Kapsel Development

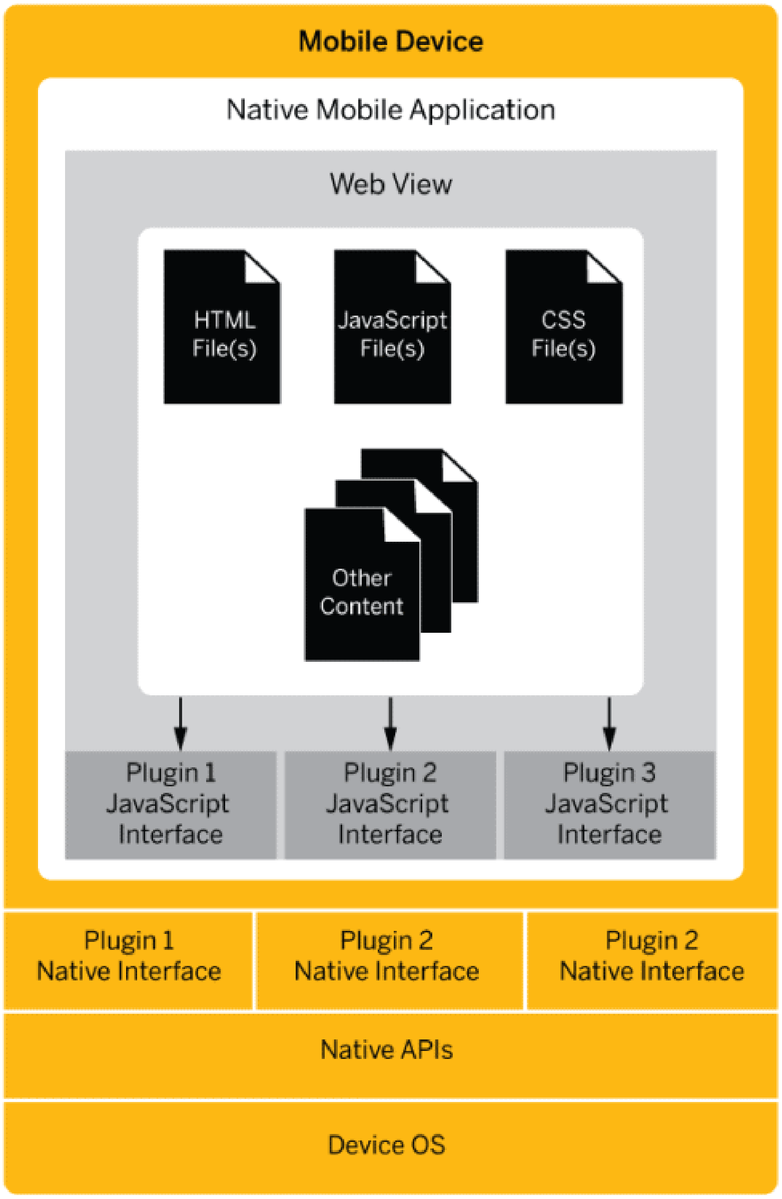
SAP Mobile Platform 3.0

Contents

Kapsel Development

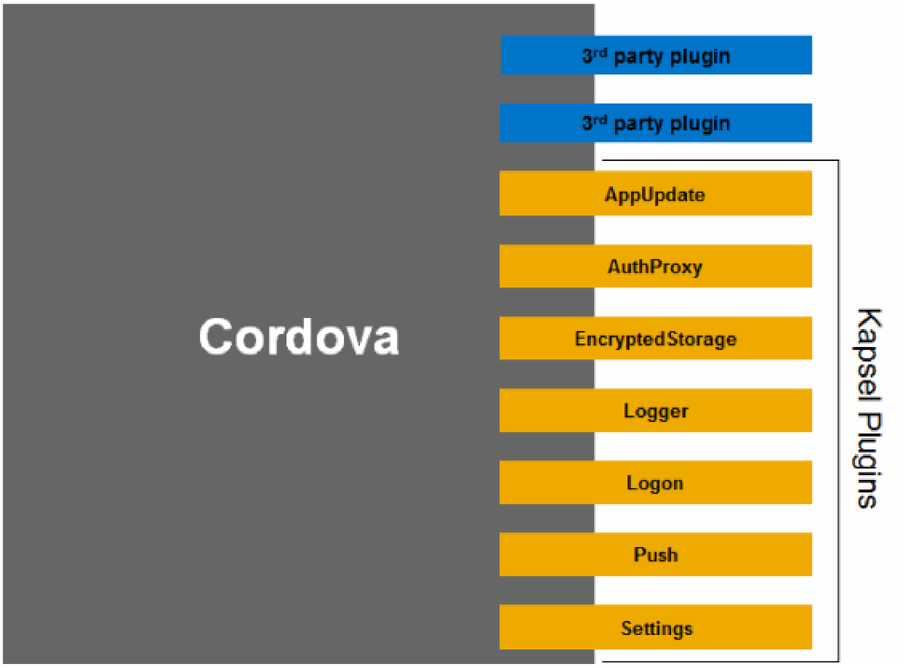
Kapsel is a set of SAP® plugins for Apache Cordova.

Apache Cordova provides a suite of APIs you can use to access native capabilities. The Cordova container provides JavaScript libraries that give you consistent APIs you can call the same way on any supported device. Beginning with Apache Cordova 3.0, the Cordova container is simply a holder in which any APIs and extensions are implemented as plugins. Apache Cordova includes a command line interface for managing Cordova applications and the application development process.


Mobile Device
Native Mobile Application
Web View
HTML File(s)
JavaScript File(s)
CSS File(s)
Other Content
Plugin 1 JavaScript Interface
Plugin 2 JavaScript Interface
Plugin 3 JavaScript Interface
Plugin 1 Native Interface
Plugin 2 Native Interface
Plugin 2 Native Interface
Native APIs
Device OS

Kapsel leverages the Cordova application container and provides SAP plugins to make the Cordova container enterprise-grade, allowing it to more seamlessly integrate with SAP Mobile Platform Server. The Kapsel plugins provide capabilities like application life cycle management, implementation of a common logon manager and single sign-on (SSO), integration with SAP Mobile Platform Server-based push notifications and so on. Since

