For user logout notifications, use `[SFAuthenticationManagerDelegate authManagerDidLogout:]`
  - For login host changes, use `[SFUserAccountManagerDelegate userAccountManager:didSwitchFromUser:toUser:]`.

    Note: Changing the login host in the Settings app effectively switches to a new user and requires a login.

iOS Native Applications

Update Mobile SDK Library Packages

The easiest way to upgrade Mobile SDK library packages is to delete the `Dependencies` folder of your app's Xcode project, and then add the new libraries.

1. In your Xcode project, in Project Navigator, locate the `Dependencies` folder. Control-click the folder, choose **Delete**, and select **Move to Trash**.
2. Download the following binary packages from the distribution repo:
   - `MKNetworkKit-iOS-Release.zip`
   - `SalesforceNetworkSDK-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCore-Release.zip`
   - `SalesforceSecurity-Release.zip`
3. Also, download the following folders from the `ThirdParty` folder link in the distribution repo:
   - `SalesforceCommonUtils`
   - `openssl`
   - `sqlcipher`
4. Recreate the `Dependencies` folder, under your app folder.
5. Unzip the new packages from step 2, and copy the folders from step 3, into the `Dependencies` folder.
6. In Project Navigator, control-click your app folder and select **Add Files to "<*app_name*>"**.
7. Select the `Dependencies` folder, making sure that **Create groups for any added folder** is selected.
8. Click **Add**.

Add a Search Path for the SalesforceSecurity Header File

Update your header file search paths to find the SalesforceSecurity project.

1. Select your project in Project Navigator.
2. Select the **Build Settings** tab of your main target.
3. Scroll down to (or search/filter for) **Header Search Paths**.
4. Add the following search path:
   - `$(SRCROOT)/[App Name]/Dependencies/SalesforceSecurity/Headers`

Migrate Passcode-Related Code to New SalesforceSecurity Classes

`SalesforceSecurity` is a new library in 2.2. Many security-related classes—particularly classes related to passcode management—move from `SalesforceSDKCore` into `SalesforceSecurity`. If your code references passcode-related functionality from `SalesforceSDKCore`, update that code to use the appropriate `SalesforceSecurity` counterparts.

Update AppDelegate Implementation

Some user management APIs have changed, as well as the patterns for handling logout and login host change events. To see the changes, we recommend that you consult the `AppDelegate` code from the 2.2 version of a forceios native app. Here's a high-level overview of what's changed.

- You now specify your connected app configuration through `SFUserAccountManager`, instead of through `SFAccountManager`. The following table shows obsolete code fragments and their replacements:

Replace This...	...With This
`[SFAccountManager setClientId:]`	`[SFUserAccountManager sharedInstance].oauthClientId`
`[SFAccountManager setRedirectUri:]`	`[SFUserAccountManager sharedInstance].oauthCompletionUrl`
`[SFAccountManager setScopes:]`	`[SFUserAccountManager sharedInstance].scopes`

- Notifications for logout events and login host changes have moved to delegate methods. Update your `AppDelegate` class to implement the `SFAuthenticationManagerDelegate` and `SFUserAccountManagerDelegate` delegates.
  - For user logout notifications, use `[SFAuthenticationManagerDelegate authManagerDidLogout:]`
  - For login host changes, use `[SFUserAccountManagerDelegate userAccountManager:didSwitchFromUser:toUser:]`.

    Note: Changing the login host in the Settings app effectively switches to a new user and requires a login.

Migrating from Version 2.0 to Version 2.1

If you developed code with Salesforce Mobile SDK2.0, follow these instructions to update your app to version 2.1.

Migrate Mobile SDK Android Applications From 2.0 to 2.1

Perform these tasks to upgrade your Android applications (native or hybrid) from Salesforce Mobile SDK 2.0 to version 2.1.

1. Replace the existing SalesforceSDK project in Eclipse with the Mobile SDK 2.1 SalesforceSDK project.
2. If your app uses SmartStore, replace the existing SmartStore project in Eclipse with the Mobile SDK 2.1 SmartStore project.
3. Right-click your project and select **Properties**.
4. Select **Android**.
5. Replace the existing SalesforceSDK entry in the library project section with the new SalesforceSDK project in your workspace.
6. If your app uses SmartStore, replace the existing SmartStore entry in the library project section with the new SmartStore project in your workspace.

   We've moved the Salesforce Mobile SDK Activity and Service declarations from the app's `AndroidManifest.xml` file to the `AndroidManifest.xml` file of the SalesforceSDK project. These settings are automatically merged into the app's manifest file if the `manifestmerger` attribute is enabled.

7. Add the following code to your app's `project.properties` file:

```
manifestmerger.enabled=true
```

Note: You're required to perform this step to use some new Mobile SDK 2.1 features, such as push notifications.

Migrate Mobile SDK iOS Applications From 2.0 to 2.1

To upgrade native and hybrid apps, we strongly recommend you create a new app from the app templates in the forceios npm package, then migrate the artifacts specific to your app into the new template.

Perform the following manual steps only if you prefer to update the Mobile SDK artifacts in your existing app.

iOS Hybrid Applications

Update Mobile SDK Library Packages

The easiest way to upgrade Mobile SDK library packages is to delete the `Dependencies` folder of your app's Xcode project, and then add the new libraries.

