



---

# Mobile SDK For Windows



© Copyright 2000–2015 salesforce.com, inc. All rights reserved. Salesforce is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

**PRERELEASE DOCUMENTATION**

# CONTENTS

<b>Chapter 1: Introduction to Salesforce Mobile Development</b>	<b>1</b>
Salesforce Development Basics	2
Developer Edition or Sandbox Environment?	2
Development Prerequisites	3
Development Prerequisites	3
Creating a Connected App	4
<b>GETTING STARTED</b>	<b>7</b>
<b>Chapter 2: Getting Started with Mobile SDK</b>	<b>7</b>
Prerequisites	8
Install Mobile SDK	8
Create a Mobile SDK Native Project for Windows 8.1	8
Add Mobile SDK to an Existing Windows 8.1 Project	9
<b>Chapter 3: Sample Apps In Mobile SDK For Windows 8.1</b>	<b>11</b>
<b>NATIVE</b>	<b>12</b>
<b>Chapter 4: Native Windows Development</b>	<b>12</b>
Mobile SDK Native Libraries	13
Developing a Native Windows App	13
About Login and Passcodes	14
Overview of Application Flow	14
Creating Pages with Authentication Support	16
About Salesforce REST APIs	16
<b>OFFLINE</b>	<b>24</b>
<b>Chapter 5: Offline Management</b>	<b>24</b>
Using SmartStore to Securely Store Offline Data	25
About SmartStore	25
Adding SmartStore to Native Windows Apps	26
Using Global SmartStore	27
Registering a Soup	27
Populating a Soup	28
Retrieving Data From a Soup	29
Smart SQL Queries	32
Manipulating Data	34
Managing Soups	34

- NativeSmartStoreSample App: Using SmartStore in Native Windows Apps . . . . . 37
- Using SmartSync to Access Salesforce Objects . . . . . 39
- Native . . . . . 39
- AUTHENTICATION AND SECURITY . . . . . 53**
- Chapter 6: Authentication, Security, and Identity in Mobile Apps . . . . . 53**
- OAuth Terminology . . . . . 54
- OAuth2 Authentication Flow . . . . . 54
  - OAuth 2.0 User-Agent Flow . . . . . 55
  - OAuth 2.0 Refresh Token Flow . . . . . 56
  - Scope Parameter Values . . . . . 56
  - Using Identity URLs . . . . . 58
  - Setting a Custom Login Server . . . . . 63
  - Revoking OAuth Tokens . . . . . 63
  - Refresh Token Revocation in Android Native Apps . . . . . 64
- Connected Apps . . . . . 64
  - About PIN Security . . . . . 65
- Portal Authentication Using OAuth 2.0 and Force.com Sites . . . . . 65
- INDEX . . . . . 67**

# CHAPTER 1 Introduction to Salesforce Mobile Development

In this chapter ...

- [Salesforce Development Basics](#)

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

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

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

# Salesforce Development Basics

---

## Developer Edition or Sandbox Environment?

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

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

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

## Types of Developer Environments

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

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

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

## Choosing an Environment

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

Developer Edition is ideal if:

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



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

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

Partner Developer Edition is ideal if:

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

Sandbox is ideal if:

- You are a salesforce.com customer with Enterprise, Unlimited, or Force.com Edition, which includes Sandbox.
- You are developing a Force.com application specifically for your production environment.

- You are not planning to build a Force.com application that will be distributed commercially.
- You have no intent to list on the AppExchange or distribute through Trialforce.

## Development Prerequisites

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

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

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

### 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.

## Development Prerequisites

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

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

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

The following requirements apply to specific platforms and technologies:

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

### Sign Up for Force.com

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

1. In your browser go to <https://developer.salesforce.com/signup>.
2. Fill in the fields about you and your company.
3. In the `Email Address` field, make sure to use a public address you can easily check from a Web browser.

4. Enter a unique `Username`. Note that this field is also in the *form* of an email address, but it does not have to be the same as your email address, and in fact, it's usually better if they aren't the same. Your username is your login and your identity on `developer.salesforce.com`, and so you're often better served by choosing a username that describes the work you're doing, such as `develop@workbook.org`, or that describes you, such as `firstname@lastname.com`.
5. Read and then select the checkbox for the `Master Subscription Agreement`.
6. Enter the Captcha words shown and click **Submit Registration**.
7. In a moment you'll receive an email with a login link. Click the link and change your password.

## Creating a Connected App

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

### IN THIS SECTION:

#### [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, navigate to **Create > Apps**.
3. Under Connected Apps, click **New**.
4. Perform steps for [Basic Information](#).
5. Perform steps for [API \(Enable OAuth Settings\)](#).
6. Click **Save**.

### Note:

- The `Callback URL` provided for OAuth doesn't have to be a valid URL; it only has to match what the app expects in this field. You can use any custom prefix, such as `sfdc://`.
- The detail page for your connected app displays a consumer key. It's a good idea to copy this key, as you'll need it later.
- After you create a new connected app, wait a few minutes for the token to propagate before running your app.

## Basic Information

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

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



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



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

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

## API (Enable OAuth Settings)

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

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

### Access and manage your Chatter feed (chatter\_api)

Allows access to Chatter REST API resources only.

**Access and manage your data (api)**

Allows access to the logged-in user's account using APIs, such as REST API and Bulk API. This value also includes `chatter_api`, which allows access to Chatter REST API resources.

**Access your basic information (id, profile, email, address, phone)**

Allows access to the Identity URL service.

**Access custom permissions (custom\_permissions)**

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

**Allow access to your unique identifier (openid)**

Allows access to the logged in user's unique identifier for OpenID Connect apps.

**Full access (full)**

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

**Perform requests on your behalf at any time (refresh\_token, offline\_access)**

Allows a refresh token to be returned if you are eligible to receive one. This lets the app interact with the user's data while the user is offline. The `refresh_token` scope is synonymous with `offline_access`.

**Provide access to custom applications (visualforce)**

Allows access to Visualforce pages.

**Provide access to your data via the Web (web)**

Allows the ability to use the `access_token` on the Web. This also includes `visualforce`, allowing access to Visualforce pages.

If your organization had the `No user approval required for users in this organization` option selected on your remote access prior to the Spring '12 release, users in the same organization as the one the app was created in still have automatic approval for the app. The read-only `No user approval required for users in this organization` checkbox is selected to show this condition. For connected apps, the recommended procedure after you've created an app is for administrators to install the app and then set `Permitted Users` to `Admin-approved users`. If the remote access option was not checked originally, the checkbox doesn't display.

# GETTING STARTED

## CHAPTER 2 Getting Started with Mobile SDK

### In this chapter ...

- [Prerequisites](#)
- [Install Mobile SDK](#)
- [Create a Mobile SDK Native Project for Windows 8.1](#)
- [Add Mobile SDK to an Existing Windows 8.1 Project](#)

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.

The next steps are:

1. [Make sure you have the required software.](#)
2. [Install Mobile SDK.](#)
3. Do any of the following:
  - [Create a new Mobile SDK native project for Windows 8.1.](#)
  - [Add Mobile SDK to an existing Windows 8.1 native project.](#)

## Prerequisites

---

Before you begin using Mobile SDK for Windows, make sure you've installed the following software.

- Microsoft Visual Studio 2013 with Update 3 or newer.
- Visual Studio SDK (<http://msdn.microsoft.com/en-us/library/bb166441.aspx>).
- SQLite for Visual Studio
  - Windows Phone 8.1: <http://www.sqlite.org/2014/sqlite-wp81-winrt-3080704.vsix>
  - Windows 8.1: <http://www.sqlite.org/2014/sqlite-winrt81-3080704.vsix>

## Install Mobile SDK

---

If you've installed the required software, installing Mobile SDK for Windows is just a matter of adding the Mobile SDK project template to Visual Studio.

1. Download the Visual Studio template for Mobile SDK:  
<https://github.com/forcedotcom/SalesforceMobileSDK-Windows/blob/unstable/template/SalesforceUniversalApplicationTemplateWin8.1.zip>.
2. Copy the template ZIP file to the `<user>\Documents\Visual Studio 2013\Templates\ProjectTemplates` directory.