Kapsel is implemented without modifying the Cordova container, it is compatible with anything else you develop with Cordova.



Developing Kapsel Applications

Once your application is developed, create a Cordova project and install the Kapsel plugins.

Setting Up the Development Environment

To build Kapsel applications, you must first set up your development environment, which includes installing both SAP Mobile Platform Server, and the SAP Mobile Platform SDK.

Prerequisites

- Verify that you can access SAP Mobile Platform Server from your machine
- If you are using Windows, download and extract Apache Ant and add it to the system variable path, `PATH=%PATH%;C:\apache-ant-<version>\bin`. See *http://ant.apache.org*.

See *http://service.sap.com/pam* to verify that you are using the supported versions for the Kapsel development environment.

Android Requirements

Android tools run on Windows, Linux, and OS X. To build Kapsel apps for Android, you need:

- Java Development Kit (JDK)
- Android SDK

See the Apache Cordova documentation at *http://cordova.apache.org/docs/en/3.0.0/guide_platforms_android_index.md.html#Android%20Platform%20Guide* for more information about getting started with Android.

Installing the Java SDK

See *http://www.oracle.com/technetwork/java/javase/downloads/index.html*.

After installing the Java SDK, define the JAVA_HOME environment variable.

Download the Plugins

Set up the Android Development Environment by downloading the required plugins.

