Mobile SDK For Windows

Salesforce, Winter '17

@salesforcedocs

Last updated: December 22, 2016

CONTENTS

Introduction to Salesforce Mobile SDK Development 1
Salesforce Prerequisites 1
Creating a Connected App 2
Getting Started 6
Development Prerequisites for Microsoft Windows 6
Mobile SDK GitHub Repositories 7
Create a Mobile SDK Native Project for Windows 10 8
Create a Mobile SDK Hybrid Project for Windows 10 9
Add Mobile SDK to an Existing Windows 10 Project 10
Windows Sample Apps 11
Native Windows Development 12
Mobile SDK Native Libraries 12
Developing a Native Windows 10 App 13
HTML5 and Hybrid Development 24
Getting Started 24
HTML5 Development Tools 28
Delivering HTML5 Content With Visualforce 29
Accessing Salesforce Data: Controllers vs. APIs 29
Hybrid Apps Quick Start 32
Creating Hybrid Apps 32
Controlling the Status Bar in Hybrid Apps 34
JavaScript Files for Hybrid Apps 34
Defer Login 35
Offline Management 36
Using SmartStore to Store Offline Data 36
Using SmartSync to Access Salesforce Objects 50
Using Communities With Mobile SDK Apps 65
Communities and Mobile SDK Apps 65
Set Up an API-Enabled Profile 66
Set Up a Permission Set 66
Grant API Access to Users 67
Configure the Login Endpoint 68
Brand Your Community 68
Customize Login, Logout, and Self-Registration Pages in Your Community 69
Using External Authentication With Communities 69

Example: Configure a Community For Mobile SDK App Access 80
Example: Configure a Community For Facebook Authentication 83

Multi-User Support in Mobile SDK 87

About Multi-User Support 87
Implementing Multi-User Support 87

Authentication, Security, and Identity in Mobile Apps 89

OAuth Terminology 89
OAuth 2.0 Authentication Flow 89
Connected Apps 99
Portal Authentication Using OAuth 2.0 and Force.com Sites 100

INTRODUCTION TO SALESFORCE MOBILE SDK DEVELOPMENT

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 Mobile SDK, take a look at how mobile development works, and learn about essential Salesforce developer resources.

Creating a Connected App

Salesforce 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.

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.

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 isn't business-critical. Development can be done inside your browser or with the Force.com IDE, which is based on the Eclipse development tool.

Types of Developer Environments

A *Developer Edition* environment is a free, fully featured copy of the Enterprise Edition environment, with less storage and users. Developer Edition is a logically separate environment, ideal as your initial development environment. You can sign up for as many Developer Edition orgs 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 Developer Edition 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 Professional, Enterprise, Performance, and Unlimited Edition customers. The sandbox copy can include data, configurations, or both. You can 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 org. 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're a:
  - Partner who intends to build a commercially available Force.com app by creating a managed package for distribution through AppExchange or Trialforce. Only Developer Edition or Partner Developer Edition environments can create managed packages.
  - Salesforce customer with a Group or Personal Edition, and you don't have access to Sandbox.
  - 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 has a Developer Edition environment and checks code in and out of this master repository environment.
  - Expect more than two developers to log in to develop and test.
  - Require a larger environment that allows more users to run robust tests against larger data sets.
- Sandbox is ideal if you:
  - Are a Salesforce customer with Professional, Enterprise, Performance, Unlimited, or Force.com Edition, which includes Sandbox.
  - Are developing a Force.com application specifically for your production environment.
  - Aren't planning to build a Force.com application to be distributed commercially.
  - Have no intention to list on the AppExchange or distribute through Trialforce.

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.

Create a Connected App

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

1. Log into your Force.com instance.
2. In Setup, enter *Apps* in the `Quick Find` box, then select **Apps**.
3. Under Connected Apps, click **New**.
4. Perform steps for Basic Information.
5. Perform steps for API (Enable OAuth Settings).
6. Click **Save**.

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 App Manager and on its App Launcher tile.

   Note: The connected app name must be unique for the connected apps in your org. You can reuse the name of a deleted connected app if the connected app was created using the Spring '14 release or later.

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 if the original app name contains any other characters, edit the default name.
3. Enter the contact email for Salesforce to use when contacting you or your support team. This address isn't given to Salesforce admins who install the app.
4. Enter the contact phone for Salesforce to use in case we need to contact you. This number isn't given to Salesforce admins who install the app.
5. Enter a logo image URL to display your logo on the App Launcher tile. It also appears on the consent page that users see when authenticating. The URL must use HTTPS. Use a GIF, JPG, or PNG file format and a file size that's preferably under 20 MB, but, at most 100 KB. We resize the image to 128 pixels by 128 pixels, so be sure that you like how it looks. If you don't supply a logo, Salesforce generates one for you using the app's initials.
   - 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 that the logo meets the size requirements.
   - You can upload your own image by clicking **Upload logo image**. Select an image from your local file system. When your upload is successful, the URL to the logo appears in the `Logo Image URL` field.

- You can also select a logo from the samples provided by Salesforce by clicking **Choose one of our sample logos**. The logos include ones for Salesforce apps, third-party apps, and standards bodies. Click the logo you want, and then copy and paste the URL into the `Logo Image URL` field.
- You can use a logo hosted publicly on Salesforce servers by uploading an image as a document using the Documents tab. View the image to get the URL, and then 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. Use an icon that's 16 pixels high and wide and on a white background.

   You can select an icon from the samples provided by Salesforce. Click **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 Web page with more information about your app, provide an info URL.
8. Enter a description, up to 256 characters, to display on the connected app's App Launcher tile. If you don't supply a description, just the name appears on the tile.

Note: The App Launcher displays the connected app's name, description, and logo (if provided) on an App Launcher tile. Make sure that the text is meaningful and mistake-free.

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, the URL is typically the one that a user's browser is redirected to after successful authentication. Because this URL is used for some OAuth flows to pass an access token, the URL must use secure HTTPS or a custom URI scheme. If you enter multiple callback URLs, at run time Salesforce matches the callback URL value specified by the app 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. The OAuth token name is 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 org associated with the connected app. It 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 the logged-in user's data, and encompasses all other scopes. `full` doesn't return a refresh token. You must explicitly request the `refresh_token` scope to get one.

Perform requests on your behalf at any time (refresh_token, offline_access)
Allows a refresh token to be returned if the app is eligible to receive one. This scope 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 use of the `access_token` on the web. It includes `visualforce`, which allows access to Visualforce pages.

4. Control how the OAuth request handles the ID token. If the OAuth request includes the `openid` scope, the returned token can include the ID token.
   - To include the ID token in refresh token responses, select `Include ID Token`. It's always included in access token responses.
   - With the primary ID token setting enabled, configure the secondary settings that control the ID token contents in both access and refresh token responses. Select at least one of these settings.

     Include Standard Claims
     Include the standard claims that contain information about the user, such as the user's name, profile, phone_number, and address. The OpenID Connect specifications define a set of standard claims to be returned in the ID token.

     Include Custom Attributes
     If your app has specified custom attributes, include them in the ID token.

     Include Custom Permissions
     If your app has specified custom permissions, include them in the ID token.

If your org had the **No user approval required for users in this organization** option selected on your remote access before the Spring '12 release, users in the org where the app was created are automatically approved for the app. This option is selected to indicate the automatic approval. For connected apps, the recommended procedure after you've created an app is for admins to install the app and then set Permitted Users to **Admin-approved users**. If the remote access option wasn't originally selected, the option doesn't show up.

GETTING STARTED

Let's get started creating custom mobile apps! Start here if you've already signed up for Force.com and have created a Salesforce Connected App.

Mobile SDK for Windows 10 is now generally available for native and hybrid development. You can provide feedback and suggestions in the SalesforceMobileSDK Google+ community.

Note: Mobile SDK for Windows 10 supports SmartStore and SmartSync in native apps only. Mobile SDK does not support these components in Windows 10 hybrid apps.

The next steps are:

1. Make sure you have the required software.
2. Do any of the following:
   - Create a Mobile SDK Native Project for Windows 10
   - Create a Mobile SDK Hybrid Project for Windows 10
   - Add Mobile SDK to an Existing Windows 10 Project

Development Prerequisites for Microsoft Windows

Before you begin using Mobile SDK for Windows, make sure you've installed the following software. Hybrid projects require the same setup as their target platforms plus a few hybrid-only steps.

Mobile SDK GitHub Repositories

The GitHub open source repo lets you keep up with the latest Mobile SDK changes, and possibly contribute to Mobile SDK development. Using GitHub allows you to monitor source code in public pre-release development branches. You also use your clone of the GitHub repo to create apps. These app includes Mobile SDK source code, which is built along with your app.

Create a Mobile SDK Native Project for Windows 10

To create native apps, use the `createapp.ps1` script.

Create a Mobile SDK Hybrid Project for Windows 10

Add Mobile SDK to an Existing Windows 10 Project

To add Mobile SDK to an existing Windows 10 project, follow these steps.

Windows Sample Apps

Mobile SDK for Windows 10 provides sample apps that demonstrate major Mobile SDK features.

Development Prerequisites for Microsoft Windows

Before you begin using Mobile SDK for Windows, make sure you've installed the following software. Hybrid projects require the same setup as their target platforms plus a few hybrid-only steps.

For all projects:

- Microsoft Windows 10
- Windows 10 SDK

- Microsoft .NET Framework 4.6.1
- Visual Studio Enterprise 2015, Update 3 or newer
- Visual Studio SDK (http://msdn.microsoft.com/en-us/library/bb166441.aspx)
- SQLite for Visual Studio (http://sqlite.org/2015/sqlite-uwp-3150000.vsix)

Extra requirements for hybrid projects:

- Git command-line utility (http://git-scm.com/download/win)
- node.js: (https://nodejs.org/)
- Cordova: After you've installed `node.js`, run the following command at the Windows command prompt: *`npm install -g cordova`*. Afterwards, run *`cordova --version`* to verify that you've installed Cordova CLI version 5.0.0 or greater. See "Introducing the Windows 10 Apache Cordova Platform" at blogs.msdn.com for more information.

Important: Windows 8.1 apps are not compatible with the Windows 10 release of Salesforce Mobile SDK

Mobile SDK GitHub Repositories

The GitHub open source repo lets you keep up with the latest Mobile SDK changes, and possibly contribute to Mobile SDK development. Using GitHub allows you to monitor source code in public pre-release development branches. You also use your clone of the GitHub repo to create apps. These app includes Mobile SDK source code, which is built along with your app.

You don't need to sign up for GitHub to access 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 release in our public repository:

- `https://github.com/forcedotcom/SalesforceMobileSDK-Windows`

Important: To submit pull requests for any Mobile SDK platform, check out the **`unstable`** branch as the basis for your changes.

If you're using GitHub to get and build source code for the current release without making changes, check out the **`master`** branch.

Cloning the Mobile SDK for Windows GitHub Repository

Mobile SDK for Windows lives in the `https://github.com/forcedotcom/SalesforceMobileSDK-Windows` GitHub repo. To clone this repo to your local file system:

1. Open a Windows command prompt (`cmd.exe`) as Administrator.
2. Run the following commands:

```
git clone git://github.com/forcedotcom/SalesforceMobileSDK-Windows.git
cd SalesforceMobileSDK-Windows
git submodule update --init --recursive
```

Cloning the Windows Samples GitHub Repository

Samples apps for Mobile SDK for Windows live in the `https://github.com/forcedotcom/SalesforceMobileSDK-WindowsSamples` GitHub repo. To clone this repo to your local file system:

1. Open a Windows command prompt (`cmd.exe`) as Administrator.
2. Run the following commands:

```
git clone git://github.com/forcedotcom/SalesforceMobileSDK-WindowsSamples.git
cd SalesforceMobileSDK-WindowsSamples
git submodule update --init --recursive
```

Create a Mobile SDK Native Project for Windows 10

To create native apps, use the `createapp.ps1` script.

Before creating a project, be sure that you've installed the prerequisite software described in Development Prerequisites for Microsoft Windows. Also, be sure that your clone of the Mobile SDK for Windows repo is up-to-date.