When you click **File > New > Project...** in Visual Studio, the Mobile SDK template appears under Visual C# templates as **Salesforce Application (Universal Apps)**. When you create your project, Visual Studio retrieves all missing Mobile SDK modules from their NuGet repositories.



### Important:

- NuGet does not update SQLite packages. If the SQLite version changes in a future release, remember to update it manually.
- To update the template, close Visual Studio, then delete the existing template before installing the new version.


## Create a Mobile SDK Native Project for Windows 8.1

---

For quickest results, use the Mobile SDK template and NuGet packages to create your projects.

Before creating a project, be sure that you've installed the prerequisite software described in [Prerequisites](#).

1. In Visual Studio, click **File > New > Project**.
2. In the left pane, choose **Visual C#**.
3. In the right pane, select the **Salesforce Application (Universal Apps)** template.
4. If you don't want to create a solution file, deselect `Create directory for solution`.
5. Set Project Name and Solution Name, if applicable.
6. To use source control for your project, select `Add to source control`.
7. Click OK.
8. (Optional) For Connected App Settings, overwrite the default Client ID and Callback URL with the client ID and callback URL from your connected app.
9. (Optional) Select the OAuth scopes that your app requires. All apps require at least the `api` scope.
10. Click **Finish**.

 **Important:** If you use the default connected app values during development, be sure to change to your own connected app settings before you publish your app.


11. If you configured your project to use source control, select a source control provider.
12. Open the Solution Explorer. Right-click the solution and click **Enable NuGet Package Restore**. When asked if you want to configure this solution to download and restore missing NuGet packages during build, click **Yes**.
13. In the **Build** menu, click **Configuration Manager**.
14. For each active solution configuration, reset the Platform values for listed projects to the appropriate processor type. Do not leave any projects set to **Any CPU**.

Your new solution contains desktop and phone projects, as well as NuGet packages for Salesforce Mobile SDK and other dependencies.

## Add Mobile SDK to an Existing Windows 8.1 Project

To add Mobile SDK to an existing Windows 8.1 project, follow these steps.

1. In Visual Studio, open the existing Windows 8.1 project.
2. Open the Solution Explorer and right-click the solution.
3. Select **Enable NuGet Package Restore**, 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 the examples from the sample apps in GitHub.
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. For Phone applications, reference `Salesforce.SDK.Core` and `Salesforce.SDK.Phone`.
12. For Windows applications, reference `Salesforce.SDK.Core` and `Salesforce.SDK.Store`.
13. 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.
14. Clean and build all projects.

## CHAPTER 3 Sample Apps In Mobile SDK For Windows 8.1

Mobile SDK for Microsoft Windows 8.1 provides sample apps that demonstrate major Mobile SDK features.

### **RestExplorer**

Demonstrates the OAuth and REST API functions of Salesforce Mobile SDK. Mobile SDK provides Store and Phone versions of this app.

### **NativeSmartStoreSample**

Demonstrates registering, populating, querying, and manipulating data in a local SmartStore database.

### **SmartSyncExplorer**

Demonstrates the power of the native SmartSync library for offline productivity.

### **NoteSync**

Demonstrates how to use non-REST APIs to retrieve Salesforce Notes.

### **Salesforce1.Container**

Provides the SalesforceA app wrapped in a Mobile SDK container.

## CHAPTER 4 Native Windows Development

### In this chapter ...

- [Mobile SDK Native Libraries](#)
- [Developing a Native Windows App](#)

Salesforce Mobile SDK for native Windows 8.1 provides the tools you need to build full-featured, flexible Salesforce apps for Windows 8.1 mobile 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 securely 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 8.1 application development principles and frameworks. If you're a newbie, consult the [Microsoft Developer Network](#) to begin learning. See [Prerequisites](#) for additional requirements.



## Mobile SDK Native Libraries

---

Mobile SDK for Windows 8.1 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 such as Salesforce.SDK.Phone for phone development or Salesforce.SDK.Store for Windows tablet/desktop development.

### **Salesforce.SDK.Phone**

This library is necessary for writing Salesforce applications on Windows Phone. It has a dependency on Salesforce.SDK.Core, which has the bulk of the necessary code. For examples of how to create applications, refer to the public GitHub repository at <https://github.com/forcedotcom/SalesforceMobileSDK-Windows>.

### **Salesforce.SDK.Store**

This library is necessary for writing Salesforce applications on Windows. It has a dependency on Salesforce.SDK.Core, which has the bulk of the necessary code. For examples of how to create applications, refer to the public GitHub repository at <https://github.com/forcedotcom/SalesforceMobileSDK-Windows>.

### **Salesforce.SDK.SmartStore**

This library is necessary for using SmartStore with a Salesforce-powered application. You can find more documentation at [https://developer.salesforce.com/docs/atlas.en-us.mobile\\_sdk.meta/mobile\\_sdk/offline\\_native\\_sql\\_aggregator.htm](https://developer.salesforce.com/docs/atlas.en-us.mobile_sdk.meta/mobile_sdk/offline_native_sql_aggregator.htm), or by studying the sample code in the GitHub repository at <https://github.com/forcedotcom/SalesforceMobileSDK-Windows>.

This library requires SQLite. See [#install\\_prerequisites\\_\\_topic-title](#) on page 8

### **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 at <https://github.com/forcedotcom/SalesforceMobileSDK-Windows>.

Prerequisites:

- Install Sqlite for WinRT 8.1: <http://www.sqlite.org/2014/sqlite-winrt81-3080702.vsix>
- After installing Sqlite, add a NuGet reference for MSOpenTech SqlitePCL to your project.
- SmartSync requires Salesforce.SDK.SmartStore.

This library requires SQLite. See [#install\\_prerequisites\\_\\_topic-title](#) on page 8

## Developing a Native Windows 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 8.1 app. You're responsible for designing and developing your app's user interface.

IN THIS SECTION:

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


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

## 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 Salesforce Universal Template, 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.

### IN THIS SECTION:

[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](https://developer.salesforce.com/docs/) 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.

### IN THIS SECTION:

[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
<b>SOQL query</b> Executes the given SOQL string and returns the resulting data set	<code>GetRequestForQuery</code>
<b>SOSL search</b> Executes the given SOSL string and returns the resulting data set	<code>GetRequestForSearch</code>
<b>Metadata</b> Returns the object's metadata	<code>GetRequestForMetadata</code>
<b>Describe global</b> Returns a list of all available objects in your organization and their metadata	<code>GetRequestForDescribeGlobal</code>
<b>Describe object type</b> Returns a complete description of all metadata for a single object type	<code>GetRequestForDescribe</code>
<b>Retrieve</b> Retrieves a single record by object ID	<code>GetRequestForRetrieve</code>
<b>Update</b> Updates a record with the given data	<code>GetRequestForUpdate</code>
<b>Upsert</b> Updates or inserts a record from external data, if and only if the given external ID matches the value of the external ID field	<code>GetRequestForUpsert</code>
<b>Create</b> Creates a record of the specified object type	<code>GetRequestForCreate</code>

Operation	Static factory method
<b>Delete</b> Deletes the object of the given type with the given ID	<code>GetRequestForDelete</code>
<b>Versions</b> Returns summary information about each Salesforce version currently available	<code>GetRequestForVersions</code>
<b>Resources</b> Returns available resources for the specified API version, including resource name and URI	<code>GetRequestForResources</code>