Prerequisites:

- Download the Java Standard Edition Development Kit from *http://www.oracle.com/technetwork/java/javase/downloads/index.html*
- Download the ADT-supported version of Eclipse from *http://www.eclipse.org/downloads/*

1. Start the Eclipse environment.
2. From the **Help** menu, select **Install New Software**.
3. Click **Add**.
4. In the Add Repository dialog, enter a name for the new plugin.
5. Enter one of the following for URL:
   - `https://dl-ssl.google.com/android/eclipse/`
   - `http://dl-ssl.google.com/android/eclipse/`
6. Click **OK**.
7. Select **Developer Tools** and click **Next**.
8. Review the tools to be downloaded.
9. Click **Next**.
10. Read and accept the license agreement and click **Finish**.
11. Once the installation is complete, restart Eclipse.

Installing the ADT Plugin

Follow the instructions for installing the ADT Plugin for Eclipse at *http://developer.android.com/sdk/installing/installing-adt.html*.

If you prefer to work in an IDE other than Eclipse, you do not need to install Eclipse or ADT. You can simply use the Android SDK tools to build and debug your application.

Installing the Google USB Driver

The Google USB Driver for Windows is as an optional SDK component you need only if you are developing on Windows and want to connect a Google Android-powered device (such as a Nexus 7) to your development environment over USB.

Download the Google USB driver package from *http://developer.android.com/sdk/win-usb.html*.

Installing the Android SDK

Install the Android SDK for plugin use with your IDE.

1. Confirm that your system meets the requirements at *http://developer.android.com/sdk/requirements.html*.
2. Download and install the supported version of the Android SDK starter package.
3. Add the Android SDK to your PATH environment variable:

   On Windows, add `<Android SDK Location>\tools` to the PATH environment variable

   On OS X, the command is: `export PATH=$PATH:<path to Android SDK>/tools`
4. Launch the Android SDK Manager and install the Android tools (SDK Tools and SDK Platform-tools) and the Android API.
5. Launch the **Android Virtual Device Manager**, and create an Android virtual device to use as your emulator.

   Note: (For offline applications only) Due to limitation on the emulator, you cannot determine the network connection state. For more information on other limitations, see **Emulator Limitations** in *http://developer.android.com/tools/devices/emulator.html#limitations* at the Android Developer Web site.

iOS Requirements

To build Kapsel apps for iOS, you need:

- Mac OS X
- Xcode and Xcode command line tools
- For testing on iOS devices (not the simulator), you need:
  - An Apple Developer account
  - iOS development certificate
  - Provisioning files for each device you are testing with

See the Apache Cordova documentation at *http://cordova.apache.org/docs/en/3.0.0/guide_platforms_ios_index.md.html#iOS%20Platform%20Guide* for more information about getting started with iOS.

Downloading the Xcode IDE

Download and install Xcode from the Apple Developers Web site.

1. Go to *http://developer.apple.com/downloads/*.

   Note: You must be a paying member of the iOS Developer Program. Free members cannot download the supported version.
2. Log in using your Apple Developer credentials.
3. (Optional) To narrow the search scope, unselect all Categories except Developer Tools.
4. Download the appropriate Xcode and SDK combination.

Installing Git

See *http://git-scm.com/book/en/Getting-Started-Installing-Git*.

Note: If you are using a proxy server you must configure git.

```
git config --global http.proxy http://proxy:8080
 git config --global https.proxy http://proxy:8080
```

Installing Node.js

Use Node.js v0.10.11 and later, and its package manager, npm, to install Apache Cordova. See *http://nodejs.org/*. You can see the version installed by using the node command: **node –v**.

You must add the `Node.js` folder to your system PATH.

Note: If you are using a proxy server you must configure npm. At the command prompt, enter:

```
npm config set proxy  http://proxy_host:port
npm config set https-proxy http://proxy_host:port
```

Installing the Apache Cordova Command Line Interface

See *http://cordova.apache.org/docs/en/3.0.0/guide_cli_index.md.html#The%20Command-line%20Interface*. Follow all of the steps in the Cordova command line interface `readme.md`.