1. Run the following PowerShell script:
   `<local_path_to>\SalesforceMobileSDK-Windows\SalesforceSDK\template\createapp.ps1 create`
2. At the prompts, enter the following values:

Parameter prompt	Value
Application type	*native*
Application name	An unqualified app name, such as "MyNative50"
Organization name	Your organization's or company's name
Output directory	Absolute path to a local directory, such as `C:\Users\xxxx\Development\native` Note: The createapp script doesn't accept relative paths.
Connected App ID	Consumer Key from your connected app. If left empty, be sure to replace the default value with your own ID before publishing your app.
Connected App Callback URI	Callback URL from your connected app. If left empty, be sure to replace the default value with your own URL before publishing your app.

For example:

```
PS> .\SalesforceSDK\template\createapp.ps1 create
Enter your application type (native): native
Enter your application name: MyNative50
```

```
Enter your organization name (Acme, Inc.): Wonderful Windows Works, Inc.
Enter the output directory for your app (defaults to the current directory):
C:\Users\xxxx\Development\native
Enter your Connected App ID (defaults to the sample app's ID):
3MVG9Iu66FKeHhINkB1l7xt7kR8czFcCTUhgoA8Ol2Ltf1eYHOU4SqQRSEitYFDUpqRWcoQ2.dBv_a1Dyu5xa
Enter your Connected App Callback URI (defaults to the sample app's URI):
testsfdc:///mobilesdk/detect/oauth/done
```

Your new solution contains a universal project, as well as NuGet packages for Salesforce Mobile SDK and other dependencies.

Create a Mobile SDK Hybrid Project for Windows 10

Before creating a project, be sure that you've installed the software described in Development Prerequisites for Microsoft Windows.

Note: Mobile SDK for Windows 10 provides limited support for hybrid apps. SmartStore and SmartSync are not supported in hybrid apps.

Create a Hybrid App

1. Make sure that the git installation directory—for example, `C:\Program Files (x86)\Git\bin`—is on your system path.
2. `cd` to a directory where you want to create your hybrid projects.
3. Run the following commands.

```
cordova create <appname>
cd <appname>
cordova platform add windows
cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
cordova prepare
```

What do these Cordova commands do?

- `cordova create` creates a vanilla Cordova app. `appname` is any name you choose.
- `cordova platform add windows` creates a Windows-specific version of your new app in the *appname*`\platforms\windows` directory.
- `cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin` imports the Cordova version of Salesforce Mobile SDK into your project.
- `cordova prepare` copies any edits from your root-level `www` folder into your platform-specific `www` folder.

Note: At the Windows command prompt, clone the repo:

```
https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
```

Build the App

You've created a Cordova app that uses Mobile SDK. Congratulations! Now, let's build it to make sure everything is in place.

1. *cd* to the `platforms/windows` folder and double-click the `CordovaApp.sln` file.

   The solution opens in Visual Studio.

2. Set the Windows 10 project as your startup project.

3. For **Solution Platforms**, select **x86** or **ARM**.
4. In **Build** > **Configuration Manager**, deselect all Build and Deploy options for **CordovaApp.Phone** and **CordovaApp.Windows**.
5. Build the project.

Add Required Files and Run the App

If you tried to run your app after building it, you probably found that it never really got off the ground. That sadness happened because you haven't provided the Salesforce-specific configuration required for authentication.

1. In Windows Explorer or at the command prompt, navigate to your local SalesforceMobileSDK-Windows repo.
2. Drill down to the `SalesforceSDK\TypeScriptLib\data` folder.
3. Copy the `bootconfig.json` and `servers.xml` files and paste them into your *<project_name>*`\www` folder.
4. Run *`cordova prepare`*.
5. In Visual Studio, accept the changes if prompted to do so.
6. Click the **Run** button.

Note:

- The hybrid app you create as described here has no functionality except for Salesforce login. After you're logged in, it merely displays a page with the header "Users".
- For information on Cordova for Windows, see **Documentation** > **Platform Guides** at http://cordova.apache.org.

Add Mobile SDK to an Existing Windows 10 Project

To add Mobile SDK to an existing Windows 10 project, follow these steps.

1. In Visual Studio, open the existing Windows 10 project.
2. Open the Solution Explorer and right-click the solution.
3. Select **Enable NuGet Package Restore** and then click **Yes** when asked if you want to configure NuGet to download and restore missing NuGet packages during build.
4. Right-click the main project and select **Manage NuGet Packages**.
5. Click **Online** and select **All**.
6. Select one or more of the following packages:
   - `SalesforceSDKCore`
   - `SalesforceSmartStore`
   - `SalesforceSmartSync`

   Note: For each package you select, Visual Studio downloads any dependent packages.
7. Configure `app.xaml` and `app.xaml.cs` to derive from `SalesforceApplication`.
   a. In `App.xaml.cs`, change the derivation of your class to the following:

```
public sealed partial class App : SalesforceApplication
```

b. In `App.xaml`, add the following namespace attribute:

```
xmlns:app="using:Salesforce.SDK.App"
```

c. In `App.xaml`, set your app block to `SalesforceApplication`, using the namespace specified in the attribute you added:

```
<app:SalesforceApplication>
...
 xmlns:app="using:Salesforce.SDK.App"
...
</app:SalesforceApplication>
```

8. In `app.xaml.cs`, override `SetRootApplicationPage()` to return the type of your root page.
9. In `app.xaml.cs`, override `InitializeConfig()`. Follow examples from the sample apps in the SalesforceMobileSDK-WindowsSamples GitHub repository.
10. Create a subclass of `SalesforceConfig`. In this class, define your bootstrap configuration, including `ClientId`, `CallbackUrl`, and `Scopes`. For an example, see the configuration in the sample apps.
11. Reference the `Salesforce.SDK.Core` library.
12. When you create an application, use `NativeMainPage` instead of `Page`. The `NativeMainPage` class helps to maintain the user's authentication state.

    Note: If you can't use `NativeMainPage` for any reason, replicate the few things that `NativeMainPage` does in your own `Page` implementation.

13. Clean and build all projects.

Windows Sample Apps

Mobile SDK for Windows 10 provides sample apps that demonstrate major Mobile SDK features.

Note: This release contains an open-source beta version of Mobile SDK for Windows that is still being tested for Windows 10. You can provide feedback and suggestions in the SalesforceMobileSDK Google+ community.

You can access and build the sample app projects through the `SalesforceSDK` solution. Source code for native sample apps lives in the SalesforceMobileSDK-WindowsSamples GitHub repo.

- **RestExplorer:** Demonstrates the OAuth and REST API functions of Salesforce Mobile SDK.
- **SmartSyncExplorer:** Demonstrates the power of the native SmartSync library for offline productivity.

You can find HTML, JavaScript, and CSS source for hybrid web apps in the `samples` folder of the SalesforceMobileSDK-Shared GitHub repo.

NATIVE WINDOWS DEVELOPMENT

Salesforce Mobile SDK provides the tools you need to build full-featured, flexible native apps for Windows 10 devices.

Major 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 managing user data offline
- SmartSync library for offline data synchronization

The native Windows SDK requires you to be proficient in C# coding. You also need to be familiar with Windows 10 application development principles and frameworks. If you're a newbie, consult the Microsoft Developer Network to begin learning. See Development Prerequisites for Microsoft Windows for additional requirements.

Note: Mobile SDK for Windows 10 provides limited support for hybrid apps. SmartStore and SmartSync are not supported in hybrid apps.

Mobile SDK Native Libraries
Mobile SDK for Windows 10 provides downloadable NuGet packages for each of its libraries. You can use these to either create projects or add modules to existing projects.

Developing a Native Windows 10 App

Mobile SDK Native Libraries

Mobile SDK for Windows 10 provides downloadable NuGet packages for each of its libraries. You can use these to either create projects or add modules to existing projects.

NuGet packages include:

Salesforce.SDK.Core

The Salesforce SDK Core package is necessary for writing Salesforce applications on Windows. It contains the base features for OAuth, security, account management, and more. This is a dependency for all Salesforce application development, and must be included along with platform specific libraries for Universal Windows development.

Salesforce.SDK.SmartStore

This library is necessary for using SmartStore with a Salesforce-powered application. You can learn how to use SmartStore by studying the sample code in the github.com/forcedotcom/SalesforceMobileSDK-WindowsSamples GitHub repository.

This library requires SQLite for Visual Studio (http://sqlite.org/2015/sqlite-uwp-3150000.vsix)

Salesforce.SDK.SmartSync

This library is necessary for using SmartSync with a Salesforce-powered application. You can find more documentation by looking at the sample code in the GitHub repository atgithub.com/forcedotcom/SalesforceMobileSDK-Windows.

Prerequisites:

- SQLite for Visual Studio (http://sqlite.org/2015/sqlite-uwp-3150000.vsix)
- After installing SQLite, add a NuGet reference for MSOpenTech SQLitePCL to your project.

- SmartSync requires `Salesforce.SDK.SmartStore`.

SEE ALSO:

Using SmartStore to Store Offline Data

Developing a Native Windows 10 App

To get started with native Mobile SDK development on Windows, first learn about native authentication and application flow. Once you have grasped the framework of a working app, continue to the functional basis of all Mobile SDK apps—the REST API classes.

Note: Mobile SDK provides only the basic UI classes required for bootstrap initialization of a Windows 10 app. You're responsible for designing and developing your app's user interface.

About Login and Passcodes

Overview of Application Flow

Creating Pages with Authentication Support

About Salesforce REST APIs

About Login and Passcodes

To access Salesforce objects from a Mobile SDK app, the customer logs in to 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. 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, it's important to understand the login flow and how OAuth tokens are handled. See About PIN Security and OAuth 2.0 Authentication Flow.

Note: Mobile SDK for iOS supports the use of Touch ID to supply the PIN. Customers must type the PIN when first launching the app. After first launch, the app prompts the customer to use either Touch ID or the keyboard to enter the PIN.

Overview of Application Flow

Native Windows apps built with Mobile SDK follow the same design as other Windows apps. When you create a project with the `createapp,ps1` script, your new app contains three class files:

- `App.xaml`
- `MainPage.xaml`
- `Config.cs`

. `App.xaml` is a Mobile SDK adaptation of the default `App.xaml` file. The Salesforce version creates a `SalesforceApplication` object and sets it as the root object for the rest of the application. `SalesforceApplication` acts as the organizer for the application launch flow. This class coordinates Salesforce authentication and passcode activities in coordination with the `AccountManager` and `PincodeManager`. If no user has been authenticated, `SalesforceApplication` shows the account manager screens. After the user is authenticated, `SalesforceApplication` passes control to either `MainPage.xaml` or another page that you've declared as your app's root entry point.

Note: The workflow demonstrated by the template app is merely an example. You can tailor your `App.xaml` to do what you want by implementing `SalesforceApplication` and supporting classes to achieve your desired workflow. For example, you can postpone Salesforce authentication until a later point. Deferring authentication is easier if your root page does not implement `NativeMainPage`, which helps sustain a logged-in state.

SalesforceApplication Class

SalesforceConfig Class

SalesforceApplication Class

`SalesforceApplication` is the required base for your `App.xaml` implementation. It performs the following tasks:

- Combines app identity and bootstrap configuration
- Provides initialization of the `SalesforceConfig`
- Implements authentication features, including:
  - Authentication flow
  - Pincode management
  - Authentication support for backgrounding and foregrounding events
- Declares the root page where your users land after authentication
- Maintains a `ClientManager` reference. You use this object to obtain REST clients for calling REST APIs, and for logging out users.
- Ensures proper sequencing during the complicated interactions between the application, the operating system, and Salesforce

The Mobile SDK template fills in most of the required code so that you can focus on your starting page and other customizations. If you're adding the SDK to a pre-existing application, you can import the generated code by referencing the `App.xaml.cs` file provided by the template.

Using the generated template files doesn't limit your development freedom. You can still customize your main `App` class to your requirements. You can perform further customization through the `SalesforceConfig` class.