1. In your Xcode project, in Project Navigator, locate the `Dependencies` folder. Control-click the folder, choose **Delete**, and select **Move to Trash**.
2. Download the following binary packages from the distribution repo:
   - `Cordova/Cordova-Release.zip`
   - `SalesforceHybridSDK-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCore-Release.zip`
3. Also, download the following folders from the `ThirdParty` folder link in the distribution repo:
   - `SalesforceCommonUtils`
   - `openssl`
   - `sqlcipher`
4. Recreate the `Dependencies` folder under your app folder.
5. Unzip the new packages from step 2, and copy the folders from step 3, into the `Dependencies` folder.
6. In Project Navigator, control-click your app folder and select **Add Files to "<*app_name*>"**.
7. Select the `Dependencies` folder, making sure that **Create groups for any added folder** is selected.
8. Click **Add**.

Update Header File Search Paths

Update the header file search paths of your Xcode project.

1. Select your project in Project Navigator.
2. Select the **Build Settings** tab of your main target.
3. Scroll down to (or search/filter for) **Header Search Paths**.
4. Add the following search paths:

- `$(SRCROOT)/[App Name]/Dependencies/SalesforceSDKCore/Headers`
- `$(SRCROOT)/[App Name]/Dependencies/SalesforceOAuth/Headers`
- `$(SRCROOT)/[App Name]/Dependencies/SalesforceCommonUtils/Headers`
- `$(SRCROOT)/[App Name]/Dependencies/SalesforceHybridSDK/Headers`

Update Hybrid Local Artifacts

1. For your hybrid local apps, replace the following files in the `www/` folder of your app with the new versions from the `libs` folder of the SalesforceMobileSDK-Shared repo:
   - `cordova.force.js`
   - `forcetk.mobilesdk.js`
   - `smartsync.js`

iOS Native Applications

Update Mobile SDK Library Packages

The easiest way to upgrade Mobile SDK library packages is to delete the `Dependencies` folder of your app's Xcode project, and then add the new libraries.

1. In your Xcode project, in Project Navigator, locate the `Dependencies` folder. Control-click the folder, choose **Delete**, and select **Move to Trash**.
2. Download the following binary packages from the distribution repo:
   - `Cordova/Cordova-Release.zip`
   - `SalesforceHybridSDK-Release.zip`
   - `SalesforceNetworkSDK-Release.zip`
   - `SalesforceOAuth-Release.zip`
   - `SalesforceSDKCore-Release.zip`
3. Also, download the following folders from the `ThirdParty` folder link in the distribution repo:
   - `SalesforceCommonUtils`
   - `openssl`
   - `sqlcipher`
4. Recreate the `Dependencies` folder, under your app folder.
5. Unzip the new packages from step 2, and copy the folders from step 3, into the `Dependencies` folder.
6. In Project Navigator, control-click your app folder and select **Add Files to "*<app_name>*"**.
7. Select the `Dependencies` folder, making sure that **Create groups for any added folder** is selected.
8. Click **Add**.

Update Header File Search Paths

Update the header file search paths of your Xcode project.

1. Select your project in Project Navigator.

2. Select the **Build Settings** tab of your main target.
3. Scroll down to (or search/filter for) **Header Search Paths**.
4. Add the following search paths:
   - `$(SRCROOT)/[App Name]/Dependencies/SalesforceSDKCore/Headers`
   - `$(SRCROOT)/[App Name]/Dependencies/SalesforceOAuth/Headers`
   - `$(SRCROOT)/[App Name]/Dependencies/SalesforceNetworkSDK/Headers`
   - `$(SRCROOT)/[App Name]/Dependencies/SalesforceCommonUtils/Headers`
   - `$(SRCROOT)/[App Name]/Dependencies/SalesforceHybridSDK/Headers`

Native Mobile SDK Library Changes

In 2.1, the Mobile SDK has replaced RestKit with MKNetworkKit as the network library for native apps. MKNetworkKit is wrapped by the new SalesforceNetworkSDK library, which in turn is wrapped by the `SFRestAPI` class and its supporting classes. Most of the interfaces remain the same. Here's a list of notable changes:

- `[SFRestAPI sharedInstance].rkClient` no longer exists.
- `[SFRestAPI send:delegate:]` now returns the new `SFNetworkOperation` associated with the request.
- `SFRestRequest.networkOperation` points to the underlying `SFNetworkOperation` object associated with the request.

If your app used any underlying RestKit members for networking, you'll need to look at the equivalent functionality in MKNetworkKit and the SalesforceNetworkSDK libraries.

Migrating From Version 1.5 to Version 2.0

If you developed code with Salesforce Mobile SDK 1.5, follow these instructions to update your app to version 2.0.

Migrate Mobile SDK Android Applications From 1.5 to 2.0

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.

Migrate Mobile SDK iOS Applications From 1.5 to 2.0

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 13 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 resource page on Force.com.