1. Open a command prompt window, and enter:
   On Windows: `npm install -g cordova@<latest_supported_version>`
   On Mac: `sudo npm install -g cordova@<latest_supported_version>`
   For example, to install the Cordova command line interface version 3.0.9, enter:
   `npm install -g cordova@3.0.9`
   -g indicates that Apache Cordova should be installed globally.

   Note: If you are installing on Mac and you see a warning message that you are installing globally into a root-only directory, run this command to change the owner of the command line interface installation folder:

```
sudo chown -R user_name /usr/local/lib/node_modules/cordova
```

You can copy the command text from the error message and paste it in at the command prompt at the bottom of the terminal window.

2. On Mac, when prompted, enter your root user password.
3. Verify the Cordova installation by entering this command at the command prompt, or in the terminal window:**cordova –v**

   The output shows the Cordova version installed, for example, `3.0.9`.

   You should also scroll back through the entire installation history shown in the terminal and look for errors to verify the installation was successful.

Installing ios-sim

To allow the Cordova command line to start the iOS simulator on Mac, you must install ios-sim.

1. Download the ios-sim tool files from *https://github.com/phonegap/ios-sim*.
2. Open a terminal window, and enter: `sudo npm install -g ios-sim`
3. When prompted, enter your root user password.
4. Verify the ios-sim installation by entering this command in the terminal window: `ios-sim --version`

   The output shows the ios-sim version installed, for example, 1.8.2.

Configuring the Application in the Management Cockpit

Configure the application settings in the Management Cockpit. These settings enable you to monitor and manage your applications.

Prerequisites

- Make sure SAP Mobile Platform Server is installed.
- Make sure the server is started.
- Launch the Management Cockpit.

Task

Defining Applications

Create a new native, hybrid, or Agentry application definition. The definition enables you to manage the application using Management Cockpit.

1. In Management Cockpit, select **Applications**, and click **New**.
2. In the New Application window, enter:

Field	Value
ID	Unique identifier for the application in reverse domain notation. This is the application or bundle identifier that is defined or generated during application development. Reverse domain notation is the practice of reversing a registered domain name; for example, the reverse domain notation for an object in `sap.com` might be `MyApp.sap.com`). The application identifier: • Must be unique. • Must start with an alphabetic character. • Can contain only alphanumeric characters, underscores (_), and periods (.). • Must not include spaces. • Can be up to 64 characters long. Formatting guidelines: • SAP recommends that application IDs contain a minimum of two dots ("."). For example, this ID is valid: `com.sap.mobile.app1`. • Application IDs should not start with a dot ("."). For example, this ID is invalid: `.com.sap.mobile.app1`. • Application IDs should not include two consecutive dots ("."). For example, this ID is invalid: `com..sap.mobile.app1`.
Name	Application name. The name: • Can contain only alphanumeric characters, spaces, underscores (_), and periods (.). • Can be up to 80 characters long.
Vendor	(Optional) Vendor who developed the application. The vendor name: • Can contain only alphanumeric characters, spaces, underscores (_), and periods (.). • Can be up to 255 characters long.
Type	Application type. • Native – native iOS and Android applications. • Hybrid – container-based applications, such as Kapsel. • Agentry – metadata-driven application. **Note:** You can configure only one Agentry application per SAP Mobile Platform Server. Once configured, Agentry no longer appears as an option.

Field	Value
Description	(Optional) Short description of the application. The description: • Can contain alphanumeric characters. • Can contain most special characters, except for percent signs (%) and ampersands (&). • Can be up to 255 characters long.

3. Click **Save**. Application-related tabs appear, such as Back End, Authentication, Push, and so forth. You are ready to configure the application, based on the application type.

   Note: These tabs appear in Management Cockpit only after you define or select an application. The tabs used differ by application type.