## 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 Salesforce 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*}");
```

## IN THIS SECTION:

[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.

Table 1:

RestRequest member	Description
<code>response.Success</code>	Returns true if the call succeeded.
<code>response.StatusCode</code>	Returns the <code>HttpStatusCode</code> of the call.
<code>response.Error</code>	Contains an <code>Exception</code> 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
<code>response.AsString</code>	Returns the body of the response as a string.
<code>response.AsJArray</code>	Returns the body of the response as an instance of <code>JArray</code> .
<code>response.AsJObject</code>	Returns the body of the response as an instance of <code>JObject</code> .
<code>response.PrettyBody</code>	Returns the body of the response ( <code>JArray</code> or <code>JObject</code> ) with indented formatting.

## CHAPTER 5 Offline Management

### In this chapter ...

- [Using SmartStore to Securely Store Offline Data](#)
- [Using SmartSync to Access Salesforce Objects](#)

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

- SmartStore lets you store app data in encrypted databases, or *soups*, on the device. When the device goes back online, you can use SmartStore APIs to synchronize data changes with the Salesforce server.
- SmartSync is a data framework that provides a mechanism for easily fetching Salesforce data, modeling it as JavaScript objects, and caching it for offline use. When it's time to upload offline changes to the Salesforce server, SmartSync gives you highly granular control over the synchronization process. SmartSync is built on the popular `Backbone.js` open source library and uses SmartStore as its default cache.

## Using SmartStore to Securely Store Offline Data

---

Mobile devices can lose connection at any time, and environments such as hospitals and airplanes often prohibit connectivity. To handle these situations, it's important that your mobile apps continue to function when they go offline.

Mobile SDK provides SmartStore, a multi-threaded, secure solution for offline storage on mobile devices. With SmartStore, your users can continue working with data in a secure environment even when the device loses connectivity.

### About SmartStore

SmartStore stores data as JSON documents in a simple, single-table database. You can define indexes for this database, and you can query the data either with SmartStore helper methods that implement standard queries, or with custom queries using SmartStore's Smart SQL language.

SmartStore uses StoreCache, a Mobile SDK caching mechanism, to provide offline synchronization and conflict resolution services. We recommend that you use StoreCache to manage operations on Salesforce data.



**Note:** Pure HTML5 apps store offline information in a browser cache. Browser caching isn't part of the Mobile SDK, and we don't document it here. SmartStore uses storage functionality on the device. This strategy requires a native or hybrid development path.

### About the Sample Code

Code snippets in this chapter use two objects--Account and Opportunity--which come predefined with every Salesforce organization. Account and Opportunity have a master-detail relationship; an account can have more than one opportunity.

### 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

Type	Description
string	Text string, stored with database encoding (UTF-8)

## IN THIS SECTION:

[Date Representation](#)

## Date Representation

SmartStore does not specify a type for dates and times. We recommend that you represent these values with SmartStore data types as shown in the following table:

Type	Format As	Description
string	An ISO 8601 string	"YYYY-MM-DD HH:MM:SS.SSS"
floating	A Julian day number	The number of days since noon in Greenwich on November 24, 4714 BC according to the proleptic Gregorian calendar. This value can include partial days that are expressed as decimal fractions.
integer	Unix time	The number of seconds since 1970-01-01 00:00:00 UTC

## Adding SmartStore to Native Windows Apps

In Windows, you can add SmartStore 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.

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. Browse and select the `Salesforce.SDK.SmartStore.dll` library under `<path_to_your_clone>\SalesforceMobileSDK-Windows\SalesforceSDK\Salesforce.SDK.SmartStore\bin\ [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 is condensed from the `SmartStoreExtension` class of the `NativeSmartStoreSample` app.

```
using Salesforce.SDK.SmartStore.Store;
...
const string AccountsSoup = "Account";
IndexSpec[] AccountsIndexSpecs = new IndexSpec[]
{
```

```

        new IndexSpec("Name", SmartStoreType.SmartString),
        new IndexSpec("Id", SmartStoreType.SmartString),
        new IndexSpec("OwnerId", SmartStoreType.SmartString)
    };

    SmartStore _store = SmartStore.GetSmartStore();
    _store.RegisterSoup(AccountsSoup, AccountsIndexSpecs);

```

## 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 with an anonymous inline definition of the `AsyncRequestCallback` interface. The following code creates and sends a REST request for the SOQL query "SELECT Name, Id, OwnerId FROM Account". If the 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 queries are single-predicate searches. Only Smart SQL queries 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.");
}
```

## 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" with ID field "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("contacts", "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

In order to track soup entries for insert, update, and delete, SmartStore adds a few fields to each entry:

- `_soupEntryId`—This field is the primary key for the soup entry in the table for a given soup.
- `_soupLastModifiedDate`—The number of milliseconds since 1/1/1970.
  - To convert to a JavaScript date, use `new Date(entry._soupLastModifiedDate)`.
  - To convert a date to the corresponding number of milliseconds since 1/1/1970, use `date.getTime()`.

When inserting or updating soup entries, SmartStore automatically sets these fields. When removing or retrieving specific entries, you can reference them by `_soupEntryId`.

## Inserting or Updating Soup Entries

If the provided soup entries already have the `_soupEntryId` slots set, then entries identified by that slot are updated in the soup. If an entry does not have a `_soupEntryId` slot, or the value of the slot doesn't match any existing entry in the soup, then the entry is added (inserted) to the soup, and the `_soupEntryId` slot is overwritten.



**Note:** Do not manipulate the `_soupEntryId` or `_soupLastModifiedDate` value yourself.

Use one of the following `Upsert()` methods to insert or update entries:

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

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.

## Upserting with an External ID

If your soup entries mirror data from an external system, you might need to refer to those entities by their ID (primary key) in the external system. For that purpose, 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, a new soup entry will be created.
- If the external ID field is found, it will be updated.
- If more than one field matches the external ID, an error will be returned.

When creating a new entry locally, use a regular upsert. Set the external ID field to a value that you can later query when uploading the new entries to the server.

When updating entries with data coming from the server, use the upsert with external ID. Doing so guarantees that you don't end up with duplicate soup entries for the same remote record.

## 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");
```

### IN THIS SECTION:

#### [Get the Database Size](#)

To query the amount of disk space consumed by the database, call the applicable database size method.

#### [Clear a Soup](#)

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

#### [Retrieve a Soup's Index Specs](#)

If you want to examine or display the index specifications for a soup, call the applicable index specs retrieval method.

#### [Change Existing Index Specs On a Soup](#)

To change existing index specs, call the applicable soup alteration method.

#### [Reindex a Soup](#)

Use reindexing if you previously altered a soup without reindexing the data, but later you want to make sure all elements in the soup are properly indexed.

#### [Remove a Soup](#)

Removing a soup deletes it. When a user signs out, all soups are deleted automatically. For other occasions in which you want to delete a soup, call the applicable soup removal method.

## Get the Database Size

To query the amount of disk space consumed by the database, call the applicable database size method.

## Windows Native Apps

In Windows native apps, call:

```
public int getDatabaseSize ()
```

## Clear a Soup

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

## Android Apps

In Android 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)
```

## Change Existing Index Specs On a Soup

To change existing index specs, call the applicable soup alteration method.

Keep these important points in mind when reindexing data:

- The `reIndexData` argument is optional, because re-indexing can be expensive. When `reIndexData` is set to false, expect your throughput to be faster by an order of magnitude.
- Altering a soup that already contains data can degrade your app's performance. Setting `reIndexData` to true worsens the performance hit.
- As a performance guideline, expect the `alterSoup()` operation to take one second per thousand records when `reIndexData` is set to true. Individual performance varies according to device capabilities, the size of the elements, and the number of indexes.
- Be aware that other SmartStore operations must wait for the soup alteration to complete.
- If the operation is interrupted—for example, if the user exits the application—the operation automatically resumes when the application re-opens the SmartStore database.

### Windows Native Apps

In an Windows native app, call:

```
public void alterSoup(String soupName, IndexSpec [] indexSpecs, boolean reIndexData) throws  
JSONException
```

## 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 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)
```

## NativeSmartStoreSample App: Using SmartStore in Native Windows Apps

The NativeSmartStoreSample app demonstrates how to use SmartStore in a native Windows app. It also demonstrates the ability to make complex SQL-like queries, including aggregate functions, such as SUM and COUNT, and joins across different soups within SmartStore.

### Creating a Soup

The first step to storing a Salesforce object in SmartStore is to create a soup for the object. The function call to register a soup takes two arguments—the name of the soup, and the index specs for the soup. Indexing supports three types of data: string, integer, and floating decimal. The following example illustrates how to initialize a soup for the Account object with indexing on Name, Id, and OwnerId fields.

**Windows:**

```
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. On Windows, `account` is of type `JObject`.

**Windows:**

```
public void InsertAccount(JObject account)
{
```

```

        if (account != null)
        {
            _store.Upsert(AccountsSoup, account);
        }
    }
}

```

## Running Queries Against SmartStore

Beginning with Mobile SDK 2.0, you can run advanced SQL-like queries against SmartStore that span multiple soups. The syntax of a SmartStore query is similar to standard SQL syntax, with a couple of minor variations. A colon (":") serves as the delimiter between a soup name and an index field. A set of curly braces encloses each `<soup-name>:<field-name>` pair. See [Smart SQL Queries](#).