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` `/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.
Search Result Layouts	`/vXX.X/search/layout/?q=`***`Comma delimited object list`***	Returns search result layout information for the objects in the query string. For each object, this call returns the list of fields displayed on the search results page as columns, the number of rows displayed on the first page, and the label used on the search results page.
Search Scope and Order	`/vXX.X/search/`***`scopeOrder`***	Returns an ordered list of objects in the default global search scope of a logged-in user. Global search keeps track of which objects the user interacts with and how

Resource Name	URI	Description
		often and arranges the search results accordingly. Objects used most frequently appear at the top of the list.

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:

- `libSalesforceNetworkSDK.a`—Underlying library for facilitating REST API calls. This library requires third-party libraries that are available through a Mobile SDK GitHub repository. See iOS Project Settings.
- `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

Use the following objects to access Salesforce data in your native app.

- `SFRestAPI`
- `SFRestAPI (Blocks)`
- `SFRestRequest`
- `SFRestAPI (QueryBuilder)`

SFRestAPI

`SFRestAPI` is the entry point for making REST requests and is generally accessed as a singleton instance 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)

Use this category extension of the `SFRestAPI` class 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 standard REST requests that `SFRestAPI` provides, you can use `SFRestRequest` methods directly to create your own:

```
NSString *path = @"/v31.0";
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"]];
```

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.

Android Packages and Classes

Java source files for the Android Mobile SDK are under `libs/SalesforceSDK/src`.

Package Catalog

Package Name	Description
`com.salesforce.androidsdk.accounts`	Classes for managing user accounts
`com.salesforce.androidsdk.app`	Contains `SalesforceSDKManager`, the entry point class for all Mobile SDK applications. This package also contains app utility classes for internal use.
`com.salesforce.androidsdk.auth`	Internal use only. Handles login, OAuth authentication, and HTTP access.
`com.salesforce.androidsdk.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..
`com.salesforce.androidsdk.push`	Components of this package register and unregister devices for Salesforce push notifications. These components then receive the notifications from a Salesforce connected app through Google Cloud Messaging (GCM). See Push Notifications and Mobile SDK.
`com.salesforce.androidsdk.rest`	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.
`com.salesforce.androidsdk.rest.files`	Classes for handling requests and responses for the Files REST API.
`com.salesforce.androidsdk.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`.
`com.salesforce.androidsdk.smartstore`	SmartStore and supporting classes.
`com.salesforce.androidsdk.ui`	Activities (for example, the login activity).
`com.salesforce.androidsdk.ui.sfhybrid`	Activity base classes for hybrid apps.
`com.salesforce.androidsdk.ui.sfnative`	Activity base classes for native apps.
`com.salesforce.androidsdk.util`	Contains utility and test classes. These classes are mostly for internal use, with some notable exceptions.

Package Name	Description
	• You can implement the `EventObserver` interface to eavesdrop on any event type. • The `EventsListenerQueue` class is useful for implementing your own tests. • Browse the `EventsObservable` source code to see a list of all supported event types.

com.salesforce.androidsdk.accounts

Class	Description
`UserAccount`	Represents a single user account that is currently logged in against a Salesforce organization
`UserAccountManager`	Used to access user accounts that are currently logged in and add new accounts for apps that don't use SmartStore
`UserAccountManagerWithSmartStore`	Used to access user accounts that are currently logged in and add new accounts for apps that use SmartStore

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
`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
SFAccountManagerPlugin	PhoneGap plugin to handle user accounts
TestRunnerPlugin	PhoneGap plugin to run javascript tests in container

com.salesforce.androidsdk.push

Class	Description
PushBroadcastReceiver	Internal use class that receives messages from Google Cloud Messaging (GCM)
PushMessaging	Internal use class that handles device registration and unregistration for push notifications, as well as storage and retrieval of registration information
PushNotificationInterface	Public interface implemented by the app to receive and handle push notifications
PushService	Internal use class that registers and unregisters the app with the Salesforce connected app to receive push notifications from the Salesforce organization

com.salesforce.androidsdk.rest

Class	Description
AdminPrefsManager	Represents custom settings made by an organization admin for a connected app
ApiVersionStrings	Encapsulates API version information
BootConfig	Encapsulates key application configuration values, including consumer key, callback URI, oAuth scopes, and refresh behavior
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

Class	Description
`RestResponse`	REST response wrapper

com.salesforce.androidsdk.rest.files

Class	Description
`ApiRequests`	Helper methods for building REST requests
`ConnectUriBuilder`	Builds Connect URIs, with special handling for user IDs and optional parameters
`FileRequests`	Defines HTTP requests that use the Connect API for files
`RenditionType`	Enumerator for rendition types that the server supports

com.salesforce.androidsdk.security

Contains the latest PRNG fixes from Google.

Class	Description
`Encryptor`	Helper class for encryption/decryption/hash computations
`PRNGFixes`	Inactivity timeout manager, kicks off passcode screen if needed
`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
AccountSwitcherActivity	Custom dialog for switching between authenticated accounts or adding a new account. This dialog pops itself off the activity stack after the account has been switched.
CustomServerUrlEditor	Custom dialog allowing user to pick a different login host
LoginActivity	Login screen
ManageSpaceActivity	Overridable activity that gives a user the option to clear user data and log out
OAuthWebviewHelper	Helper class to manage a WebView instance that is going through the OAuth login process
PasscodeActivity	Passcode (PIN) screen
SalesforceAccountRadioButton	Custom radio button that represents a Salesforce account. Use the custom `setText()` method to display text in this radio button.
SalesforceR	Class that allows references to resources defined outside the SDK
SalesforceServerRadioButton	Custom radio button that represents a custom server endpoint. Use the custom `setText()` method to display text in this radio button.
ServerPickerActivity	Activity for changing the login server URL during an OAuth flow. The user can add custom servers or choose from a list of servers.