Example: Use the following code in the `App.xaml.cs` file if you didn't create your app with the Mobile SDK template.

```
protected override SalesforceConfig InitializeConfig()
{
    var settings = new EncryptionSettings(new HmacSHA256KeyGenerator());
    Encryptor.init(settings);
    var config = SalesforceConfig.RetrieveConfig<Config>();
    if (config == null)
        config = new Config();
    return config;
}

protected override Type SetRootApplicationPage()
{
    return typeof (MainPage);
}
```

SalesforceConfig Class

You configure your application's Salesforce settings by creating a class that implements `SalesforceConfig`. The template app includes a generated version of the `Config.cs` file in the `settings` folder of the shared project. To connect with Salesforce, implement a few key items and an additional set of properties that allow you to customize the authentication screens of your application.

ClientId
This property corresponds to the consumer key in your connected app.

CallbackUrl
This property corresponds to the callback URL in your connected app.

Scopes
This property corresponds to the OAuth scopes your app requires. These scopes match some or all of the scopes selected in the connected app. For example, `{ "api", "web" }`.

LoginBackgroundColor
This property determines the background color of the login screen.

LoginBackgroundLogo
This property is used to provide a customized logo for the login screen.

ApplicationTitle
This is the application name displayed on the login screen.

About Application Pages

If you create a page class that enforces a logged in state, implement `Salesforce.SDK.Native.NativeMainPage` or `ISalesforcePage` instead of `Page`. `NativeMainPage` provides methods that check the logged in status and kick off the login flow if the user is no longer authenticated.

Example:

```
class Config : SalesforceConfig
{
    public override string ClientId
    {
        get { return "{Replace with connected app client id}"; }
    }

    public override string CallbackUrl
    {
       get { return "{Replace with the callback url from your connected app settings}";
 }
    }

    public override string[] Scopes
    {
        get { return new string[] { "api", "web" }; }
    }

    public override Color? LoginBackgroundColor
    {
        get { return Colors.DarkSeaGreen; }
    }
```

```
    public override string ApplicationTitle
    {
        get { return "My Salesforce Application"; }
    }

    public override Uri LoginBackgroundLogo
    {
        get { return null; }
    }
}
```

Creating Pages with Authentication Support

If you plan to create a page that enforces login, implement `Salesforce.SDK.Native.NativeMainPage` or `ISalesforcePage` instead of `Page`. `NativeMainPage` provides methods that check for login status and kick off the login flow if the user is no longer authenticated.

About Salesforce REST APIs

To query, describe, create, or update Salesforce data, 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 in the Force.com REST API Developer's Guide and in other developer's guides at https://developer.salesforce.com/docs/.

REST API Classes

Windows native apps require minimal coding to access Salesforce records through REST calls. Classes in the `Salesforce.SDK.Rest` namespace initialize 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 Mobile SDK.

`RestClient`
Handles protocol for sending REST API requests to the Salesforce server. Your app never directly creates instances of `RestClient`. Instead, it calls the `ClientManager.GetRestClient()` method implemented by Mobile SDK.

`RestRequest`
Formats REST API requests from the data your app provides. Also serves as a factory for instances of itself. Your app never directly create instances of `RestRequest`. Instead, it calls an appropriate `RestRequest` static getter function such as `RestRequest.GetRequestForCreate()`. Implemented by Mobile 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 all `RestResponse` instances and returns them to your app through your implementation of the `RestClient.AsyncRequestCallback` interface. Your app never creates instances of `RestResponse`. Implemented by Mobile SDK.

Use REST Classes

Here's the basic procedure for using the REST classes.

1. Create an instance of `ClientManager`. This is automatically done by Mobile SDK through `SalesforceApplication`. You can access it through `SalesforceApplication.GlobalClientManager`.
2. Call `SalesforceApplication.GlobalClientManager.GetRestClient()` to get a `RestClient` object.
3. Create a `RestRequest` object.

   ```
   var request = RestRequest.GetRequestForDescribe(apiVersion, objectType);
   ```

4. Pass the `RestRequest` object to the client.

   ```
   RestResponse response = await client.SendAsync(request);
   ```

   At this point you've created a request and sent it to Salesforce. When the `SendAsync()` method returns, you receive the response and assign it to the `response` variable.
5. Handle the response. The `RestResponse` class defines methods and properties to help you parse the results of the response.

Supported REST Requests
The `RestRequest` class provides wrapper methods for standard Salesforce CRUD operations.
Obtaining a RestClient Instance
Obtaining an Unauthenticated RestClient Instance
Create a REST Request
Send a REST Request
Process the REST Response

Supported REST Requests

The `RestRequest` class provides wrapper methods for standard Salesforce CRUD operations.

Mobile SDK REST APIs supports the standard requests offered by Salesforce REST and SOAP APIs. To obtain a formatted `RestRequest` object, call the `RestRequest` static factory method that fits your needs. Each factory method returns a `RestRequest` instance.

Operation	Static factory method
SOQL query Executes the given SOQL string and returns the resulting data set	`GetRequestForQuery`
SOSL search Executes the given SOSL string and returns the resulting data set	`GetRequestForSearch`
Metadata Returns the object's metadata	`GetRequestForMetadata`
Describe global	`GetRequestForDescribeGlobal`

Operation	Static factory method
Returns a list of all available objects in your organization and their metadata	
Describe object type Returns a complete description of all metadata for a single object type	`GetRequestForDescribe`
Retrieve Retrieves a single record by object ID	`GetRequestForRetrieve`
Update Updates a record with the given data	`GetRequestForUpdate`
Upsert Updates or inserts a record from external data, if and only if the given external ID matches the value of the external ID field	`GetRequestForUpsert`
Create Creates a record of the specified object type	`GetRequestForCreate`
Delete Deletes the object of the given type with the given ID	`GetRequestForDelete`
Versions Returns summary information about each Salesforce version currently available	`GetRequestForVersions`
Resources Returns available resources for the specified API version, including resource name and URI	`GetRequestForResources`

Obtaining a RestClient Instance

Mobile SDK provides the `RestClient` class for sending REST requests. You configure service endpoints and payloads in a `RestRequest` object. You then pass that object to a `RestClient.sendAsync()` method to deliver the request to the server. All REST responses are delivered to the app asynchronously.

To get a `RestClient` instance, call `GetRestClient()` on the global client manager.

Example:

```
RestClient rc = SDKManager.GlobalClientManager.GetRestClient();
```

Obtaining an Unauthenticated RestClient Instance

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` instance on Windows, call `GetUnAuthenticatedRestClient()` on the global client manager. Pass in the unauthenticated endpoint URL.

Example:

```
string url = "https://www.acme.com/services/[path_to_some_service]";
RestClient rc = SDKManager.GlobalClientManager.GetUnAuthenticatedRestClient(url);
```

Create a REST Request

To create a REST request for a standard Salesforce endpoint, call a static factory method on the `RestRequest` class.

- ```
  public static RestRequest
  GetRequestForVersions();
  ```
- ```
  public static RestRequest
  GetRequestForResources(string apiVersion);
  ```
- ```
  public static RestRequest
  GetRequestForDescribeGlobal(string apiVersion);
  ```
- ```
  public static RestRequest
  GetRequestForMetadata(string apiVersion, string objectType);
  ```
- ```
  public static RestRequest
  GetRequestForDescribe(string apiVersion, string objectType);
  ```
- ```
  public static RestRequest
  GetRequestForCreate(string apiVersion, string objectType,
      Dictionary<string, object> fields);
  ```
- ```
  public static RestRequest
  GetRequestForRetrieve(string apiVersion, string objectType,
      string objectId, string[] fieldsList);
  ```
- ```
  public static RestRequest
  GetRequestForUpdate(string apiVersion, string objectType,
      string objectId, Dictionary<string, object> fields);
  ```
- ```
  public static RestRequest
  GetRequestForUpsert(string apiVersion, string objectType,
      string externalIdField, string externalId,
      Dictionary<string, object> fields);
  ```
- ```
  public static RestRequest
  GetRequestForDelete(string apiVersion, string objectType, string objectId);
  ```
- ```
  public static RestRequest
  GetRequestForSearch(string apiVersion, string q);
  ```
- ```
  public static RestRequest
  GetRequestForQuery(string apiVersion, string q);
  ```

Example: In `RestRequest` factory methods:

- `apiVersion` is a string, such as "v30.0".
- `objectType` is the name of the record type, such as "Contact".
- `objectId` is the SalesforceSalesforce ID of the record, taken from the Id field.

```
RestRequest request = RestRequest.GetRequestForResources("v30.0");
```

```
RestRequest request = RestRequest.GetRequestForMetadata("v30.0", "Contact");
```

```
RestRequest request = RestRequest.GetRequestForDelete("v30.0", "Contact", OBJECT_ID);
```

To create a new record, you provide a `Dictionary` object that contains `<fieldname>,<value>` pairs expressed as `<string, object>`.

```
var result = new Dictionary<string, object>();
RestRequest request = RestRequest.GetRequestForCreate("v30.0", "Contact",
result.Add("Name","acme"));
```

To retrieve records from the server, you provide a JSON string that contains the list of `<fieldname>,<value>` pairs.

```
RestRequest request = RestRequest.GetRequestForRetrieve("v30.0", "Contact", OBJECT_ID,
 "{\"Name\":\"acme\"}");
```

To update a record on the server, you provide a JSON string that contains the list of `<fieldname>:<value>` pairs.

```
RestRequest request = RestRequest.GetRequestForUpdate("v30.0", "Contact", OBJECT_ID,
"{\"Name\":\"acme\"}");
```

To upsert a record from SmartStore to Salesforce that originated in an external data source, you provide the local record's external ID and the field that contains the external ID. This extra data enables Mobile SDK to find the record in subsequent sync down operations. You also provide the updated field values as a JSON string that contains the list of `<fieldname>:<value>` pairs.

```
RestRequest request = RestRequest.GetRequestForUpsert("v30.0", "Contact",
EXTERNAL_ID_FIELD, EXTERNAL_ID, "{\"Name\":\"acme\"}");
```

For SOQL and SOSL requests, the `q` parameter contains the query string.

```
RestRequest request = RestRequest.GetRequestForQuery("v30.0", "SELECT Id,Name FROM
Account");
```

```
RestRequest request = RestRequest.GetRequestForSearch("v30.0", "FIND {acme*}");
```

Creating REST Requests Manually

Creating REST Requests Manually

For most REST interactions with Salesforce, use the `RestRequest` factory methods. These convenience method provide the easiest and most error-free means of creating the `RestRequest` object you require. However, if none of the factory methods fit your needs, you can manually create `RestRequest` objects using the following public constructors:

```
public RestRequest(HttpMethod method, string path)
           : this(method, path, null, ContentTypeValues.None, null);

public RestRequest(HttpMethod method, string path, string requestBody)
    : this(method, path, requestBody, ContentTypeValues.FormUrlEncoded, null);

public RestRequest(HttpMethod method, string path, string requestBody, ContentTypeValues
contentType)
    : this(method, path, requestBody, contentType, null);
```

```
public RestRequest(HttpMethod method, string path, string requestBody, ContentTypeValues
contentType,
    Dictionary<string, string> additionalHeaders);
```

For these constructors, the minimum amount of data you provide is:

- The HTTP method—GET, POST, and so on—expressed using the appropriate `HttpMethod` static method. For example:

  ```
  HttpMethod.Post
  ```

- The URI path to the service. This string is a relative path to a Salesforce REST endpoint. It is automatically resolved against the host of the user's current Salesforce instance.

Send a REST Request

After you've generated your `RestRequest` object, sent it to one of the `RestClient.SendAsync` methods that takes a `RestRequest` object. The `RestRequest` class defines two of these methods:

```
public async Task<RestResponse> SendAsync(RestRequest request);
public async void SendAsync(RestRequest request, AsyncRequestCallback callback);
```

These methods differ only in how they handle the asynchronous response. The first version returns the response as an asynchronous `Task` object. This `Task` executes when the `SendAsync()` method asynchronously resumes. To the caller of `SendAsync()`, the return type is `RestResponse`. The second version takes a callback method of type `AsyncRequestCallback`. Use this callback method to handle the REST result.