Here's an example of an aggregate query run against SmartStore:

```

SELECT {Account:Name},
       COUNT({Opportunity:Name}),
       SUM({Opportunity:Amount}),
       AVG({Opportunity:Amount}), {Account:Id}, {Opportunity:AccountId}
FROM {Account}, {Opportunity}
WHERE {Account:Id} = {Opportunity:AccountId}
GROUP BY {Account:Name}

```

This query represents an implicit join between two soups, Account and Opportunity. It returns:

- Name of the Account
- Number of opportunities associated with an Account
- Sum of all the amounts associated with each Opportunity of that Account
- Average amount associated with an Opportunity of that Account
- Grouping by Account name

The code snippet below demonstrates how to run such queries from within a native app. In this example, `smartSql` is the query and `pageSize` is the requested page size. The `pageIndex` argument specifies which page of results you want returned.

### Windows:

```

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;
}

```

## Using SmartSync to Access Salesforce Objects

The SmartSync library is a collection of APIs that make it easy for developers to sync data between Salesforce databases and their mobile apps. It provides the means for getting and posting data to a server endpoint, caching data on a device, and reading cached data. For sync operations, SmartSync predefines cache policies for fine-tuning interactions between cached data and server data in offline and online scenarios. A set of SmartSync convenience methods automate common network activities, such as fetching sObject metadata, fetching a list of most recently used objects, and building SOQL and SOSL queries.

### Native

#### Using SmartSync in Native Apps

The native SmartSync library provides native APIs that simplify the development of offline-ready apps.

SmartSync libraries offer parallel architecture and functionality across Mobile SDK platforms, expressed in each platform's native language. The shared functional concepts are straightforward:

- Query Salesforce object metadata by calling Salesforce REST APIs.
- Store the retrieved object data locally and securely for offline use.
- Sync data changes when the device goes from an offline to an online state.

With SmartSync native libraries, you can:

- Get and post data by interacting with a server endpoint. SmartSync helper APIs encode the most commonly used endpoints. These APIs help you fetch sObject metadata, retrieve the list of most recently used (MRU) objects, and build SOQL and SOSL queries. You can also use arbitrary endpoints that you specify in a custom class.
- Fetch Salesforce records and metadata and cache them on the device, using one of the pre-defined cache policies.
- Edit records offline and save them offline in SmartStore.
- Synchronize batches of records by pushing locally modified data to the Salesforce cloud.

#### SmartSync Components

The following components form the basis of SmartSync architecture.

##### Sync Manager

- **Windows class:** `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:

<ul style="list-style-type: none"> <li>• <b>Windows:</b> <code>Salesforce.SDK.SmartSync.Model.SyncState.SyncStatusTypes</code> enum</li> </ul>	<p>Tracks the state of a sync operation. States include:</p> <ul style="list-style-type: none"> <li>• New—The sync operation has been initiated but has not yet entered a transaction with the server.</li> <li>• Running—The sync operation is negotiating a sync transaction with the server.</li> <li>• Done—The sync operation finished successfully.</li> </ul>
--	--

	<ul style="list-style-type: none"> <li>Failed—The sync operation finished unsuccessfully.</li> </ul>
<ul style="list-style-type: none"> <li><b>Windows:</b> <code>Salesforce.SDK.SmartSync.Model.SyncTarget</code></li> </ul>	Base class for custom sync targets.
<ul style="list-style-type: none"> <li><b>Windows:</b> <code>Salesforce.SDK.SmartSync.Model.SyncOptions</code></li> </ul>	Specifies configuration options for “sync up” and “sync down” operations. Options include the list of field names to be synced.

## Metadata Manager

- Windows class:** `Salesforce.SDK.SmartSync.Manager.MetadataManager`

Performs data loading functions. This class helps you handle more full-featured queries and configurations than the sync manager protocols support. For example, metadata manager APIs can:

- Load SmartScope object types.
- Load MRU lists of sObjects. Results can be either global or limited to a specific sObject.
- Load the complete object definition of an sObject, using the `describe` API.
- Load the list of all sObjects available in an organization.
- Determine if an sObject is searchable, and, if so, load the search layout for the sObject type.
- Load the color resource for an sObject type.
- Mark an sObject as viewed on the server, thus moving it to the top of the MRU list for its sObject type.

To interact with the server, `MetadataManager` uses the standard Mobile SDK REST API classes:

- Windows:** `RestClient`, `RestRequest`

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

## Cache Manager

- Windows class:** `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

- Windows class:** `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

- Windows class:** `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

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:

- `DoesCacheExist()` to verify the current existence of the cache
- `GetLastCacheUpdateTime()` to check the cache's current update status.

Here's a list of cache policies.

**Table 2: Cache Policies**

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

## Object Representation

When you use the metadata manager, SmartSync model information arrives as a result of calling metadata manager load methods. The metadata manager loads the data from the current user's organization and presents it in one of three classes:

- Object
- Object Type
- Object Type Layout

### Object

- **Windows class:** `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

- **Windows class** `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

- **Windows class** `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

In Windows, 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. Browse and select the `Salesforce.SDK.SmartStore.dll` library under  
`<path_to_your_clone>\SalesforceMobileSDK-Windows\SalesforceSDK\Salesforce.SDK.SmartStore\bin\ [Release | Debug]`.
5. Browse and select the `Salesforce.SDK.SmartSync.dll` library under  
`<path_to_your_clone>\SalesforceMobileSDK-Windows\SalesforceSDK\Salesforce.SDK.SmartSync\bin\ [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\Salesforce.SDK.SmartStore\Salesforce.SDK.SmartStore.csproj`
  - `<path_to_your_clone>\SalesforceMobileSDK-Windows\SalesforceSDK\Salesforce.SDK.SmartStore\Salesforce.SDK.SmartSync.csproj`
3. In Solution Explorer, RIGHT-CLICK **References** on one of your projects, then click **Add Reference...**
4. Click **Solution > Projects**.
5. Select **Salesforce.SDK.SmartStore** and **Salesforce.SDK.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](#).

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.

### IN THIS SECTION:

[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:

```
public static SyncManager getInstance(UserAccount account);
```

### 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:

- **Windows 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:
  - **Windows:** `SyncState.MergeModeOptions.Overwrite`
2. Preserve all local changes and locally modified records. Set the `options` parameter to the following value:
  - **Windows:** `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: Windows:

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 then needs only a trivial implementation, as carried out in the `HandleSyncUpdate()` method:

```
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 deletel = delete;
        await Task.Delay(10).ContinueWith(async a => await RemoveContact(deletel));
    }
}

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)

```

## Custom Targets

### Using Custom Sync Down Targets

During sync down operations, a sync down target controls the set of records to be downloaded and the request endpoint. You can use pre-formatted MRU, SOQL-based, and SOSL-based targets, or you can create custom sync down targets. Custom targets can access arbitrary REST endpoints both inside and outside of Salesforce.

#### IN THIS SECTION:

[Defining a Custom Sync Down Target](#)

[Invoking the Sync Down Method with a Custom Target](#)

[Sample Apps](#)

### 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:

- **Windows:** `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.

##### Windows:

```
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.

##### Windows:

```
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`.

##### Windows:

```
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.

##### Windows:

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

### Invoking the Sync Down Method with a Custom Target

To use your custom target:

**Windows:**

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);
```

**Sample Apps**

The `NoteSync` native sample app defines and uses the `NoteSync.Data.ContentSqlSyncDownTarget` sync down target.

**Using Custom Sync Up Targets**

During sync up operations, a sync up target controls the set of records to be uploaded and the REST endpoint for updating records on the server. You can access arbitrary REST endpoints—both inside and outside of Salesforce—by creating custom sync up targets.

**IN THIS SECTION:**

[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.

**Windows:**

```
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.

**Windows:**

```
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.

**Windows:**

```
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.

##### Windows:

```
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.

##### Windows:

```
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.

##### Windows:

```
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.