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

Important: Every activity in a native Mobile SDK app must extend or duplicate the functionality of one of the classes in this package.

Class	Description
`SalesforceActivity`	Main activity of native applications, based on the Android `Activity` class
`SalesforceListActivity`	Main activity of native applications, based on the Android `ListActivity` class
`SalesforceExpandableListActivity`	Main activity of native applications, based on the Android `ExpandableListActivity` class

com.salesforce.androidsdk.util

Class	Description
`EventsListenerQueue`	Class to track activity events using a queue, allowing for tests to wait for certain events to turn up
`EventsObservable`	Used to register and receive events generated by the SDK (used primarily in tests)
`EventsObserver`	Observer of SDK events
`ForceAppInstrumentationTestCase`	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
`UriFragmentParser`	Parses URI fragments that use query string style to pass parameters (for example, `foo=bar&bar=foo2`)

Class	Description
`UserSwitchReceiver`	Listener for the user switch event

Libraries

The following libraries are under `/libs/SalesforceSDK/libs`.

Library Name	Description
`android-junit-report-1.5.8.jar`	Custom instrumentation test runner for Android that creates XML test reports in a format that's similar to reports that are created by Ant JUnit task's XML formatter
`apache-mime4j-0.7.2.jar`	MIME message parser based on Java streams
`guava-18.0.jar`	Java library required by sqlcipher
`httpcore-4.3.2.jar`	HTTP transport components
`httpmime-4.3.2.jar`	MIME-coded entities
`volley_android-4.4.2_r2.jar`	Android networking library from Google

The following libraries are under `/external/sqlcipher/libs`.

Library Name	Description
`armeabi/*.so`	Native libaries required by sqlcipher on ARM-based devices (**)
`commons-code.jar, guava-18.0.jar`	Java libraries required by sqlcipher
`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

(*) denotes files required for hybrid application.

(**) denotes files required for SmartStore.

Android Resources

Resources are under `/res`.

`drawable-hdpi`

File	Use
`sf__edit_icon.png`	Server picker screen

File	Use
sf__highlight_glare.png	Login screen
sf__icon.png	Native 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-xhdpi

File	Use
sf__icon.png	Native application icon

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)

File	Use
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)

drawable-xxhdpi

File	Use
sf__hybrid__icon.png	Hybrid application icon
sf__icon.png	Native application icon

drawable

File	Use
sf__header_bg.png	Login screen
sf__progress_spinner.xml	Login screen
sf__toolbar_background.xml	Login screen

layout

File	Use
sf__account_switcher.xml	Account switching screen
sf__custom_server_url.xml	Server picker screen
sf__login.xml	Login screen
sf__manage_space.xml	Screen that allows the user to clear app data and log out

File	Use
`sf__passcode.xml`	Pin screen
`sf__server_picker.xml`	Server picker screen (deprecated)
`sf__server_picker_list.xml`	Server picker screen

`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
`sf__dimens.xml`	Dimensions
`sf__strings.xml`	SDK strings
`sf__style.xml`	Styles
`strings.xml`	Other strings (app name)

`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.

Files API Reference

API access for the Files feature is available in Android, iOS, and hybrid flavors.

FileRequests Methods (Android)

All `FileRequests` methods are static, and each returns a `RestRequest` instance. Use the `RestClient.sendAsync()` or the `RestClient.sendSync()` method to send the `RestRequest` object to the server. See Using REST APIs.

For a full description of the REST request and response bodies, see "Files Resources" under *Chatter REST API Resources* at http://www.salesforce.com/us/developer/docs/chatterapi.

ownedFilesList

Generates a request that retrieves a list of files that are owned by the specified user. Returns one page of results.

Signature

```
public static RestRequest ownedFilesList(String userId, Integer pageNum);
```

Parameters

Name	Type	Description
`userId`	`String`	ID of a user. If null, the ID of the context (logged-in) user is used.
`pageNum`	`Integer`	Zero-based index of the page of results to be fetched. If null, fetches the first page.

Example

```
RestRequest request = FileRequests.ownedFilesList(null, null);
```

filesInUsersGroups

Generates a request that retrieves a list of files that are owned by groups that include the specified user.

Signature

```
public static RestRequest filesInUsersGroups(String userId, Integer pageNum);
```

Parameters

Name	Type	Description
`userId`	`String`	ID of a user. If null, the ID of the context (logged-in) user is used.
`pageNum`	`Integer`	Zero-based index of the page of results to be fetched. If null, fetches the first page.

Example

```
RestRequest request = FileRequests.filesInUsersGroups(null, null);
```

filesSharedWithUser

Generates a request that retrieves a list of files that are shared with the specified user.