Use the third `sendAsync()` method only for requests you've formatted manually:

```
public async Task<RestResponse> SendAsync(HttpMethod method, string url);
```

Example: Because all REST requests are asynchronous in Mobile SDK for Windows apps, always call the `SendAsync()` method from an asynchronous context. Use the `await` keyword to suspend the calling method's execution until the `SendAsync()` method returns. For example, the RestExplorer sample app channels every request through an asynchronous `Execute()` method:

```
public async void Execute(object parameter)
{
    RestClient rc = SalesforceApplication.GlobalClientManager.GetRestClient();
    if (rc != null)
    {
        RestRequest request = BuildRestRequest();
        RestResponse response = await rc.SendAsync(request);
        _vm.ReturnedRestResponse = response;
    }
}
```

Process the REST Response

Use the following `RestResponse` status properties to obtain information about the request status. If `response.Success` evaluates to false, you can use the `StatusCode` and `Error` properties to get information about the failure.

RestRequest member	Description
`response.Success`	Returns true if the call succeeded.
`response.StatusCode`	Returns the `HttpStatusCode` of the call.
`response.Error`	Contains an `Exception` if an error occurred. Otherwise, returns null.

If `response.Success` evaluates to true, the request succeeded. Use one of the following getter properties to obtain the response in the format you prefer. You can then handle the result to suit your requirements. To access the `JObject` and `JArray` properties, use the NewtonSoft JSON namespaces in your source files. For example:

```
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
```

RestRequest member	Description
`response.AsString`	Returns the body of the response as a string.
`response.AsJArray`	Returns the body of the response as an instance of `JArray`.
`response.AsJObject`	Returns the body of the response as an instance of `JObject`.
`response.PrettyBody`	Returns the body of the response (`JArray` or `JObject`) with indented formatting.

HTML5 AND HYBRID DEVELOPMENT

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 a standalone HTML6 app and distribute it in the Microsoft® Store®, 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 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;
    }
}
```

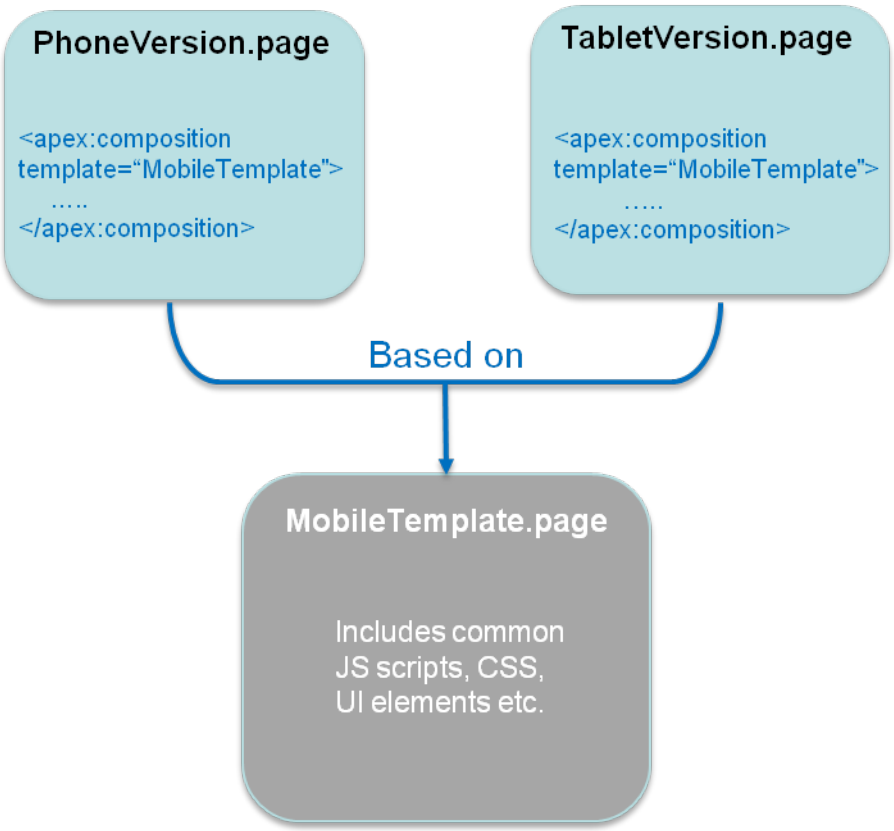
Note that User-Agent parsing in the code snippet above is far from comprehensive and you should implement something more robust that detects all the devices that you need to support based on regular expression matching. A good place to start is to look at the RegEx included in the detectmobilebrowsers.com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

HTML5 Development Tools

Modern Web developers frequently leverage open source tools to speed up their app development cycles. These tools can make HTML5 coding surprisingly simple. For example, to create Salesforce-enabled apps in only a few hours, you can couple Google's Polymer framework with Force.com JavaScript libraries. Salesforce provides a beta open source library—Mobile UI Elements—that does exactly that.

To investigate and get started with Mobile UI Elements, see Mobile UI Elements with Polymer.

Mobile UI Elements with Polymer

Happy mobile app developers spend their time creating innovative functionality—not writing yet another detail page bound to a set of APIs. The Salesforce Mobile UI Elements library wraps Force.com APIs in Google's Polymer framework for rapid HTML5 development.

Mobile UI Elements empower HTML and JavaScript developers to build powerful Salesforce mobile apps with technologies they already know. The open source Mobile UI Elements project provides a pre-built component library that is 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 at github.com/ForceDotComLabs/mobile-ui-elements.

Third-Party Code

The Mobile UI Elements library uses 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 use 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.

See github.com/ForceDotComLabs/mobile-ui-elements for a catalog of currently available elements.

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 `force.js`.

Using JavaScript Remoting to Invoke Your Apex Controller

Apex supports the following two means of invoking Apex controller methods from JavaScript:

- `apex:actionFunction`
- JavaScript remoting

Both techniques use an AJAX request to invoke Apex controller methods directly from JavaScript. The JavaScript code must be hosted on a Visualforce page.

In comparison to `apex:actionFunction`, 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 https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_RemoteAction.htm to learn more about `@RemoteAction` annotations.

Accessing the Salesforce API with Force.js

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

1. Create an archive file, such as a ZIP file, that contains `app.js`, `force.js`, and any other static resources your project requires.

   If you're developing for Android 19 and using Mobile SDK promise-based APIs, include this file: https://www.promisejs.org/polyfills/promise-7.0.4.min.js. Mobile SDK promised-based APIs include:

   - The `smartstoreclient` Cordova plugin (`com.salesforce.plugin.smartstore.client`)
   - `force+promise.js`
   - `smartsync.js`

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 in a Salesforce Mobile SDK environment. Mobile SDKuses Cordova containers for creating standalone hybrid apps. You create a create a Cordova app, add the SalesforceMobileSDK plugin to it, and then add your web app code. At that point you can add support for various platforms, such as Windows, Android, and iOS. With hybrid apps you can reuse any HTML5 and Visualforce code that you've developed for the desktop.

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

1. Make sure you have installed the software listed at Development Prerequisites for Microsoft Windows.
2. If you don't already have a connected app, see Creating 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`.

3. Create and run a hybrid app as described at Create a Mobile SDK Hybrid Project for Windows 10. Use *`hybrid_local`* for the application type.

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 `force.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. Container 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 5.0 provides a hybrid container that is compatible with Apache Cordova 4.4 or later. Architecturally, Mobile SDK hybrid apps are Cordova apps that use Salesforce Mobile SDK as a Cordova plug-in. Cordova provides a simple command line tool for updating the plug-in in an app.

Before creating a project, be sure that you've installed the software described in Development Prerequisites for Microsoft Windows.

Note: Mobile SDK for Windows 10 provides limited support for hybrid apps. SmartStore and SmartSync are not supported in hybrid apps.

Create a Hybrid App

1. Make sure that the git installation directory—for example, `C:\Program Files (x86)\Git\bin`—is on your system path.
2. `cd` to a directory where you want to create your hybrid projects.
3. Run the following commands.

```
cordova create <appname>
cd <appname>
cordova platform add windows
cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
cordova prepare
```

What do these Cordova commands do?

- `cordova create` creates a vanilla Cordova app. `appname` is any name you choose.
- `cordova platform add windows` creates a Windows-specific version of your new app in the *appname*`\platforms\windows` directory.
- `cordova plugin add https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin` imports the Cordova version of Salesforce Mobile SDK into your project.
- `cordova prepare` copies any edits from your root-level `www` folder into your platform-specific `www` folder.

Note: At the Windows command prompt, clone the repo:

```
https://github.com/forcedotcom/SalesforceMobileSDK-CordovaPlugin
```

Build the App

You've created a Cordova app that uses Mobile SDK. Congratulations! Now, let's build it to make sure everything is in place.

1. *cd* to the `platforms/windows` folder and double-click the `CordovaApp.sln` file.

   The solution opens in Visual Studio.
2. Set the Windows 10 project as your startup project.
3. For **Solution Platforms**, select **x86** or **ARM**.
4. In **Build** > **Configuration Manager**, deselect all Build and Deploy options for **CordovaApp.Phone** and **CordovaApp.Windows**.
5. Build the project.

Add Required Files and Run the App

If you tried to run your app after building it, you probably found that it never really got off the ground. That sadness happened because you haven't provided the Salesforce-specific configuration required for authentication.

1. In Windows Explorer or at the command prompt, navigate to your local SalesforceMobileSDK-Windows repo.
2. Drill down to the `SalesforceSDK\TypeScriptLib\data` folder.
3. Copy the `bootconfig.json` and `servers.xml` files and paste them into your `<project_name>\www` folder.
4. Run *`cordova prepare`*.
5. In Visual Studio, accept the changes if prompted to do so.
6. Click the **Run** button.

Note:

- The hybrid app you create as described here has no functionality except for Salesforce login. After you're logged in, it merely displays a page with the header "Users".
- For information on Cordova for Windows, see **Documentation** > **Platform Guides** at http://cordova.apache.org.

Controlling the Status Bar in Hybrid Apps

By default, Mobile SDK hybrid apps show the status bar, which overlays the web view in Salesforce Mobile SDK 2.3 and later. 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.

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 plug-in 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`
- `force.js`

Defer Login

Mobile SDK hybrid apps always present a Salesforce login screen at startup. Sometimes, however, these apps can benefit from deferring authentication until some later point.With a little configuration, you can defer login to any logical place in your app.

Deferred login implementation with `force.js` is a two-step process:

1. Configure the project to skip authentication at startup.
2. In your JavaScript code, call the `force.init()` function, followed by the `force.login()` function, at the point where you plan to initiate authentication.

Step 1: Configure the Project to Skip Authentication

1. In your platform-specific project, open the `www/bootconfig.json` file.
2. Set the `shouldAuthenticate` property to "false".

Step 2: Initiate Authentication in JavaScript

To initiate the authentication process, call the `force.js login()` functions at the point of deferred login. The `force.init()` method is usually necessary only for testing or other non-production scenarios.