##### Windows:

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

### Invoking the Sync Up Method with a Custom Target

To use your custom sync up target:

#### Android:

On a `SyncManager` instance, call:

```
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 Classes

- **Windows:** `Salesforce.SDK.SmartSync.Manager.CacheManager`

## Read and Write Methods

Here are the `CacheManager` methods for reading and writing `sObject` metadata, MRU lists, and `sObject` layouts.

- **Windows:**

### `sObject` 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 methods:

- **Windows:**

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

These methods let you clear a selected portion of the cache. To clear the entire cache, call the method for each cache key and data type you've stored.



# AUTHENTICATION AND SECURITY

## CHAPTER 6 Authentication, Security, and Identity in Mobile Apps

### In this chapter ...

- [OAuth Terminology](#)
- [OAuth2 Authentication Flow](#)
- [Connected Apps](#)
- [Portal Authentication Using OAuth 2.0 and Force.com Sites](#)

Secure authentication is essential for enterprise applications running on mobile devices. OAuth2 is the industry-standard protocol that allows secure authentication for access to a user's data, without handing out the username and password. It is often described as the valet key of software access: a valet key only allows access to certain features of your car: you cannot open the trunk or glove compartment using a valet key.

Mobile app developers can quickly and easily embed the Salesforce OAuth2 implementation. The implementation uses an HTML view to collect the username and password, which are then sent to the server. A session token is returned and securely stored on the device for future interactions.

A Salesforce *connected app* is the primary means by which a mobile device connects to Salesforce. A connected app gives both the developer and the administrator control over how the app connects and who has access. For example, a connected app can be restricted to certain users, can set or relax an IP range, and so forth.

## OAuth Terminology

---

### Access Token

A value used by the consumer to gain access to protected resources on behalf of the user, instead of using the user's Salesforce credentials. The access token is a session ID, and can be used directly.

### Authorization Code

A short-lived token that represents the access granted by the end user. The authorization code is used to obtain an access token and a refresh token.

### Connected App

An application external to Salesforce that uses the OAuth protocol to verify both the Salesforce user and the external application. Replaces remote access application.

### Consumer Key

A value used by the consumer to identify itself to Salesforce. Referred to as `client_id`.

### Refresh Token

A token used by the consumer to obtain a new access token, without having the end user approve the access again.

### Remote Access Application (DEPRECATED)

A *remote access application* is an application external to Salesforce that uses the OAuth protocol to verify both the Salesforce user and the external application. A remote access application is implemented as a "connected app" in the Salesforce Help. Remote access applications have been deprecated in favor of connected apps.

## OAuth2 Authentication Flow

---

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

### First Time Authentication Flow

1. User opens a mobile application.
2. An authentication dialog/window/overlay appears.
3. User enters username and password.
4. App receives session ID.
5. User grants access to the app.
6. App starts.

### Ongoing Authentication

1. User opens a mobile application.
2. If the session ID is active, the app starts immediately. If the session ID is stale, the app uses the refresh token from its initial authorization to get an updated session ID.
3. App starts.

### PIN Authentication (Optional)

1. User opens a mobile application after not using it for some time.

2. If the elapsed time exceeds the configured PIN timeout value, a passcode entry screen appears. User enters the PIN.



**Note:** PIN protection is a function of the mobile policy and is used only when it's enabled in the Salesforce connected app definition. It can be shown whether you are online or offline, if enough time has elapsed since you last used the application. See [About PIN Security](#).

3. App re-uses existing session ID.
4. App starts.

## OAuth 2.0 User-Agent Flow

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

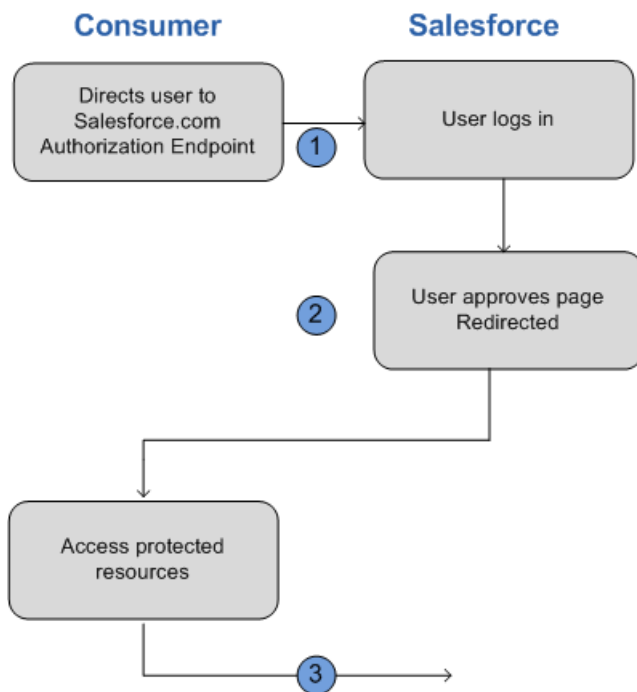
In the user-agent flow, the client application receives the access token in the form of an HTTP redirection. The client application requests the authorization server to redirect the user-agent to another web server or local resource accessible to the user-agent, which is capable of extracting the access token from the response and passing it to the client application. Note that the token response is provided as a hash (#) fragment on the URL. This is for security, and prevents the token from being passed to the server, as well as to other servers in referral headers.

This user-agent authentication flow doesn't utilize the client secret since the client executables reside on the end-user's computer or device, which makes the client secret accessible and exploitable.



**Warning:** Because the access token is encoded into the redirection URI, it might be exposed to the end-user and other applications residing on the computer or device.

If you are authenticating using JavaScript, call `window.location.replace()` ; to remove the callback from the browser's history.



1. The client application directs the user to Salesforce to authenticate and authorize the application.

2. The user must always approve access for this authentication flow. After approving access, the application receives the callback from Salesforce.


After obtaining an access token, the consumer can use the access token to access data on the end-user's behalf and receive a refresh token. Refresh tokens let the consumer get a new access token if the access token becomes invalid for any reason.

## OAuth 2.0 Refresh Token Flow

After the consumer has been authorized for access, they can use a refresh token to get a new access token (session ID). This is only done after the consumer already has received a refresh token using either the Web server or user-agent flow. It is up to the consumer to determine when an access token is no longer valid, and when to apply for a new one. Bearer flows can only be used after the consumer has received a refresh token.

The following are the steps for the refresh token authentication flow. More detail about each step follows:

1. The consumer uses the existing refresh token to request a new access token.
2. After the request is verified, Salesforce sends a response to the client.

 **Note:** Mobile SDK apps can use the SmartStore feature to store data locally for offline use. SmartStore data is inherently volatile. Its lifespan is tied to the authenticated user as well as to OAuth token states. When the user logs out of the app, SmartStore deletes all soup data associated with that user. Similarly, when the OAuth refresh token is revoked or expires, the user's app state is reset, and all data in SmartStore is purged. Carefully consider the volatility of SmartStore data when designing your app. This warning is especially important if your org sets a short lifetime for the refresh token.

## Scope Parameter Values

OAuth requires scope configuration both on server and on client. The agreement between the two sides defines the scope contract.

- **Server side**—Define scope permissions in a connected app on the Salesforce server. These settings determine which scopes client apps, such as Mobile SDK apps, can request. At a minimum, configure your connected app OAuth settings to match what's specified in your code. For hybrid apps and iOS native apps, `refresh_token`, `web`, and `api` are usually sufficient. For Android native apps, `refresh_token` and `api` are usually sufficient.
- **Client side**—Define scope requests in your Mobile SDK app. Client scope requests must be a subset of the connected app's scope permissions.

## Server Side Configuration

The `scope` parameter enables you to fine-tune what the client application can access in a Salesforce organization. The valid values for `scope` are:

Value	Description
<code>api</code>	Allows access to the current, logged-in user's account using APIs, such as REST API and Bulk API. This value also includes <code>chatter_api</code> , which allows access to Chatter REST API resources.
<code>chatter_api</code>	Allows access to Chatter REST API resources only.
<code>custom_permissions</code>	Allows access to the custom permissions in an organization associated with the connected app, and shows whether the current user has each permission enabled.

Value	Description