Signature

```
public static RestRequest filesSharedWithUser(String userId, Integer pageNum);
```

Parameters

Name	Type	Description
`userId`	`String`	ID of a user. If null, the ID of the context (logged-in) user is used.
`pageNum`	`Integer`	Zero-based index of the page of results to be fetched. If null, fetches the first page.

Example

```
RestRequest request = FileRequests.filesSharedWithUser(null, null);
```

fileDetails

Generates a request that can fetch the file details of a particular version of a file.

Signature

```
public static RestRequest fileDetails(String sfdcId, String version);
```

Parameters

Name	Type	Description
`sfdcId`	`String`	ID of a file. If null, `IllegalArgumentException` is thrown.
`version`	`String`	Version to fetch. If null, fetches the most recent version.

Example

```
String id = <some_file_id>;
RestRequest request = FileRequests.fileDetails(id, null);
```

batchFileDetails

Generates a request that can fetch details of multiple files.

Signature

```
public static RestRequest batchFileDetails(List sfdcIds);
```

Parameters

Name	Type	Description
`sfdcIds`	`List`	List of IDs of one or more files. If any ID in the list is null, `IllegalArgumentException` is thrown.

Example

```
List<String> ids = Arrays.asList("id1", "id2", ...);
RestRequest request = FileRequests.batchFileDetails(ids);
```

fileRendition

Generates a request that can fetch a rendered preview of a page of the specified file.

Signature

```
public static RestRequest fileRendition(String sfdcId,
   String version,
   RenditionType renditionType,
   Integer pageNum);
```

Parameters

Name	Type	Description
`sfdcId`	`String`	ID of a file to be rendered. If null, `IllegalArgumentException` is thrown.
`version`	`String`	Version to fetch. If null, fetches the most recent version.
`renditionType`	`RenditionType`	Specifies the type of rendition to be returned. Valid values include: • `PDF` • `FLASH` • `SLIDE` • `THUMB120BY90` • `THUMB240BY180` • `THUMB720BY480` If null, `THUMB120BY90` is used.
`pageNum`	`Integer`	Zero-based index of the page to be fetched. If null, fetches the first page.

Example

```
String id = <some_file_id>;
RestRequest request = FileRequests.fileRendition(id, null, "PDF", 0);
```

fileContents

Generates a request that can fetch the binary contents of the specified file.

Signature

```
public static RestRequest fileContents(String sfdcId, String version);
```

Parameters

Name	Type	Description
`sfdcId`	`String`	ID of a file to be rendered. If null, `IllegalArgumentException` is thrown.
`version`	`String`	Version to fetch. If null, fetches the most recent version.

Example

```
String id = <some_file_id>;
RestRequest request = FileRequests.fileContents(id, null);
```

fileShares

Generates a request that can fetch a page from the list of entities that share the specified file.

Signature

```
public static RestRequest fileShares(String sfdcId, Integer pageNum);
```

Parameters

Name	Type	Description
`sfdcId`	`String`	ID of a file to be rendered. If null, `IllegalArgumentException` is thrown.
`pageNum`	`Integer`	Zero-based index of the page of results to be fetched. If null, fetches the first page.

Example

```
String id = <some_file_id>;
RestRequest request = FileRequests.fileShares(id, null);
```

addFileShare

Generates a request that can share the specified file with the specified entity.

Signature

```
public static RestRequest addFileShare(String fileId, String entityId,
   String shareType);
```

Parameters

Name	Type	Description
`fileId`	`String`	ID of a file to be shared. If null, `IllegalArgumentException` is thrown.

Name	Type	Description
`entityID`	`String`	ID of a user or group with whom to share the file. If null, `IllegalArgumentException` is thrown.
`shareType`	`String`	Type of share. Valid values are "V" for view and "C" for collaboration.

Example

```
String idFile = <some_file_id>;
String idEntity = <some_user_or_group_id>;
RestRequest request = FileRequests.addFileShare(idFile, idEntity, "V");
```

deleteFileShare

Generates a request that can delete the specified file share.

Signature

```
public static RestRequest deleteFileShare(String shareId);
```

Parameters

Name	Type	Description
`shareId`	`String`	ID of a file share to be deleted. If null, `IllegalArgumentException` is thrown.

Example

```
String id = <some_fileShare_id>;
RestRequest request = FileRequests.deleteFileShare(id);
```

uploadFile

Generates a request that can upload a local file to the server. On the server, this request creates a file at version 1.

Signature

```
public static RestRequest uploadFile(File theFile,
   String name, String description, String mimeType)
   throws UnsupportedEncodingException;
```

Parameters

Name	Type	Description
`theFile`	`File`	Path of the local file to be uploaded to the server.
`name`	`String`	Name of the file.
`description`	`String`	Description of the file.

Name	Type	Description
mimeType	String	MIME type of the file, if known. Otherwise, null.

Throws

UnsupportedEncodingException

Example

```
RestRequest request = FileRequests.uploadFile("/Users/JayVee/Documents/",
   "mypic.png", "Profile pic", "image/png");
```

SFRestAPI (Files) Category—Request Methods (iOS)

In iOS native apps, the SFRestAPI (Files) category defines file request methods. You send request messages to the SFRestAPI singleton.

```
SFRestRequest *request = [[SFRestAPI sharedInstance] requestForOwnedFilesList:nil page:0];
```

Each method returns an SFRestRequest instance. Use the SFRestAPI singleton again to send the request object to the server. In the following example, the calling class (self) is the delegate, but you can specify any other object that implements SFRestDelegate.