```
/* Do login */
force.login(
    function() {
        console.log("Auth succeeded");
        // Call your app's entry point
        // ...
    },
    function(error) {
        console.log("Auth failed: " + error);
    }
);
```

The `force.login()` function takes two arguments: a success callback function and a failure callback function. If authentication fails, your failure callback is invoked. If authentication succeeds, the `force` object caches the access token in its `oauth.access_token` configuration property and invokes your success callback.

OFFLINE MANAGEMENT

Salesforce Mobile SDK provides two modules that help you store and synchronize data for offline use:

- SmartStore lets you store app data in 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.

Using SmartSync to Access Salesforce Objects

Using SmartStore to 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 solution for offline storage on mobile devices. With SmartStore, your users can continue working with data even when the device loses connectivity.

Note:

- Mobile SDK for Windows supports SmartStore in native apps only. SmartStore is not supported for hybrid apps using the Windows platform.
- Mobile SDK for Windows does not implement SmartStore encryption "out of the box". To ensure a secure offline experience, we encourage developers to purchase a license for a product such as SQLCipher for Windows, and add their own encryption.

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.

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)

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

Adding SmartStore to Native Windows Apps

Native apps created with the Windows `createapp.ps1` PowerShell script automatically include SmartStore. If you manually created a Mobile SDK for Windows app, you can add SmartStore by using Mobile SDK from GitHub. You can either reference the binary DLL files or import the open source projects for SmartStore.

To use the Mobile SDK binaries:

1. In a command prompt window, clone the github.com/forcedotcom/SalesforceMobileSDK-Windows repository from GitHub:

   ```
   git clone https://github.com/forcedotcom/SalesforceMobileSDK-Windows.git
   ```

2. Open your solution in Visual Studio, and then open Solution Explorer.
3. In one of your projects, right-click **References**, then click **Add Reference...**.
4. Browse and select the `Salesforce.SDK.SmartStore.dll` library under `<path_to_your_clone>\SalesforceSDK\Dlls\[Release | Debug]`.
5. Click **OK**.
6. Repeat steps 3-5 for every other project in your solution.

To use the open source projects instead of the binaries:

1. In a command prompt window, clone the SalesforceMobileSDK-Windows repository from GitHub:

   ```
   git clone https://github.com/forcedotcom/SalesforceMobileSDK-Windows.git
   ```

2. In Visual Studio, add the following project to your solution:
   - `<path_to_your_clone>\SalesforceMobileSDK-Windows\SalesforceSDK\Salesforce.SDK.SmartStore\Salesforce.SDK.SmartStore.csproj`
3. In Solution Explorer, RIGHT-CLICK **References** in one of your projects, then click **Add Reference...**.
4. Click **Solution** > **Projects**.
5. Select **Salesforce.SDK.SmartStore**.
6. Click **OK**.
7. Repeat steps 3-6 for every other project in your solution.

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.

Windows APIs

In Windows, you access global SmartStore through static `SmartStore` methods.

public static SmartStore GetGlobalSmartStore()
Returns a reference to a new global SmartStore instance.

public static async Task<bool> HasGlobalSmartStore()

Checks if a global SmartStore instance exists. This method runs asynchronously.

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. The following example 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.

Windows Native Apps

For Windows, you define index specs in an array of type `Salesforce.SDK.SmartStore.Store.IndexSpec`. The following example shows the sequence of calls. This code is extracted from the `ContactSyncViewModel` class of the SmartSyncExplorer sample app.

```
using Salesforce.SDK.SmartStore.Store;
...
const string AccountsSoup = "contacts";

private static readonly IndexSpec[] ContactsIndexSpecs = new IndexSpec[]
{
    new IndexSpec("Id", SmartStoreType.SmartString),
    new IndexSpec("FirstName", SmartStoreType.SmartString),
    new IndexSpec("LastName", SmartStoreType.SmartString)
};

private readonly SmartStore _store;

Account account = AccountManager.GetAccount();
if (account == null) return;
_store = SmartStore.GetSmartStore(account);
_store.RegisterSoup(ContactSoup, ContactsIndexSpecs);
```

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.

Windows Native Apps

For REST API interaction, Windows native apps typically use the `RestClient.SendAsync()` method. The following code creates and sends a REST request for the SOQL query `SELECT Name, Id, OwnerId FROM Account`. If the response indicates success,

the code converts the set of returned account records to a `JArray` object. It then upserts each returned record into the Accounts soup.

```
private async void SaveOffline_OnClick(object sender, RoutedEventArgs e)
{
    try
    {
        DisplayProgressFlyout("Loading Remote Data, please wait!");
        await SendRequest("SELECT Name, Id, OwnerId FROM Account", "Account");
    }
    catch
    {
    }
    finally
    {
        MessageFlyout.Hide();
    }
}

private async Task<bool> SendRequest(string soql, string obj)
{
    RestRequest restRequest = RestRequest.GetRequestForQuery(ApiVersion, soql);
    RestClient client = SDKManager.GlobalClientManager.GetRestClient();
    RestResponse response = await client.SendAsync(restRequest);
    if (response.Success)
    {
        var records = response.AsJObject.GetValue("records").ToObject<JArray>();
        if ("Account".Equals(obj, StringComparison.CurrentCultureIgnoreCase))
        {
            _store.InsertAccounts(records);
        }
        else
        {
            /*
            * If the object is not an account,
            * we do nothing. This block can be used to save
            * other types of records.
            */
        }
        return true;
    }
    return false;
}


public const string AccountsSoup = "Account";

public void InsertAccounts(JArray accounts)
{
    if (accounts != null && accounts.Count > 0)
    {
        foreach (var account in accounts.Values<JObject>())
        {
            InsertAccount(account);
        }
```

```
    }
}

public void InsertAccount(JObject account)
{
    if (account != null)
    {
        _store.Upsert(AccountsSoup, account);
    }
}
```

Retrieving Data From a Soup

The SmartStore `QuerySpec` class provides a set of static helper methods that build query strings for you. To query a specific set of records, call the `Build*` method that suits your query specification.

Note: All query builder queries are single-predicate searches. Only Smart SQL queries that you define can support joins.

Query Everything

To query everything in the soup, use this method:

```
public static QuerySpec
BuildAllQuerySpec(string soupName, string path, SqlOrder order,
int pageSize)
```

This method returns all entries in the soup, with no particular order. Use this query to traverse everything in the soup.

The `order` parameter is optional and defaults to ascending. If you specify `pageSize`, you must also specify `order`.

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

To process a Smart SQL query, use this method:

```
public static QuerySpec
BuildSmartQuerySpec(string smartSql, int pageSize)
```

This method executes the query specified by `smartSql`. This method allows greater flexibility than other query factory methods because you provide your own Smart SQL SELECT statement. See Smart SQL Queries.

The following sample code demonstrates a class method that uses the static `QuerySpec.BuildSmartQuerySpec()` method to build a Smart SQL query. It then calls the `SmartStore.CountQuery()` method to find out how many records can be returned, then optimizes the query to obtain the actual number of records.

Windows native:

```
public JArray Query(string smartSql)
{
    SmartStore _store = SmartStore.GetSmartStore();
    JArray result = null;
```

```
    QuerySpec querySpec = QuerySpec.BuildSmartQuerySpec(smartSql, 10);
    var count = (int)_store.CountQuery(querySpec);
    querySpec = QuerySpec.BuildSmartQuerySpec(smartSql, count);
    try
    {
        result = _store.Query(querySpec, 0);
    }
    catch (SmartStoreException e)
    {
        Debug.WriteLine("Error occurred while attempting to run query. Please verify
validity of the query.");
    }
}
```

Query by Exact

To query by an exact value, use this method:

```
public static QuerySpec
BuildExactQuerySpec(string soupName, string path, string exactMatchKey,
int pageSize)
```

This method finds entries that exactly match the given `exactMatchKey` for the `path` value. Use this method 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.

In the following example, "`sfdcId`" is the path to the Salesforce parent ID field, and "some-sfdc-id" is the parent ID value:

```
var querySpec = QuerySpec.BuildExactQuerySpec(
    "Catalogs",
   "sfdcId",
   "some-sfdc-id",
    10);
```

Query by Range

To query by range, use this method:

```
public static QuerySpec
BuildRangeQuerySpec(string soupName, string path, string beginKey,
string endKey, SqlOrder order, int pageSize)
```

This method finds entries whose `path` values fall into the range defined by `beginKey` and `endKey`. Use this method to search by numeric ranges, such as a range of dates stored as integers.

The `order` parameter is optional and defaults to ascending. If you specify `pageSize`, you must also specify `order`.

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 `path` is >= `beginKey`.
- Passing `null` to `beginKey` finds all records where the field at `path` is <= `endKey`.
- Passing `null` to both `beginKey` and `endKey` is the same as querying everything.

Query by Like

To query by partial matching, use this method:

```
public static QuerySpec
BuildLikeQuerySpec(string soupName, string path,
string likeKey, SqlOrder order, int pageSize)
```

This method finds entries whose `path` 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 `path` value. Use this function for general searching and partial name matches. The `order` parameter is optional and defaults to ascending.

Note: Query by Like is the slowest of the query methods.

Executing the Query

Queries return a `JArray` object containing the returned rows. To execute the query, use this method:

```
public JArray Query(QuerySpec querySpec, int pageIndex)
```

Pass your query specification to the `querySpec` parameter. Query results are returned one page at a time. The second parameter, `pageIndex`, is the zero-based index of the results page you're requesting.

```
try
{
    result = _store.Query(querySpec, 0);
}
catch (SmartStoreException e)
{
    Debug.WriteLine("Error occurred while attempting to run query. Please verify validity
 of the query.");
}
```

Counting Records Returned by a Query

The SmartStore interface provides a convenience method that returns the number of records retrieved by a Smart SQL query. Call this method on the global SmartStore object for the current authenticated user. You can use the return value to know how many records you're dealing with before you iterate through them.

```
long CountQuery(QuerySpec querySpec);
```

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). You can use the ID to look up soup entries by combining the `SmartStore.Retrieve()` and `LookupSoupEntryId()` methods. The `Retrieve()` method provides the fastest way to retrieve a soup entry, but you can use it only if you know the soup's ID field name.

The following example shows the required technique applied to a soup named "contacts" (the constant string `ContactSoup`) with ID field "Id" (constant string `Constants.Id`). The `contact` object models a Salesforce Contact record that contains the Salesforce ID in its `ObjectId` member:

```
var item = _store.Retrieve(ContactSoup,
    _store.LookupSoupEntryId(ContactSoup, Constants.Id,
     contact.ObjectId))[0].ToObject<JObject>();
```

Note: When you use the `Retrieve()` method, remember these points:

- Return order is not guaranteed.
- Deleted entries are not included in the resulting array.

Smart SQL Queries

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.

Manipulating Data

To track soup entries for insert, update, and delete actions, 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`, `_soupCreatedDate`—The number of milliseconds since 1/1/1970.
  - To convert a date value 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 you insert or update soup entries, SmartStore automatically sets these fields. When you remove or retrieve specific entries, you can reference them by `_soupEntryId`.

To insert or update soup entries—letting SmartStore determine which action is appropriate—you use an *upsert* method.

Inserting or Updating Soup Entries

If `_soupEntryId` is already set in any of the entries presented for upsert, SmartStore updates the soup entry that matches that ID. If an upsert entry doesn't have a `_soupEntryId` slot, or if the provided `_soupEntryId` doesn't match an existing soup entry, SmartStore inserts the entry into the soup and overwrites its `_soupEntryId`.

Note: Do not directly edit the `_soupEntryId` or `_soupLastModifiedDate` value.

Upserting with an External ID

If your soup entries mirror data from an external system, you usually refer to those entries by their external primary key IDs. For that purpose, SmartStore supports upsert with an external ID. When you perform an upsert, you can designate any index field as the external ID field. SmartStore looks 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, SmartStore creates a soup entry.
- If the external ID field is found, SmartStore updates the entry with the matching external ID value.
- If more than one field matches the external ID, SmartStore returns an error.

To create an entry locally, set the external ID field to a value that you can query when uploading the new entries to the server.

When you update the soup with external data, always use the external ID. Doing so guarantees that you don't end up with duplicate soup entries for the same remote record.

SmartStore also lets you track inter-object relationships. For example, imagine that you create a product offline that belongs to a catalog that doesn't yet exist on the server. You can capture the product's relationship with the catalog entry through the `parentSoupEntryId` field. Once the catalog exists on the server, you can capture the external relationship by updating the local product record's `parentExternalId` field.

Upsert Methods

The Windows `SmartStore` class provides three upsert methods. In these methods:

- `soupName` is the name of the target soup.
- `soupElt` contains one or more entries that match the soup's data structure.
- `externalIdPath` is the index path to the field containing an external ID if the record was imported from a non-Salesforce source.
- If `handleTx` is true, SmartStore handles the upsert in a transactional model. By default, SmartStore uses transactions in upsert operations.

```
JObject Upsert(string soupName, JObject soupElt, string externalIdPath);JObject Upsert(string
 soupName, JObject soupElt);JObject Upsert(string soupName, JObject soupElt, string
externalIdPath, bool handleTx);
```

Managing Soups

SmartStore provides utility functionality that lets you retrieve soup metadata and perform other soup-level operations.

Windows Native Apps

To use soup management APIs in a native SmartStore app, you call methods on the shared SmartStore instance:

```
private readonly SmartStore _store;
_store = SmartStore.GetSmartStore();

_store.ClearSoup("user1Soup");
```

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.

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.

Clear a Soup

To remove all entries from a soup, call the applicable soup clearing method.

Windows Apps

In Windows apps, call:

```
public void ClearSoup (string 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.