<code>full</code>	Allows access to all data accessible by the logged-in user, and encompasses all other scopes. <code>full</code> does not return a refresh token. You must explicitly request the <code>refresh_token</code> scope to get a refresh token.
<code>id</code>	Allows access to the identity URL service. You can request <code>profile</code> , <code>email</code> , <code>address</code> , or <code>phone</code> , individually to get the same result as using <code>id</code> ; they are all synonymous.
<code>openid</code>	Allows access to the current, logged in user's unique identifier for OpenID Connect apps.  The <code>openid</code> 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 <a href="#">OpenID Connect specifications</a> in addition to the access token.
<code>refresh_token</code>	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 <code>offline_access</code> .
<code>visualforce</code>	Allows access to Visualforce pages.
<code>web</code>	Allows the ability to use the <code>access_token</code> on the Web. This also includes <code>visualforce</code> , 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
<code>refresh_token</code>	Implicitly requested by Mobile SDK for your app; no need to include in your request.
<code>api</code>	Include in your request if you're making any Salesforce REST API calls (applies to most apps).
<code>web</code>	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.)
<code>full</code>	Include if you wish to request all permissions. (Mobile SDK implicitly requests <code>refresh_token</code> for you.)
<code>chatter_api</code>	Include in your request if your app calls Chatter REST APIs.
<code>id</code>	(Not needed)
<code>visualforce</code>	Use <code>web</code> instead.

## Using Identity URLs

In addition to the access token, an identity URL is also returned as part of a token response, in the `id` scope parameter.

The identity URL is both a string that uniquely identifies a user, as well as a RESTful API that can be used to query (with a valid access token) for additional information about the user. Salesforce returns basic personalization information about the user, as well as important endpoints that the client can talk to, such as photos for the user, and API endpoints it can access.

The format of the URL is: `https://login.salesforce.com/id/orgID/userID`, where `orgID` is the ID of the Salesforce organization that the user belongs to, and `userID` is the Salesforce user ID.



**Note:** For a sandbox, `login.salesforce.com` is replaced with `test.salesforce.com`.

The URL must always be HTTPS.

## Identity URL Parameters

The following parameters can be used with the access token and identity URL. The access token can be used in an authorization request header or in a request with the `oauth_token` parameter.

Parameter	Description
Access token	See “Using the Access Token” in the Salesforce Help.
Format	<p>This parameter is optional. Specify the format of the returned output. Valid values are:</p> <ul style="list-style-type: none"> <li>• <code>json</code></li> <li>• <code>xml</code></li> </ul> <p>Instead of using the <code>format</code> parameter, the client can also specify the returned format in an accept-request header using one of the following:</p> <ul style="list-style-type: none"> <li>• <code>Accept: application/json</code></li> <li>• <code>Accept: application/xml</code></li> <li>• <code>Accept: application/x-www-form-urlencoded</code></li> </ul> <p>Note the following:</p> <ul style="list-style-type: none"> <li>• Wildcard accept headers are allowed. <code>*/*</code> is accepted and returns JSON.</li> <li>• A list of values is also accepted and is checked left-to-right. For example: <code>application/xml,application/json,application/html,*/*</code> returns XML.</li> <li>• The <code>format</code> parameter takes precedence over the accept request header.</li> </ul>
Version	<p>This parameter is optional. Specify a SOAP API version number, or the literal string, <code>latest</code>. If this value isn’t specified, the returned API URLs contains the literal value <code>{version}</code>, in place of the version number, for the client to do string replacement. If the value is specified as <code>latest</code>, the most recent API version is used.</p>
PrettyPrint	<p>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: <code>X-PrettyPrint:1</code>. If this value isn’t specified, the returned XML or JSON is optimized for size rather than readability.</p>

Parameter	Description
Callback	This parameter is optional. Specify a valid JavaScript function name. This parameter is only used when the format is specified as JSON. The output is wrapped in this function name (JSONP.) For example, if a request to <code>https://server/id/orgid/userid/</code> returns <code>{"foo": "bar"}</code> , a request to <code>https://server/id/orgid/userid/?callback=baz</code> returns <code>baz({"foo": "bar"})</code> .

## Identity URL Response

A valid request returns the following information in JSON format.

- `id`—The identity URL (the same URL that was queried)
- `asserted_user`—A boolean value, indicating whether the specified access token used was issued for this identity
- `user_id`—The Salesforce user ID
- `username`—The Salesforce username
- `organization_id`—The Salesforce organization ID
- `nick_name`—The community nickname of the queried user
- `display_name`—The display name (full name) of the queried user
- `email`—The email address of the queried user
- `email_verified`—Indicates whether the organization has email verification enabled (`true`), or not (`false`).
- `first_name`—The first name of the user
- `last_name`—The last name of the user
- `timezone`—The time zone in the user's settings
- `photos`—A map of URLs to the user's profile pictures



**Note:** Accessing these URLs requires passing an access token. See “Using the Access Token” in the Salesforce Help.

- `picture`
  - `thumbnail`
- `addr_street`—The street specified in the address of the user's settings
- `addr_city`—The city specified in the address of the user's settings
- `addr_state`—The state specified in the address of the user's settings
- `addr_country`—The country specified in the address of the user's settings
- `addr_zip`—The zip or postal code specified in the address of the user's settings
- `mobile_phone`—The mobile phone number in the user's settings
- `mobile_phone_verified`—The user confirmed this is a valid mobile phone number. See the Mobile User field description.
- `status`—The user's current Chatter status
  - `created_date:xsdt` `datetime` value of the creation date of the last post by the user, for example, 2010-05-08T05:17:51.000Z
  - `body`: the body of the post
- `urls`—A map containing various API endpoints that can be used with the specified user

 **Note:** Accessing the REST endpoints requires passing an access token. See “Using the Access Token” in the Salesforce Help.

- enterprise (SOAP)
- metadata (SOAP)
- partner (SOAP)
- rest (REST)
- subjects (REST)
- search (REST)
- query (REST)
- recent (REST)
- profile
- feeds (Chatter)
- feed-items (Chatter)
- groups (Chatter)
- users (Chatter)
- custom\_domain—This value is omitted if the organization doesn’t have a custom domain configured and propagated
- active—A boolean specifying whether the queried user is active
- user\_type—The type of the queried user
- language—The queried user’s language
- locale—The queried user’s locale
- utcOffset—The offset from UTC of the timezone of the queried user, in milliseconds
- last\_modified\_date—xsd datetime format of last modification of the user, for example, 2010-06-28T20:54:09.000Z
- is\_app\_installed—The value is `true` when the connected app is installed in the org of the current user and the access token for the user was created using an OAuth flow. If the connected app is not installed, the property does not exist (instead of being `false`). When parsing the response, check both for the existence and value of this property.
- mobile\_policy—Specific values for managing mobile connected apps. These values are only available when the connected app is installed in the organization of the current user and the app has a defined session timeout value and a PIN (Personal Identification Number) length value.
  - screen\_lock—The length of time to wait to lock the screen after inactivity
  - pin\_length—The length of the identification number required to gain access to the mobile app
- push\_service\_type—This response value is set to `apple` if the connected app is registered with Apple Push Notification Service (APNS) for iOS push notifications or `androidGcm` if it’s registered with Google Cloud Messaging (GCM) for Android push notifications. The response value type is an array.
- custom\_permissions—When a request includes the `custom_permissions` scope parameter, the response includes a map containing custom permissions in an organization associated with the connected app. If the connected app is not installed in the organization, or has no associated custom permissions, the response does not contain a `custom_permissions` map. The following shows an example request.