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

requestForOwnedFilesList:page:

Generates a request that retrieves a list of files that are owned by the specified user. Returns one page of results.

Signature

```
- (SFRestRequest *)
  requestForOwnedFilesList:(NSString *)userId
                      page:(NSUInteger)page;
```

Parameters

Name	Type	Description
userId	NSString *	ID of a user. If nil, the ID of the context (logged-in) user is used.
page	NSUInteger	Zero-based index of the page to be fetched. If nil, fetches the first page.

Example

```
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForOwnedFilesList:nil
                                                   page:0];
```

requestForFilesInUsersGroups:page:

Generates a request that retrieves a list of files that are owned by groups that include the specified user.

Signature

```
- (SFRestRequest *)
  requestForFilesInUsersGroups:(NSString *)userId
                          page:(NSUInteger)page;
```

Parameters

Name	Type	Description
`userId`	`NSString *`	ID of a user. If nil, the ID of the context (logged-in) user is used.
`page`	`NSUInteger`	Zero-based index of the page to be fetched. If nil, fetches the first page.

Example

```
SFRestRequest *request = [[SFRestAPI sharedInstance]
                              requestForFilesInUsersGroups:nil
                                                      page:0];
```

requestForFilesSharedWithUser:page:

Generates a request that retrieves a list of files that are shared with the specified user.

Signature

```
- (SFRestRequest *)
  requestForFilesSharedWithUser:(NSString *)userId
                           page:(NSUInteger)page;
```

Parameters

Name	Type	Description
`userId`	`NSString *`	ID of a user. If nil, the ID of the context (logged-in) user is used.
`page`	`NSUInteger`	Zero-based index of the page to be fetched. If nil, fetches the first page.

Example

```
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForFilesSharedWithUser:nil
                                                        page:0];
```

requestForFileDetails:forVersion:

Generates a request that can fetch the file details of a particular version of a file.

Signature

```
- (SFRestRequest *)
  requestForFileDetails:(NSString *)sfdcId
             forVersion:(NSString *)version;
```

Parameters

Name	Type	Description
`sfdcId`	`NSString *`	ID of a file. If nil, the request fails.
`version`	`NSString *`	Version to fetch. If nil, fetches the most recent version.

Example

```
NSString *id = [NSString stringWithString:@"some_file_id"];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForFileDetails:id
                                          forVersion:nil];
```

requestForBatchFileDetails:

Generates a request that can fetch details of multiple files.

Signature

```
- (SFRestRequest *)
  requestForBatchFileDetails:(NSArray *)sfdcIds;
```

Parameters

Name	Type	Description
`sfdcIds`	`NSArray *`	Array of IDs of one or more files. IDs are expressed as strings.

Example

```
NSArray *ids = [NSArray arrayWithObject:@"id1",@"id2",...,nil];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForBatchFileDetails:ids];
```

requestForFileRendition:version:renditionType:page:

Generates a request that can fetch a rendered preview of a page of the specified file.

Signature

```
- (SFRestRequest *)
  requestForFileRendition:(NSString *)sfdcId
                  version:(NSString *)version
             renditionType:(NSString *)renditionType
                     page:(NSUInteger)page;
```

Parameters

Name	Type	Description
`sfdcId`	`NSString *`	ID of a file to be rendered. If nil, the request fails.

Name	Type	Description
`version`	`NSString *`	Version to fetch. If nil, fetches the most recent version.
`renditionType`	`NSString *`	Specifies the type of rendition to be returned. Valid values include: • `"PDF"` • `"FLASH"` • `"SLIDE"` • `"THUMB120BY90"` • `"THUMB240BY180"` • `"THUMB720BY480"` If nil, `THUMB120BY90` is used.
`page`	`NSUInteger`	Zero-based index of the page to be fetched. If nil, fetches the first page.

Example

```
NSString *id = [NSString stringWithString:@"some_file_id"];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForFileRendition:id
                                               version:nil
                                         renditionType:nil
                                                  page:nil];
```

requestForFileContents:version:

Generates a request that can fetch the binary contents of the specified file.

Signature

```
- (SFRestRequest *)
  requestForFileContents:(NSString *) sfdcId
                 version:(NSString*) version;
```

Parameters

Name	Type	Description
`sfdcId`	`NSString *`	ID of a file to be rendered. If nil, the request fails.
`version`	`NSString *`	Version to fetch. If nil, fetches the most recent version.

Example

```
NSString *id = [NSString stringWithString:@"some_file_id"];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForFileContents:id
                                              version:nil];
```

requestForFileShares:page:

Generates a request that can fetch a page from the list of entities that share the specified file.

Signature

```
- (SFRestRequest *)
  requestForFileShares:(NSString *)sfdcId
                  page:(NSUInteger)page;
```

Parameters

Name	Type	Description
sfdcId	NSString *	ID of a file to be rendered. If nil, the request fails.
page	NSUInteger	Zero-based index of the page to be fetched. If nil, fetches the first page.