Windows Apps

In Windows apps, call:

```
public IndexSpec[] GetSoupIndexSpecs(string soupName);
```

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.

Windows Apps

In Windows native apps, call:

```
public void ReIndexSoup(string soupName, string[] indexPaths, bool handleTx)
```

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.

Windows Apps

In Windows apps, call:

```
public void DropSoup (String soupName)
```

Example: Using SmartStore in Native Windows Apps

This example combines the examples from previous sections to create a SmartStore soup for account records—define index specs, register the soup, upsert records, and query the soup.

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.

```
public class SmartStoreExtension
{
    public const string AccountsSoup = "Account";
    private readonly SmartStore _store;
    public SmartStoreExtension()
    {
        _store = SmartStore.GetSmartStore();
        CreateAccountsSoup();
        //...
    }

    public readonly IndexSpec[] AccountsIndexSpecs = new IndexSpec[]
    {
        new IndexSpec("Name", SmartStoreType.SmartString),
        new IndexSpec("Id", SmartStoreType.SmartString),
        new IndexSpec("OwnerId", SmartStoreType.SmartString)
    };

    public void CreateAccountsSoup()
    {
        _store.RegisterSoup(AccountsSoup, AccountsIndexSpecs);
    }
    //...
}
```

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. To upsert, you store the data in a `JObject` object—a JSON object as defined in the Json.NET framework—and then pass that object to the `SmartStore.Upsert()` method.

You might have obtained the account data by:

- **Collecting your customer's input in your app.** You create a `JObject` and populate it with the field names and their associated data.
- **Sending a REST request to the Salesforce cloud and processing the response.** In your REST response handler, you can use `RestResponse.AsJObject` to obtain the proper format for upserting.
- **Using the `syncDown()` method from the SmartSync library.** The `syncDown()` method automatically handles upserts for you, so you're off the hook.

When the data is ready as a `JObject`, you simply call `SmartStore.Upsert()`.

```
public void InsertAccount(JObject account)
{
    if (account != null)
    {
        _store.Upsert(AccountsSoup, account);
    }
}
```

Running Queries Against SmartStore

Mobile SDK lets you query SmartStore soups with its own Smart SQL query language. Smart SQL queries support advanced queries such as joins and aggregate queries. The syntax of a Smart SQL query differs only slightly from standard SQL For example, a colon (":") serves as the delimiter between a soup name and an index field. Also, a set of curly braces encloses each *<soup-name>*:*<field-name>* pair.

Here's an example of an aggregate query that can 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 Smart SQL queries against soups in a native app. In this example, the `smartSql` parameter contains the query. On line 2, the second parameter of the `BuildSmartQuerySpec()` method specifies the number

of records per page of results. The second argument to the `_store.Query()` method (line 7) represents the zero-based index of a page of results. In this case, you haven't pre-checked the number of records in the query, so it's only sensible to request page 0.

```
public JArray Query(string smartSql)
{
    JArray result = null;
    QuerySpec querySpec = QuerySpec.BuildSmartQuerySpec(smartSql, 10);
    var count = (int)_store.CountQuery(querySpec);
    querySpec = QuerySpec.BuildSmartQuerySpec(smartSql, count);
    try
    {
        result = _store.Query(querySpec, 0);
    }
    catch (SmartStoreException e)
    {
        Debug.WriteLine("Error occurred while attempting to run query. Please verify
validity of the query.");
    }
    return result;
}
```

SEE ALSO:

- Send a REST Request
- Process the REST Response
- Retrieving Data From a Soup
- Smart SQL Queries

Using SmartSync to Access Salesforce Objects

The SmartSync library is a collection of APIs that make it easy for developers to sync data between the Salesforce cloud and their mobile apps. It provides the means for getting and posting data via a server endpoint, caching data on a device, and reading cached data. For sync operations, SmartSync predefines cache policies for fine-tuning synchronization between local data and server data. 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.

Note: Mobile SDK for Windows 10 provides limited support for hybrid apps. SmartStore and SmartSync are not supported in hybrid apps.

Using SmartSync in Native Apps

Before you start coding, let's examine the SmartSync architectural components that all Mobile SDK platforms share.

Using SmartSync in Native Apps

Before you start coding, let's examine the SmartSync architectural components that all Mobile SDK platforms share.

At the highest functional level, SmartSync involves three actions:

- Querying Salesforce objects by calling Salesforce REST APIs.
- Storing retrieved object data and metadata locally for offline use.

- Syncing CRUD operations between SmartStore and the Salesforce cloud when the device goes offline or online.

The first two actions—query and storage—are handled by the Mobile SDK REST and SmartStore libraries, respectively. SmartSync stores metadata on each SmartStore record to reflect whether the user has updated, deleted, or created the record. When a SmartSync app senses a change in connectivity status, it can call SmartSync to perform any necessary synchronization. SmartSync then checks its SmartStore metadata and uses the REST API to perform appropriate synchronizations.

Beyond the basics, SmartSync also lets you:

- 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.
- Select a predefined caching policy to fine-tune synchronization operations.
- Synchronize batches of locally modified records with the Salesforce cloud.

SmartSync Components

The following components form the basis of SmartSync architecture.

Sync Manager

- `Salesforce.SDK.SmartSync.Manager.SyncManager`

  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 and enums:

Class or Enum	Description
• `Salesforce.SDK.SmartSync.Model.SyncState.SyncStatusTypes` enum	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.
• `Salesforce.SDK.SmartSync.Model.SyncTarget`	Base class for custom sync targets.
• `Salesforce.SDK.SmartSync.Model.SyncOptions`	Specifies configuration options for "sync up" and "sync down" operations. Options include the list of field names to be synced.

Metadata Manager

- `Salesforce.SDK.SmartSync.Manager.MetadataManager`

Performs data loading functions. This class handles 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:

- `RestClient`, `RestRequest`

It also uses the SmartSync cache manager to read and write data to the cache.

Cache Manager

- `Salesforce.SDK.SmartSync.Manager.CacheManager`

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

- `Salesforce.SDK.SmartSync.Manager.SOQLBuilder`

Utility class that makes it easy to build a SOQL query statement, by specifying the individual query clauses.

SOSL Builder

- `Salesforce.SDK.SmartSync.Manager.SOSLBuilder`

Utility class that makes it easy to build a SOSL query statement, by specifying the individual query clauses.

Note: To support multi-user switching, SmartSync creates unique instances of its components for each user account.

Cache Policies
Object Representation
Adding SmartSync to Native Apps
Syncing Data
Using Custom Sync Down Targets
Using Custom Sync Up Targets
Storing and Retrieving Cached Data

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 Windows, cache policies are defined in the `Salesforce.SDK.SmartSync.Manager.CacheManager.CachePolicy` class.

You specify a cache policy when you call the following method to ask the `CacheManager` if it's time to reload data.

```
public bool NeedToReloadCache(bool cacheExists, CachePolicy cachePolicy,
    long lastCachedTime,long refreshifOlderThan)
```

Before calling `NeedToReloadCache()`, call one or both of the following `CacheManager` methods:

- ```
  public bool DoesCacheExist(string soupName)
  ```

  Verifies the current existence of the cache.
- ```
  public long GetLastCacheUpdateTime(string cacheType, string
        cacheKey)
  ```

  Returns the soup's "last modified" date.

Here's a list of cache policies.

Table 1: Cache Policies

Cache Policy (Windows)	Description
`IgnoreCacheData`	Ignores cached data. Always goes to the server for fresh data.
`ReloadAndReturnCacheOnFailure`	Attempts to load data from the server, but falls back on cached data if the server call fails.
`ReturnCacheDataDontReload`	Returns data from the cache,\ and doesn't attempt to make a server call.
`ReloadAndReturnCacheData`	Reloads data from the server and updates the cache with the new data.
`ReloadIfExpiredAndReturnCacheData`	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`	Clears the cache and does not reload data from the server.
`InvalidateCacheAndReload`	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 formats:

- Object
- Object Type
- Object Type Layout

Object

- `Salesforce.SDK.SmartSync.Model.SalesforceObject`

This class encapsulates the data that you retrieve from an sObject in Salesforce. The object class reads the data from a `JObject` that contains the results of your query. It then stores the object's ID, type, and name as properties. It also stores the `JObject` itself as raw data.

Object Type

- `Salesforce.SDK.SmartSync.Model.SalesforceObjectType`

The object type class stores details of an sObject, including the prefix, name, label, plural label, and fields.

Object Type Layout

- `Salesforce.SDK.SmartSync.Model.SalesforceObjectTypeLayout`

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.

Adding SmartSync to Native Apps

If you use the createapp.ps PowerShell script to create your native app, SmartStore and SmartSync are automatically included.

If you created your native Windows 10 app manually from GitHub, you can add SmartSync to your Mobile SDK project by cloning the Mobile SDK source code from GitHub. You can then either reference the binary DLL files or import the open source projects for SmartStore and SmartSync.

To use the Mobile SDK binaries:

1. In a command prompt window, clone the SalesforceMobileSDK-Windows repository from GitHub:

   ```
   git clone https://github.com/forcedotcom/SalesforceMobileSDK-Windows.git
   ```

2. Open your solution in Visual Studio, then open Solution Explorer.
3. In one of your projects, right-click **References**, then click **Add Reference...**.
4. Click **Browse** and then select the `Salesforce.SDK.SmartStore.dll` library under *<path_to_your_clone>*`\SalesforceSDK\Dlls\SmartStore\[`*Release* | *Debug*`]`.
5. Click **Browse** and then select the `Salesforce.SDK.SmartSync.dll` library under *<path_to_your_clone>*`\SalesforceSDK\Dlls\SmartSync\[`*Release* | *Debug*`]`.
6. Click **OK**.
7. Repeat steps 3-6 for every other project in your solution.

To use the open source projects instead of the binaries:

1. In a command prompt window, clone the SalesforceMobileSDK-Windows repository from GitHub:

   ```
   git clone https://github.com/forcedotcom/SalesforceMobileSDK-Windows.git
   ```

2. In Visual Studio, add the following projects to your solution:
   - *<path_to_your_clone>*`\SalesforceMobileSDK-Windows\SalesforceSDK\SmartStore\SmartStore.csproj`
   - *<path_to_your_clone>*`\SalesforceMobileSDK-Windows\SalesforceSDK\SmartSync\SmartSync.csproj`
3. In Solution Explorer, right-click **References** on one of your projects, then click **Add Reference...**.
4. Click **Projects** > **Solution**.
5. Select **SmartStore** and **SmartSync**.

6. Click **OK**.
7. Repeat steps 3-6 for every other project in your solution.

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 Windows SmartSync apps are asynchronous. Asynchronous in Windows means that the app does not suspend execution while it waits for the response. Instead, the sync method is suspended, and control returns to its caller. When the server response arrives, the sync method regains control on the line where it was paused. See the Microsoft Developer Network documentation for information on asynchronous programming in C#.

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 or sync up operation, other options).
- Type of operation (up or down).
- Progress percentage (integer, 0–100).
- Total number of records in the transaction.

Using the Sync Manager
Syncing Down
Incrementally Syncing Down
Syncing Up
Using the Sync Manager with Global SmartStore

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 (Windows)

In Windows, you use a different factory method for each of the following scenarios:

For the current user:

```
public static SyncManager GetInstance();
```

For a specified user, optionally in a specified community:

```
public static SyncManager GetInstance(Account account, string communityId = null);
```

For a specified user in a specified community using the specified SmartStore database:

```
public static SyncManager GetInstance(Account account, string communityId,
SmartStore.store.SmartStore smartStore);
```

Syncing Down

To download sObjects from the server to your local SmartSync soup, use the "sync down" method:

- `SyncManager` **methods:**

```
public SyncState SyncDown(SyncDownTarget target, string soupName,
    Action<SyncState> callback, SyncOptions options = null);
```

For "sync down" methods, you define a target that provides the list of sObjects to be downloaded. To provide an explicit list, use `JObject` on Windows. 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 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:
   - `SyncState.MergeModeOptions.Overwrite`
2. Preserve all local changes and locally modified records. Set the `options` parameter to the following value:
   - `SyncState.MergeModeOptions.LeaveIfChanged`

Important: If you call `SyncDown` without specifying an `options` parameter, existing sObjects in the cache can be overwritten. To preserve local changes, always run sync up before running sync down.