```
http://login.salesforce.com/services/oauth2/authorize?response_type=token&client_id=3MVG9lKcPoNINVBKv6EgVJiF.snSDwh6_2wSS7BrOhHGEJkC_&redirect_uri=http://www.example.org/qa/security/oauth/useragent_flow_callback.jsp&scope=api%20id%20custom_permissions
```



The following shows the JSON block in the identity URL response.

```
"custom_permissions":
{
  "Email.View":true,
  "Email.Create":false,
  "Email.Delete":false
}
```

The following is a response in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<user xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<id>http://na1.salesforce.com/id/00Dx0000001T0zk/005x0000001S2b9</id>
<asserted_user>true</asserted_user>
<user_id>005x0000001S2b9</user_id>
<organization_id>00Dx0000001T0zk</organization_id>
<nick_name>admin1.2777578168398293E12foofoofoofoo</nick_name>
<display_name>Alan Van</display_name>
<email>admin@2060747062579699.com</email>
<status>
  <created_date xsi:nil="true"/>
  <body xsi:nil="true"/>
</status>
<photos>
  <picture>http://na1.salesforce.com/profilephoto/005/F</picture>
  <thumbnail>http://na1.salesforce.com/profilephoto/005/T</thumbnail>
</photos>
<urls>
  <enterprise>http://na1.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk
  </enterprise>
  <metadata>http://na1.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk
  </metadata>
  <partner>http://na1.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk
  </partner>
  <rest>http://na1.salesforce.com/services/data/v{version}/
  </rest>
  <subjects>http://na1.salesforce.com/services/data/v{version}/subjects/
  </subjects>
  <search>http://na1.salesforce.com/services/data/v{version}/search/
  </search>
  <query>http://na1.salesforce.com/services/data/v{version}/query/
  </query>
  <profile>http://na1.salesforce.com/005x0000001S2b9
  </profile>
</urls>
<active>true</active>
<user_type>STANDARD</user_type>
<language>en_US</language>
<locale>en_US</locale>
<utcOffset>-28800000</utcOffset>
<last_modified_date>2010-06-28T20:54:09.000Z</last_modified_date>
</user>
```

The following is a response in JSON format:

```
{
  "id": "http://na1.salesforce.com/id/00Dx0000001T0zk/005x0000001S2b9",
  "asserted_user": true,
  "user_id": "005x0000001S2b9",
  "organization_id": "00Dx0000001T0zk",
  "nick_name": "admin1.2777578168398293E12foofoofoofoo",
  "display_name": "Alan Van",
  "email": "admin@2060747062579699.com",
  "status": { "created_date": null, "body": null },
  "photos": { "picture": "http://na1.salesforce.com/profilephoto/005/F",
    "thumbnail": "http://na1.salesforce.com/profilephoto/005/T" },
  "urls": {
    "enterprise": "http://na1.salesforce.com/services/Soap/c/{version}/00Dx0000001T0zk",
    "metadata": "http://na1.salesforce.com/services/Soap/m/{version}/00Dx0000001T0zk",
    "partner": "http://na1.salesforce.com/services/Soap/u/{version}/00Dx0000001T0zk",
    "rest": "http://na1.salesforce.com/services/data/v{version}/",
    "subjects": "http://na1.salesforce.com/services/data/v{version}/subjects/",
    "search": "http://na1.salesforce.com/services/data/v{version}/search/",
    "query": "http://na1.salesforce.com/services/data/v{version}/query/",
    "profile": "http://na1.salesforce.com/005x0000001S2b9"
  },
  "active": true,
  "user_type": "STANDARD",
  "language": "en_US",
  "locale": "en_US",
  "utcOffset": -28800000,
  "last_modified_date": "2010-06-28T20:54:09.000+0000"
}
```

After making an invalid request, the following are possible responses from Salesforce:

Error Code	Request Problem
403 (forbidden) — HTTPS_Required	HTTP
403 (forbidden) — Missing_OAuth_Token	Missing access token
403 (forbidden) — Bad_OAuth_Token	Invalid access token
403 (forbidden) — Wrong_Org	Users in a different organization
404 (not found) — Bad_Id	Invalid or bad user or organization ID
404 (not found) — Inactive	Deactivated user or inactive organization
404 (not found) — No_Access	User lacks proper access to organization or information
404 (not found) — No_Site_Endpoint	Request to an invalid endpoint of a site
404 (not found) — Internal Error	No response from server
406 (not acceptable) — Invalid_Version	Invalid version
406 (not acceptable) — Invalid_Callback	Invalid callback

## Setting a Custom Login Server

For special cases—for example, if you're a Salesforce partner using Trialforce—you might need to redirect your customer login requests to a non-standard login URI. For iOS apps, you set the Custom Host in your app's iOS settings bundle. If you've configured this setting, it will be used as the default connection.

## Android Configuration

In Android, login hosts are known as server connections. Prior to Mobile SDK v. 1.4, server connections for Android apps were hard-coded in the SalesforceSDK project. In v. 1.4 and later, the host list is defined in the `res/xml/servers.xml` file. The SalesforceSDK library project uses this file to define production and sandbox servers.

You can add your servers to the runtime list by creating your own `res/xml/servers.xml` file in your application project. The root XML element for this file is `<servers>`. This root can contain any number of `<server>` entries. Each `<server>` entry requires two attributes: `name` (an arbitrary human-friendly label) and `url` (the web address of the login server.)

Here's an example of a `servers.xml` file.

```
<?xml version="1.0" encoding="utf-8"?>
<servers>
  <server name="XYZ.com Login" url="https://<username>.cloudforce.com"/>
</servers>
```

## Server Whitelisting Errors

If you get a whitelist rejection error, you'll need to add your custom login domain to the `ExternalHosts` list for your project. This list is defined in the `<project_name>/<platform_path>/config.xml` file. Add those domains (e.g. `cloudforce.com`) to the app's whitelist in the following files:

For Mobile SDK 2.0:

- **iOS:** `/Supporting Files/config.xml`
- **Android:** `/res/xml/config.xml`

## Revoking OAuth Tokens

When a user logs out of an app, or the app times out or in other ways becomes invalid, the logged-in users' credentials are cleared from the mobile app. This effectively ends the connection to the server. Also, Mobile SDK revokes the refresh token from the server as part of logout.

## Revoking Tokens

To revoke OAuth 2.0 tokens, use the revocation endpoint:

```
https://login.salesforce.com/services/oauth2/revoke
```

Construct a POST request that includes the following parameters using the `application/x-www-form-urlencoded` format in the HTTP request entity-body. For example:

```
POST /revoke HTTP/1.1
Host: https://login.salesforce.com/services/oauth2/revoke
Content-Type: application/x-www-form-urlencoded
```

```
token=currenttoken
```

If an access token is included, we invalidate it and revoke the token. If a refresh token is included, we revoke it as well as any associated access tokens.

The authorization server indicates successful processing of the request by returning an HTTP status code 200. For all error conditions, a status code 400 is used along with one of the following error responses.

- `unsupported_token_type`—token type not supported
- `invalid_token`—the token was invalid

For a sandbox, use `test.salesforce.com` instead of `login.salesforce.com`.

## Refresh Token Revocation in Android Native Apps

When a refresh token is revoked by an administrator, the default behavior is to automatically log out the current user. As a result of this behavior:

- Any subsequent REST API calls your app makes will fail.
- The system discards your user's account information and cached offline data.
- The system forces the user to navigate away from your page.
- The user must log into Salesforce again to continue using your app.

These side effects provide a secure response to the administrator's action.

## Token Revocation Events

When a token revocation event occurs, the `ClientManager` object sends an Android-style notification. The intent action for this notification is declared in the `ClientManager.ACCESS_TOKEN_REVOKE_INTENT` constant.

`SalesforceActivity.java`, `SalesforceListActivity.java`, `SalesforceExpandableListActivity.java`, and `SalesforceDroidGapActivity.java` implement `ACCESS_TOKEN_REVOKE_INTENT` event listeners. These listeners automatically take logged out users to the login page when the refresh token is revoked. A toast message notifies the user of this occurrence.

## Connected Apps

A connected app integrates an application with Salesforce using APIs. Connected apps use standard SAML and OAuth protocols to authenticate, provide Single Sign-On, and provide tokens for use with Salesforce APIs. In addition to standard OAuth capabilities, connected apps allow administrators to set various security policies and have explicit control over who may use the corresponding applications.

A developer or administrator defines a connected app for Salesforce by providing the following information.

- Name, description, logo, and contact information
- A URL where Salesforce can locate the app for authorization or identification
- The authorization protocol: OAuth, SAML, or both
- Optional IP ranges where the connected app might be running
- Optional information about mobile policies the connected app can enforce

Salesforce Mobile SDK apps use connected apps to access Salesforce OAuth services and to call Salesforce REST APIs.

## About PIN Security

Salesforce Connected Apps have an additional layer of security via PIN protection on the app. This PIN protection is for the mobile app itself, and isn't the same as the PIN protection on the device or the login security provided by the Salesforce organization.