Example

```
NSString *id = [NSString stringWithString:@"some_file_id"];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForFileShares:id
                                               page:nil];
```

requestForAddFileShare:entityId:shareType: Method

Generates a request that can share the specified file with the specified entity.

Signature

```
- (SFRestRequest *)
  requestForAddFileShare:(NSString *)fileId
                entityId:(NSString *)entityId
               shareType:(NSString*)shareType;
```

Parameters

Name	Type	Description
fileId	NSString *	ID of a file to be shared. If nil, the request fails.
entityId	NSString *	ID of a user or group with whom to share the file. If nil, the request fails.
shareType	NSString *	Type of share. Valid values are "V" for view and "C" for collaboration.

Example

```
NSString *id = [NSString stringWithString:@"some_file_id"];
NSString *entId = [NSString stringWithString:@"some_entity_id"];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForAddFileShare:id
                                             entityId:entId
                                            shareType:@"V"];
```

requestForDeleteFileShare:

Generates a request that can delete the specified file share.

Signature

```
- (SFRestRequest *)
  requestForDeleteFileShare:(NSString *)shareId;
```

Parameters

Name	Type	Description
shareId	NSString *	ID of a file share to be deleted. If nil, the request fails.

Example

```
NSString *id = [NSString stringWithString:@"some_fileshare_id"];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForDeleteFileShare:id];
```

requestForUploadFile:name:description:mimeType: Method

Generates a request that can upload a local file to the server. On the server, this request creates a new file at version 1.

Signature

```
- (SFRestRequest *)
  requestForUploadFile:(NSData *)data
                  name:(NSString *)name
           description:(NSString *)description
              mimeType:(NSString *)mimeType;
```

Parameters

Name	Type	Description
data	NSData *	Data to upload to the server.
name	NSString *	Name of the file.
description	NSString *	Description of the file.
mimeType	NSString *	MIME type of the file, if known. Otherwise, nil.

Example

```
NSData *data = [NSData dataWithContentsOfFile:@"/Users/JayVee/Documents/mypic.png"];
SFRestRequest *request =
   [[SFRestAPI sharedInstance] requestForUploadFile:data
                                               name:@"mypic.png"
                                        description:@"Profile pic"
                                           mimeType:@"image/png"];
```

Files Methods For Hybrid Apps

Hybrid methods for the Files API reside in the `forcetk.mobilesdk.js` library. Examples in the following reference topics assume that you've declared a local `ftkclient` variable, such as:

```
var ftkclient = new forcetk.Client(creds.clientId, creds.loginUrl, creds.proxyUrl, reauth);
```

Note: In `smartsync.js`, the `forcetk.Client` object is wrapped as `Force.forcetkClient`. You're free to use either client in a SmartSync app. However, REST API methods called on `Force.forcetkClient` differ from their `forcetk.Client` cousins in that they return JavaScript promises. If you use `Force.forcetkClient`, reformat the examples that require success and error callbacks in the following manner:

```
Force.forcetkClient.ownedFilesList(null, null)
    .done(function(response) {/* do something with the returned JSON data */})
    .fail(function(error) { alert("Error!");});
```

ownedFilesList Method

Returns a page from the list of files owned by the specified user.

Signature

```
forcetk.Client.prototype.
ownedFilesList =
function(userId, page, callback, error)
```

Parameters

Name	Description
`userId`	An ID of an existing user. If null, the ID of the context (currently logged in) user is used.
`page`	Zero-based index of the page of results to be fetched. If null, fetches the first page.
`callback`	A function that receives the server response asynchronously and handles it.
`error`	A function that handles server errors.

Example

```
ftkclient.ownedFilesList(null, null,
    function(response){ /* do something with the returned JSON data */},
    function(error){ alert("Error!");}
);
```

filesInUsersGroups Method

Returns a page from the list of files owned by groups that include specified user.

Signature

```
forcetk.Client.prototype.
filesInUsersGroups =
function(userId, page, callback, error)
```

Parameters

Name	Description
`userId`	An ID of an existing user. If null, the ID of the context (currently logged in) user is used.
`page`	Zero-based index of the page of results to be fetched. If null, fetches the first page.
`callback`	A function that receives the server response asynchronously and handles it.
`error`	A function that handles server errors.

Example

```
ftkclient.filesInUsersGroups(null, null,
    function(response){ /* do something with the returned JSON data */},
    function(error){ alert("Error!");}
);
```

filesSharedWithUser Method

Returns a page from the list of files shared with the specified user.

Signature

```
forcetk.Client.prototype.
filesSharedWithUser =
function(userId, page, callback, error)
```

Parameters

Name	Description
`userId`	An ID of an existing user. If null, the ID of the context (currently logged in) user is used.
`page`	Zero-based index of the page of results to be fetched. If null, fetches the first page.
`callback`	A function that receives the server response asynchronously and handles it.
`error`	A function that handles server errors.

Example

```
ftkclient.filesSharedWithUser(null, null,
    function(response){ /* do something with the returned JSON data */},
    function(error){ alert("Error!");}
);
```

fileDetails Method

Generates a request that can fetch the file details of a particular version of a file.