Example: The native SmartSyncExplorer sample app demonstrates how to use SmartSync with Contact records. In Windows, it defines a `ContactObject` class that represents a Salesforce Contact record as a C# 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.

```
public void SyncDownContacts()
{
    RegisterSoup();
    if (syncId == -1)
    {
        string soqlQuery =
            SOQLBuilder.GetInstanceWithFields(ContactObject.ContactFields)
                .From(Constants.Contact)
                .Limit(Limit)
                .Build();
        SyncOptions options =
SyncOptions.OptionsForSyncDown(SyncState.MergeModeOptions.LeaveIfChanged);
        SyncDownTarget target = new SoqlSyncDownTarget(soqlQuery);
        try
        {
            SyncState sync = _syncManager.SyncDown(target, ContactSoup,
HandleSyncUpdate);
            syncId = sync.Id;
        }
        catch (SmartStoreException)
        {
            // log here
        }
    }
    else
    {
        _syncManager.ReSync(syncId, HandleSyncUpdate);
    }
}
```

If the sync down operation succeeds—that is, if `SyncState.Status` equals `SyncState.SyncStatusTypes.Done`—the received data goes into the specified soup. The callback method performs some cleanup and "bookkeeping". For a sync up call, it's necessary to sync down (`SyncDownContacts()`) after the operation has completed successfully. After a sync down, the app resets its in-memory model to match the new data in SmartStore.

```
private void HandleSyncUpdate(SyncState sync)
{
    if (SyncState.SyncStatusTypes.Done != sync.Status) return;
    switch (sync.SyncType)
    {
        case SyncState.SyncTypes.SyncUp:
            RemoveDeleted();
            ResetUpdated();
            SyncDownContacts();
            break;
        case SyncState.SyncTypes.SyncDown:
            LoadDataFromSmartStore(false);
            break;
```

```
        }
}
```

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

Windows:

On a `SyncManager` instance, call:

```
public SyncState ReSync(long syncId, Action<SyncState> callback);
```

Sample Apps

Windows

The SmartSyncExplorer sample app uses `reSync()` in the `ContactSyncViewModel` class.

Syncing Up

To apply local changes on the server, use one of the "sync up" methods:

- **Windows `SyncManager` method:**

```
public SyncState SyncUp(SyncUpTarget target, SyncOptions options,
    string soupName, Action<SyncState> callback)
```

This method updates the server with data from the given SmartStore soup. It looks for created, updated, or deleted records in the soup, and then replicate those changes on the server. The `options` argument includes 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 merge mode 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:
   - **Windows:** Set the `options` parameter to `SyncState.MergeModeOptions.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:
   - **Windows:** Set the `options` parameter to `SyncState.MergeModeOptions.LeaveIfChanged`

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 `SyncState.MergeModeOptions.LeaveIfChanged` sync up operation reverts to an overwrite sync up.

Important: The `SyncState.MergeModeOptions.LeaveIfChanged` 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: **Windows:**

When it's time to sync up to the server, you call `SyncUp()` with a `SyncUpTarget` instance, the list of fields, name of source SmartStore soup, and an update callback method. You format the list of affected fields as an instance of `SyncOptions`. Here's the way it's handled in the SmartSyncExplorer sample:

```
public void SyncUpContacts()
{
    RegisterSoup();
    SyncOptions options =
SyncOptions.OptionsForSyncUp(ContactObject.ContactFields.ToList(),
SyncState.MergeModeOptions.LeaveIfChanged);
    var target = new SyncUpTarget();
    _syncManager.SyncUp(target, options, ContactSoup, HandleSyncUpdate);
}
```

In the update callback, the SmartSyncExplorer sample 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 (SyncState.SyncStatusTypes.Done != sync.Status) return;
    switch (sync.SyncType)
    {
        case SyncState.SyncTypes.SyncUp:
            RemoveDeleted();
            ResetUpdated();
            SyncDownContacts();
            break;
        case SyncState.SyncTypes.SyncDown:
            LoadDataFromSmartStore(false);
            break;
    }
}
```

```
private async void RemoveDeleted()
{
    CoreDispatcher core = CoreApplication.MainView.CoreWindow.Dispatcher;
   List<ContactObject> todelete = Contacts.Select(n => n).Where(n => n.IsLocallyModified
 || n.ObjectId.Contains("local")).ToList();
    foreach (var delete in todelete)
    {
        ContactObject delete1 = delete;
        await Task.Delay(10).ContinueWith(async a => await RemoveContact(delete1));
    }
}

private async void ResetUpdated()
{
    CoreDispatcher core = CoreApplication.MainView.CoreWindow.Dispatcher;
    List<ContactObject> updated = Contacts.Select(n => n).Where(n =>
n.UpdatedOrCreated).ToList();
    for (int index = 0; index < updated.Count; index++)
    {
        ContactObject update = updated[index];
        await core.RunAsync(CoreDispatcherPriority.Normal, () =>
        {
            update.UpdatedOrCreated = false;
            update.Deleted = false;
        });
        await UpdateContact(update);
    }
}
```

Using the Sync Manager with Global SmartStore

To use SmartSync with a global SmartStore instance, pass the global SmartStore instance to the following static `SyncManager.GetInstance()` overload:

Windows:

```
public static SyncManager GetInstance(Account account, string communityId,
SmartStore.Store.SmartStore smartStore)
```

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

Invoking the Sync Down Method with a Custom Target

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. The base sync down target class for Windows is:

- `SyncDownTarget`

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.

```
public abstract Task<JArray> StartFetch(SyncManager syncManager, long maxTimeStamp);
```

Continue Fetching Method

Called by the sync manager repeatedly until the method returns null. This process retrieves all records that require syncing.

```
public abstract Task<JArray> ContinueFetch(SyncManager syncManager);
```

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 name is `lastModifiedDate`.

```
public const string ModificationDateFieldName = "modificationDateFieldName";
```

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. The default field name is `Id`.

```
public const string IdFieldName = "idFieldName";
```

Invoking the Sync Down Method with a Custom Target

To use your custom target:

Pass an instance of your custom `SyncDownTarget` class to the `SyncManager` sync down method:

```
public SyncState SyncDown(SyncDownTarget target, string soupName,
    Action<SyncState> callback, SyncOptions options = null);
```

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

Invoking the Sync Up Method with a Custom Target

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:

- **Windows:** `SyncUpTarget`

Every custom target class must implement the following required methods.

Create On Server Method

Sync up a locally created record.

```
public async Task<String> CreateOnServerAsync(SyncManager syncManager, String objectType,

    Dictionary<String, Object> fields);
```

Update On Server Method

Sync up a locally updated record. For the `objectId` parameter, SmartSync uses the field specified in the `getIdFieldName()` method (Windows) of the custom target.

```
public async Task<bool> UpdateOnServer(SyncManager syncManager, String objectType,
    String objectId, Dictionary<String, Object> fields)
```

Delete On Server Method

Sync up a locally deleted record. For the `objectId` parameter, SmartSync uses the field specified in the `IdFieldName()` variable of the custom target.

```
public async Task<bool> DeleteOnServer(SyncManager syncManager, String objectType,
    String objectId)
```

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.

```
public HashSet<String> GetIdsOfRecordsToSyncUp(SyncManager syncManager, String
soupName);
```

Modification Date Field Name

Field used during a `LeaveIfChanged` sync up operation to determine whether a record was remotely modified.

```
public const string ModificationDateFieldName = "modificationDateFieldName";
```

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.

```
public async Task<String> FetchLastModifiedDate(SyncManager syncManager,
    String objectType, String objectId);
```

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.

```
public const string IdFieldName = "idFieldName";
```

Invoking the Sync Up Method with a Custom Target

To use your custom sync up target, pass it to the `SyncUp()` method on a `SyncManager` instance:

```
public SyncState SyncUp(SyncUpTarget target, SyncOptions options, string soupName,
Action<SyncState> callback);
```

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 Class

The cache manager class is `Salesforce.SDK.SmartSync.Manager.CacheManager`.

-

Read and Write Methods

`CacheManager` methods read and write sObject metadata, MRU lists, and sObject layouts.

- **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> objectTypes,
      string cacheKey, string cacheType);
  ```

  sObject Layouts

  ```
  public List<SalesforceObjectTypeLayout> ReadObjectLayouts(string cacheType, string
  cacheKey);
  ```

  ```
  public void WriteObjectLayouts(List<SalesforceObjectTypeLayout> objectLayouts,
      string cacheKey, string cacheType);
  ```

Clearing the Cache

When your app is ready to clear the cache, use the following cache manager method:

```
public void RemoveCache(string soupName);
```

USING COMMUNITIES WITH MOBILE SDK APPS

Salesforce Communities is a social aggregation feature that supersedes the Portal feature of earlier releases. Communities can include up to millions of users, as allowed by Salesforce limits. With proper configuration, your customers 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.

To learn more about the Salesforce communties features, see "Salesforce Communities Overview" in Salesforce Help.

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.

Set Up an API-Enabled Profile

Set Up a Permission Set

Grant API Access to Users

Configure the Login Endpoint

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.

Customize Login, Logout, and Self-Registration Pages in Your Community

Configure the standard login, logout, password management, and self-registration options for your community, or customize the behavior with Apex and Visualforce or Community Builder (Site.com Studio) pages.

Using External Authentication With Communities

You can use an external authentication provider, such as Facebook©, to log community users into your Mobile SDK app.

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.

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.

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 Enable 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 Create 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.

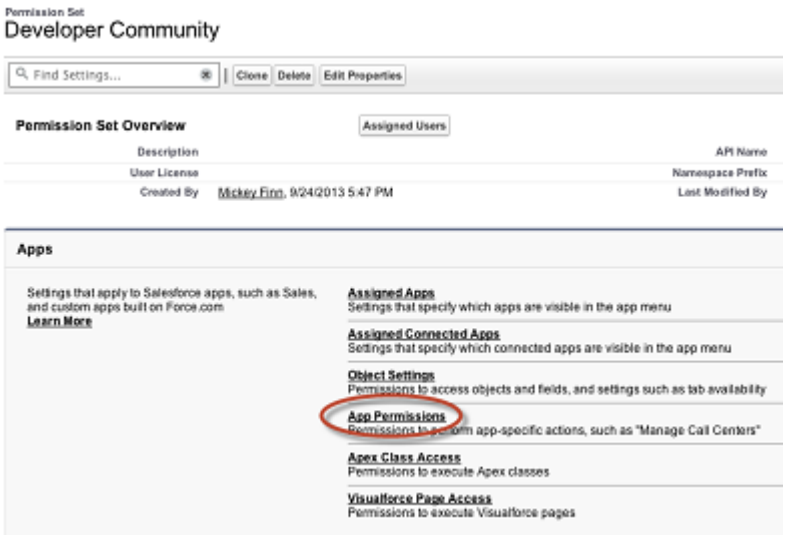
1. Create a new profile or edit an existing one.
2. Edit the profile's details to select API Enabled under **Administrative Permissions**.
3. Save your changes, and then edit your community from Setup by entering *Communities* in the Quick Find box and then selecting **All Communities**.
4. Select the name of your community. Then click **Administration** > **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.