In order to use PIN protection, the developer must select the **Implements Screen Locking & Pin Protection** checkbox when creating the Connected App. Mobile app administrators then have the options of enforcing PIN protection, customizing timeout duration, and setting PIN length.



**Note:** Because PIN security is implemented in the mobile device's operating system, only native and hybrid mobile apps can use PIN protection; HTML5 Web apps can't use PIN protection.

In practice, PIN protection can be used so that the mobile app locks up if it's isn't used for a specified number of minutes. When a mobile app is sent to the background, the clock continues to tick.

To illustrate how PIN protection works:

1. User turns on phone and enters PIN for the device.
2. User starts mobile app (Connected App).
3. User enters login information for Salesforce organization.
4. User enters PIN code for mobile app.
5. User works in the app, then sends it to the background by opening another app (or receiving a call, and so on).
6. The mobile app times out.
7. User re-opens the app, and the app PIN screen displays (for the mobile app, not the device).
8. User enters app PIN and can resume working.

## Portal Authentication Using OAuth 2.0 and Force.com Sites

The Salesforce Spring '13 Release adds enhanced flexibility for portal authentication. If your app runs in a Salesforce portal, you can use OAuth 2.0 with a Force.com site to obtain API access tokens on behalf of portal users. In this configuration you can:

- Authenticate portal users via Auth providers and SAML, rather than a SOAP API `login()` call.
- Avoid handling user credentials in your app.
- Customize the login screen provided by the Force.com site.

Here's how to get started.

1. Associate a Force.com site with your portal. The site generates a unique URL for your portal. See [Associating a Portal with Force.com Sites](#).
2. Create a custom login page on the Force.com site. See [Managing Force.com Site Login and Registration Settings](#).
3. Use the unique URL that the site generates as the redirect domain for your users' login requests.

The OAuth 2.0 service recognizes your custom host name and redirects the user to your site login page if the user is not yet authenticated.



**Example:** For example, rather than redirecting to `https://login.salesforce.com`:

```
https://login.salesforce.com/services/oauth2/authorize?response_type=
code&client_id=<your_client_id>&redirect_uri=<your_redirect_uri>
```

redirect to your unique Force.com site URL, such as `https://mysite.secure.force.com`:

```
https://mysite.secure.force.com/services/oauth2/authorize?response_type=
code&client_id=<your_client_id>&redirect_uri=<your_redirect_uri>
```

For more information and a demonstration video, see [OAuth for Portal Users](#) on the Force.com Developer Relations Blogs page.

# INDEX

## A

- authentication
  - Force.com Sites 65
  - and portal authentication 65
  - portal 65
  - portal authentication 65
- Authentication 53
- Authentication flow 54
- Authorization 65

## C

- cache policies for SmartSync 41
- CachePolicy class 41
- caching data 24
- Callback URL 4
- connected app, creating 4
- Connected apps 53, 64
- Consumer key 4

## D

- data types
  - date representation 26
  - SmartStore 25–26
- Delete soups 27, 29, 34
- Developer Edition
  - vs. sandbox 2
- Developer.force.com 3
- Development 3

## E

- endpoints, REST requests 43–44, 46, 48
- Events
  - Refresh token revocation 64

## F

- Flow 54–56

## G

- Getting Started 7
- GlobalClientManager object 19
- Glossary 54

## I

- Identity URLs 58

- installing
  - on Windows 8
- iOS native app, developing 12
- IP ranges 64

## L

- login and passcodes 14

## M

- Mobile development 1
- Mobile policies 64

## N

- native
  - application flow 14
  - SalesforceApplication class 14
  - SalesforceConfig class 15
- native apps
  - developing 13
- Native apps
  - Android 64
- native projects, creating, Windows 8.1 8
- NativeMainPage class 16

## O

- OAuth
  - custom login host 63
- OAuth2 53–54
- offline management 24
- Offline storage 25–26

## P

- Page with authentication support 16
- Parameters, scope 56
- PIN protection 65
- Prerequisites 3

## Q

- Queries, Smart SQL 32
- Querying a soup 27, 29, 34
- querySpec 27, 29, 34

## R

- Refresh token
  - Revocation 64
- Refresh token flow 56

- Refresh token revocation [64](#)
- Refresh token revocation events [64](#)
- registerSoup [27](#), [29](#), [34](#)
- Remote access [54](#)
- Remote access application [4](#)
- REST APIs
  - native [16](#)
  - native classes [16](#)
- REST request endpoints [43–44](#), [46](#), [48](#)
- REST requests
  - manual requests [21](#)
  - sending [22](#)
- REST responses
  - processing [22](#)
- RestClient class
  - GetRestClient() method [19](#)
  - GetUnauthenticatedRestClient() method [19](#)
  - obtaining an instance [19](#)
  - unauthenticated, obtaining an instance [19](#)
- RestRequest class
  - convenience methods [18](#)
  - manual requests [21](#)
  - obtaining an instance [19](#)
  - static factory methods [19](#)
- RestResponse class [19](#), [22](#)
- Restricting user access [64](#)
- Revoking tokens [63](#)

## S

- SalesforceSDKManager.shouldLogoutWhenTokenRevoked()
  - method [64](#)
- sample apps
  - Windows [11](#)
- sandbox.org [2](#)
- Scope parameters [56](#)
- SDK prerequisites [3](#)
- Security [53](#)
- SFSmartSyncSyncManager [43](#)
- shouldLogoutWhenTokenRevoked() method [64](#)
- Sign up [3](#)
- Smart SQL [25](#), [32](#)
- SmartStore
  - about [25](#)
  - adding to existing Android apps [26](#)
  - alterSoup() function [34](#), [36](#)
  - clearSoup() function [34–35](#)
  - data types [25](#)
  - date representation [26](#)
  - getDatabaseSize() function [34–35](#)
- SmartStore (*continued*)
  - getSoupIndexSpecs() function [34](#)
  - global SmartStore [27](#)
  - managing soups [34–36](#)
  - populate soups [28](#)
  - reindexSoup() function [34](#)
  - reIndexSoup() function [36](#)
  - removeSoup() function [34](#), [36](#)
  - soups [25](#)
- SmartStore functions [27](#), [29](#), [34](#)
- SmartSync
  - CacheManager [39](#)
  - CachePolicy class [41](#)
  - custom sync down target, samples [50](#)
  - custom sync down targets [49](#)
  - custom sync down targets, defining [49](#)
  - custom sync down targets, invoking [49](#)
  - custom sync up targets [50](#)
  - custom sync up targets, invoking [51](#)
  - custom sync up targets, defining [50](#)
  - incremental sync [46](#)
  - Metadata API [39](#)
  - MetadataManager [39](#)
  - native apps, creating [42](#)
  - NetworkManager [39](#)
  - object representation [41](#)
  - resync [46](#)
  - reSync:updateBlock: iOS method [46](#)
  - reSync() Windows method [46](#)
  - Salesforce endpoints [43–44](#), [46](#), [48](#)
  - search layouts [39](#)
  - sending requests [43–44](#), [46](#), [48](#)
  - SmartSyncSDKManager [39](#)
  - SObject types [39](#)
  - SOQLBuilder [39](#)
  - SOSLBuilder [39](#)
  - storing and retrieving cached data [51](#)
  - sync manager, using [43](#)
  - tutorial [37](#)
  - using in native apps [39](#)
- SmartSync Data Framework [24](#)
- soups
  - populate [28](#)
- Soups [27](#), [29](#), [34](#)
- soups, managing [34–36](#)
- StoreCache [25](#)
- sync manager
  - using [43](#)
- SyncManager [43](#)



### T

- Terminology [54](#)
- Tokens, revoking [63](#)
- tutorial
  - SmartSync [37](#)

### U

- upsertSoupEntries [27](#), [29](#), [34](#)
- URLs, indentity [58](#)
- User-agent flow [55](#)

### W

- Windows
  - add Mobile SDK to existing Windows 8.1 project [9](#)
  - install Mobile SDK [8](#)
  - native apps [13–14](#)
  - prerequisites [8](#)
  - SalesforceApplication class [14](#)
  - SalesforceConfig class [15](#)
  - sample apps [11](#)
- Windows 8.1
  - creating native projects [8](#)