Signature

```
forcetk.Client.prototype.
fileDetails = function
(fileId, version, callback, error)
```

Parameters

Name	Description
`sfdcId`	An ID of an existing file. If null, an error is returned.
`version`	The version to fetch. If null, fetches the most recent version.
`callback`	A function that receives the server response asynchronously and handles it.
`error`	A function that handles server errors.

Example

```
ftkclient.fileDetails(id, null,
    function(response){ /* do something with the returned JSON data */},
    function(error){ alert("Error!");}
);
```

batchFileDetails Method

Returns file details for multiple files.

Signature

```
forcetk.Client.prototype.
batchFileDetails =
function(fileIds, callback, error)
```

Parameters

Name	Description
`fileIds`	A list of IDs of one or more existing files. If any ID in the list is null, an error is returned.
`callback`	A function that receives the server response asynchronously and handles it.
`error`	A function that handles server errors.

Example

```
ftkclient.batchFileDetails(ids,
    function(response){ /* do something with the returned JSON data */},
    function(error){ alert("Error!");}
);
```

fileRenditionPath Method

Returns file rendition path relative to `service/data`. In HTML (for example, an `img` tag), use the bearer token URL instead.

Signature

```
forcetk.Client.prototype.
fileRenditionPath =
    function(fileId, version, renditionType,
        page)
```

Parameters

Name	Description
`fileId`	ID of an existing file to be rendered. If null, an error is returned.
`version`	The version to fetch. If null, fetches the most recent version.
`renditionType`	Specify the type of rendition to be returned. Valid values include: • `PDF` • `FLASH` • `SLIDE` • `THUMB120BY90` • `THUMB240BY180` • `THUMB720BY480` If null, `THUMB120BY90` is used.
`page`	Zero-based index of the page to be fetched. If null, fetches the first page.

Example

```
ftkclient.fileRenditionPath(id, null, "THUMB240BY180", null);
```

fileContentsPath Method

Returns file content path (relative to `service/data`). From html (for example, an `img` tag), use the bearer token URL instead.

Signature

```
forcetk.Client.prototype.
fileContentsPath =
function(fileId, version)
```

Parameters

Name	Description
`fileId`	ID of an existing file to be rendered. If null, an error is returned.
`version`	The version to fetch. If null, fetches the most recent version.

Example

```
ftkclient.fileContentsPath(id, null);
```

fileShares Method

Returns a page from the list of entities that share this file.

Signature

```
forcetk.Client.prototype.
fileShares = function
(fileId, page, callback, error)
```

Parameters

Name	Description
`fileId`	ID of an existing file to be rendered. If null, an error is returned.
`page`	Zero-based index of the page of results to be fetched. If null, fetches the first page.
`callback`	A function that receives the server response asynchronously and handles it.
`error`	A function that handles server errors.

Example

```
ftkclient.fileShares(id, null,
    function(response){ /* do something with the returned JSON data */},
    function(error){ alert("Error!");}
);
```

addFileShare Method

Adds a file share for the specified file ID to the specified entity ID.

Signature

```
forcetk.Client.prototype.
addFileShare = function
(fileId, entityId, shareType, callback, error)
```

Parameters

Name	Description
`fileId`	ID of an existing file to be shared. If null, `IllegalArgumentException` is thrown.
`entityID`	ID of an existing user or group with whom to share the file. If null, `IllegalArgumentException` is thrown.
`shareType`	The type of share. Valid values are "V" for view and "C" for collaboration.
`callback`	A function that receives the server response asynchronously and handles it.

Name	Description
`error`	A function that handles server errors.

Example

```
ftkclient.addFileShare(id, null, "V",
    function(response){ /* do something with the returned JSON data */},
    function(error){ alert("Error!");}
);
```

deleteFileShare Method

Deletes the specified file share.

Signature

```
forcetk.Client.prototype.
deleteFileShare =
function(sharedId, callback, error)
```

Parameters

Name	Description
`shareId`	ID of an existing file share to be deleted. If null, `IllegalArgumentException` is thrown.
`callback`	A function that receives the server response asynchronously and handles it.
`error`	A function that handles server errors.

Example

```
ftkclient.deleteFileShare(id,
    function(response){
        /* do something with the returned JSON data */
    },
    function(error){ alert("Error!");}
);
```

Forceios Parameters

These are the descriptions of the forceios command parameters:

Parameter Name	Description
`--apptype`	One of the following: • "native" • "hybrid_remote" (server-side hybrid app using VisualForce)

Parameter Name	Description
	• "hybrid_local" (client-side hybrid app that doesn't use VisualForce)
`--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.".
`--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. Note: If you don't specify your own 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. Note: • If you don't specify your own value here, you're required to change it in the app before you publish to the App Store. • If you accept the default value for `--appid`, be sure to also accept the default value for `--callbackuri`.

Forcedroid Parameters

The following table describes the forcedroid command parameters.

Parameter Name	Description
`--apptype`	One of the following: • "native" • "hybrid_remote" (server-side hybrid app using VisualForce) • "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=yes`	(Optional) Include only if you want to use SmartStore and, optionally, SmartSync for offline data. Defaults to no if not specified.

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X