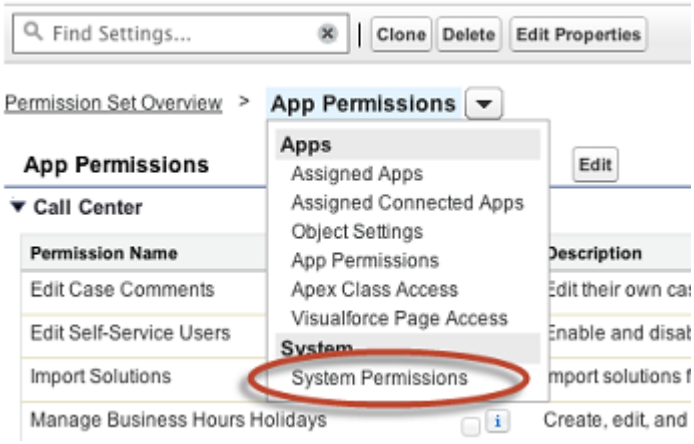
Set Up a Permission Set

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, enter *Permission Sets* in the Quick Find box, then select **Permission Sets**, select the permission set, and skip to Step 6.
2. To create a permission set, in Setup, enter *Permission Sets* in the Quick Find box, then select **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. From Setup, enter *Communities* in the `Quick Find` box, select **All Communities**, and click **Manage** next to your community name.
11. In Administration, 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, you configure the app to use your community login endpoint.

In Windows, Mobile SDK looks for the list of login servers in the `Resources\servers.xml` file of your Shared project. The SalesforceSDK library uses this file to define production and sandbox servers.You can add your servers to the runtime list by creating your own `Resources\servers.xml` file. 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.)

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.

1. Access Community Management in one of the following ways.
   - From the community:
     - In Salesforce Tabs + Visualforce communities, click ⚙ in the global header.
     - In Community Builder-based communities, use the drop-down menu next to your name and click **Community Management**.
   - From Setup, enter *All Communities* in the Quick Find box, then select **All Communities** and click the **Manage** link next to a community.
   - From Community Builder, in the header, use the drop-down menu next to the name of your template and click **Community Management**.
2. Click **AdministrationBranding**.
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.

EDITIONS

Available in: Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

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.

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 Pages in Your Community

Configure the standard login, logout, password management, and self-registration options for your community, or customize the behavior with Apex and Visualforce or Community Builder (Site.com Studio) pages.

By default, each community comes with default login, password management, and self-registration pages and associated Apex controllers that drive this functionality under the hood. You can use Visualforce, Apex, or Community Builder (Site.com Studio) to create custom branding and change the default behavior:

- 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.
- Use custom Change Password and Forgot Password pages
- Set up self-registration for unlicensed guest users in your community.

EDITIONS

Available in: Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

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

Using the Community URL Parameter

Send your user to a specific Community after authenticating.

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.

Configuring a Facebook Authentication Provider

Configure a Salesforce Authentication Provider

You can use a connected app as an authentication provider.

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.

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.

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.

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

USER PERMISSIONS

To view the settings:

- "View Setup and Configuration"

To edit the settings:

- "Customize Application"

  AND

  "Manage Auth. Providers"

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 – 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: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

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: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

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.

Value	Description
`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: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

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, enter *`Auth. Providers`* in the `Quick Find` box, then select **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.

1. Define a Connected App.
2. Define the Salesforce authentication provider in your organization.
3. Test the connection.

EDITIONS

Available in: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

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. From Setup, enter *Apps* in the `Quick Find` box, then select **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 Org

To set up the authentication provider in your org, you need the values from the `Consumer Key` and `Consumer Secret` fields of the connected app definition.

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, enter *Auth. Providers* in the Quick Find box, then select **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 s 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://login.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://login.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 that you created earlier (on the Apps page in Setup, click 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.

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: Lightning Experience and Salesforce Classic

Available in: **Enterprise**, **Performance**, **Unlimited**, and **Developer** Editions

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, enter *`Auth. Providers`* in the `Quick Find` box, then select **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, a user that's not associated with a contact cannot 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.
- When you define a custom login host in an iOS app, be sure to remove the `http[s]://` prefix. The iOS core appends the prefix at runtime. Explicitly including it could result in an invalid address.

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.

Create a Community

Create a community and a community login URL.

Add the API User Profile To Your Community

Add the API User profile to your community setup on the Members page.

Create a New Contact and User

Instead of creating users directly, create a contact on an account and then create the user from that contact.

Test Your New Community Login

Test your community setup by logging in to your Mobile SDK native or hybrid local app as your new contact.

Add Permissions to a Profile

Create a profile that has API Enabled and Enable Chatter permissions.

1. From Setup, enter *Profiles* in the Quick Find box, then select **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 Enable Salesforce Communities and Creating Communities in Salesforce Help.

1. In Setup, enter *Communities* in the Quick Find box.
2. If you don't see **All Communities**:
   a. Click **Communities Settings**.
   b. Select **Enable communities**.
   c. Enter a unique name for your domain 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. From Setup, enter *Communities* in the Quick Find box, then select **All Communities**.
4. Click **New Community**.
5. Choose a template and name the new community *FineApps Users*.
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 Community**, and then click **Go to Community Management**.

Add the API User Profile To Your Community

Add the API User profile to your community setup on the Members page.

1. Click **Administration** > **Members**.
2. For Search, select **All**.
3. Select **FineApps API User** in the Available Profiles list and 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 and 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. From Setup, enter *Users* in the Quick Find box, select **Users**, and then 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** and 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 and 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 in to your Mobile SDK native or hybrid local app as your new contact.

To log in to your community from your Mobile SDK app, configure your app to recognize your community login URL.

1. For Windows:
   a. Open your project in Visual Studio.
   b. In the Project Explorer, go to or create the `Resources` folder and select or create the `servers.xml` file.
   c. In `servers.xml`, add the following content. 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>
```

   d. Save the file.
2. Start your app on your device, simulator, or emulator, and log in to the Fineapps server with username *`jimparker@fineapps.com`* and password *`mobiletest1234`*.

Note: If your mobile app remains at the login screen for an extended time, you can get an "insufficient privileges" error upon login. In this case, close and reopen the app, and 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.

Define a Salesforce Auth. Provider
To enable external authentication in Salesforce, create an Auth. Provider.

Configure Your Facebook App
Next, you need to configure the community to use your Salesforce Auth. Provider for logins.

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.

Configure Your Salesforce Community
For the final step, configure the community to use your Salesforce Auth. Provider for logins.

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, enter *Auth. Providers* in the Quick Find box, then select **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 Changes**.
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, enter *`Apex Classes`* in the `Quick Find` box, then select **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, enter *`Communities`* in the `Quick Find` box, then select **All Communities**.
2. Click **Manage** next to your community name.
3. Click **Administration** > **Login & Registration**.
4. Under Login, 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
```

MULTI-USER SUPPORT IN MOBILE SDK

If you need to enable simultaneous logins for multiple users, Mobile SDK provides a basic implementation for user switching.

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.)

Implementing Multi-User Support

For apps that allow multiple users to remain logged in simultaneously, Mobile SDK provides a multi-user API.

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:

Implementing Multi-User Support

For apps that allow multiple users to remain logged in simultaneously, Mobile SDK provides a multi-user API.

Although Mobile SDK implements the underlying functionality, multi-user switching isn't initialized at runtime unless and until your app calls the following static `AccountManager` API:

```
AccountManager.SwitchAccount();
```

Apps built from the Mobile SDK template automatically call this API when the current user logs out. To allow user switching from other places in your app, you call `AccountManager.SwitchAccount()` 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 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 multi-user feature, 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:

- SmartStore Soups (if your app uses SmartStore)

- Account Management

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

AUTHENTICATION, SECURITY, AND IDENTITY IN MOBILE APPS

Secure authentication is essential for enterprise applications running on mobile devices. OAuth 2.0, the industry-standard protocol, enables secure authorization for access to a customer'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 OAuth 2.0 implementation. The implementation uses an HTML view to collect the username and password, which are then sent to the server. The server returns a session token and a persistent refresh token that are stored on the device for future interactions.

A Salesforce *connected app* is the primary means by which a mobile app 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 restrict access to a set of customers, set or relax an IP range, and so on.

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.

Consumer Key
A value used by the consumer—in this case, the Mobile SDK app—to identify itself to Salesforce. Referred to as `client_id`.

Consumer Secret
A secret that the consumer uses to verify ownership of the consumer key. To heighten security, Mobile SDK apps do not use the consumer secret.

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. Remote access applications have been deprecated in favor of connected apps.

OAuth 2.0 Authentication Flow

The authentication flow depends on the state of authentication on the device.

First Time Authorization 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 Authorization

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

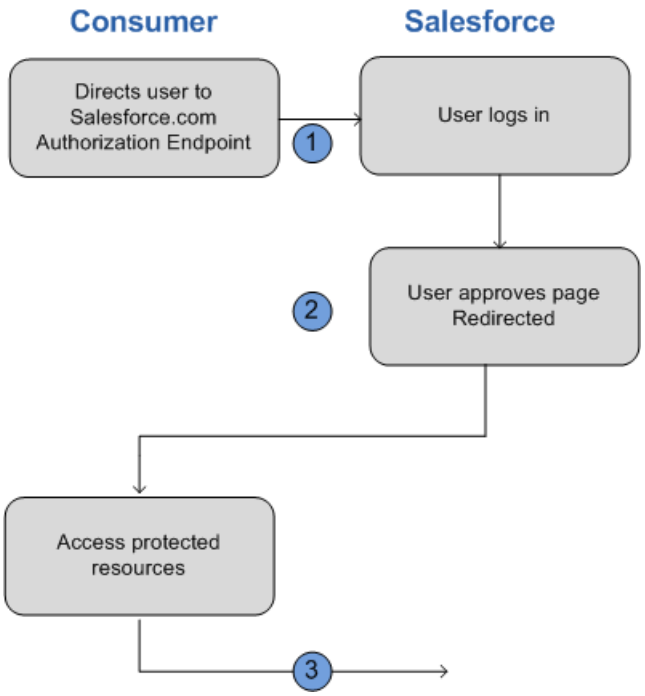
The user-agent authentication flow is used by client applications residing on the user's mobile device. The authentication is based on the user-agent's same-origin policy.

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.
`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.
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`

Parameter	Description
	• `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.
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.

  - `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.

  - `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>https://yourInstance.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>https://yourInstance.salesforce.com/profilephoto/005/F</picture>
   <thumbnail>https://yourInstance.salesforce.com/profilephoto/005/T</thumbnail>
</photos>
<urls>
   <enterprise>https://yourInstance.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk

   </enterprise>
   <metadata>https://yourInstance.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk

   </metadata>
```

```
   <partner>https://yourInstance.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk

   </partner>
   <rest>https://yourInstance.salesforce.com/services/data/v{version}/
   </rest>
   <sobjects>https://yourInstance.salesforce.com/services/data/v{version}/sobjects/
   </sobjects>
   <search>https://yourInstance.salesforce.com/services/data/v{version}/search/
   </search>
   <query>https://yourInstance.salesforce.com/services/data/v{version}/query/
   </query>
   <profile>https://yourInstance.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":"https://yourInstance.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":"https://yourInstance.salesforce.com/profilephoto/005/F",
   "thumbnail":"https://yourInstance.salesforce.com/profilephoto/005/T"},
"urls":

{"enterprise":"https://yourInstance.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk",


"metadata":"https://yourInstance.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk",


"partner":"https://yourInstance.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk",

   "rest":"https://yourInstance.salesforce.com/services/data/v{version}/",
   "sobjects":"https://yourInstance.salesforce.com/services/data/v{version}/sobjects/",
   "search":"https://yourInstance.salesforce.com/services/data/v{version}/search/",
   "query":"https://yourInstance.salesforce.com/services/data/v{version}/query/",
   "profile":"https://yourInstance.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

In Windows, Mobile SDK looks for the list of login servers in the `Resources\servers.xml` file of your Shared project. The SalesforceSDK library uses this file to define production and sandbox servers.

You can add your servers to the runtime list by creating your own `Resources\servers.xml` file. 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://myloginserver.cloudforce.com"/>
</servers>
```

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 Salesforce admins to set various security policies and have explicit control over who can use the corresponding apps.

A developer or Salesforce admin 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 that 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 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 launches a Mobile SDK app.
3. User enters login information for Salesforce organization.
4. User enters PIN code for the Mobile SDK app.
5. User works in the app and then sends it to the background by opening another app (or receiving a call, and so on).
6. The app times out.
7. User re-opens the app, and the app PIN screen displays (for the Mobile SDK 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.

SEE ALSO:

Authenticating Apps with OAuth