Defining Back-end Connections for Native and Hybrid Apps

Define back-end connections for the selected native or hybrid application.

1. From Management Cockpit, select **Applications** > **Back End**, and enter values for the selected application.

Field	Value
Connection Name **Note:** Appears only when adding a connection under Back-End Connections.	Identifies the back-end connection by name. The connection name: • Must be unique. • Must start with an alphabetic character. • Can contain only alphanumeric characters, underscores (_), and periods (.). • Must not include spaces.

Endpoint	The back-end connection URL, or service document URL the application uses to access business data on the back-end system or service. The service document URL is the document destination you assigned to the service in Gateway Management Cockpit. Typical format:
	`http://`*`host`*`:`*`port`*`/gateway/odata/`*`namespace`*`/`*`Connection_or_ServiceName`*`...`
	Examples:
	`http://testapp:65908/help/abc/app1/opg/sdata/TESTFLIGHT`
	`http://srvc3333.xyz.com:30003/sap/opu/odata/RMTSAMPLE`

Field	Value
Use System Proxy	(Optional) Whether to use system proxy settings in the SAP Mobile Platform `props.ini` file to access the back-end system. This setting is typically disabled, because most back-end systems can be accessed on the intranet without a proxy. The setting should only be enabled in unusual cases, where proxy settings are needed to access a remote back-end system outside of the network. When enabled, this particular connection is routed via the settings in `props.ini` file.
Rewrite URL	(Optional) Whether to mask the back-end URL with the equivalent SAP Mobile Platform Server URL. This is necessary to ensure the client makes all requests via SAP Mobile Platform Server and directly to the back end. Rewriting the URL also ensures that client applications need not do any additional steps to make requests to the back end via SAP Mobile Platform Server. If enabled, the back-end URL is rewritten with the SAP Mobile Platform Server URL. By default, the property is enabled.
Allow anonymous connections	(Optional) Whether to enable anonymous access. This means the application user can access the application without entering a user name and password. However, the back-end system still requires log on credentials to access the data, whether it is a read-only user, or a back-end user with specific roles. If enabled, enter the log on credential values used to access the back-end system: • **User name** – supply the user name for the back-end system. • **Password** – supply the password for the back-end system. If disabled, you do not need to provide these credentials. By default, the property is disabled. **Note:** If you use Allow Anonymous Connections for a native OData application, do not assign the No Authentication Challenge security profile to the application, or the anonymous OData requests will not be sent (Status code: 401 is reported).
Maximum Connections	The number of back-end connections that are available for connection pooling for this application. The larger the pool, the larger the number of possible parallel connections to this specific connection. The default is 500 connections. Factors to consider when resetting this property: • The expected number of concurrent users of the application. • The load acceptable to the back-end system. • The load that the underlying hardware and network can handle. **Note:** The maximum connections can be increased only if SAP Mobile Platform Server hardware can support the additional parallel connections, and if the underlying hardware and network infrastructure can handle it.

Field	Value
Certificate Alias	The name under which the administrator has imported the certificate key-pair in the `smp_keystore` file. The alias must be set when the back-end URL is `https://`, and the back-end server requires mutual authentication. There are conditions when https is used but the server does not require a client certificate. This certificate alias is required when the back end requires mutual SSL connectivity. Use the alias of a certificate stored in the SAP Mobile Platform Server keystore. SAP recommends that the CN value of the generated certificate be the fully qualified domain name of SAP Mobile Platform Server.

2. (Optional) Under Back-end Connections, view additional connections, or add new connections.

a) Click **New**, to add additional back-end connections in the server.
b) Enter values for the new back-end connection, using the values shown above.
c) Click **Save**. The new connection is added to the list.

Administrator maintains a list of server-level back-end connections (it includes all the connections in the SAP Mobile Platform Server) and application specific back-end connections. Application specific back-end connections are the connections enabled for an application. Users registered to an application can access only these back-end connections. Request-response to a back end connection that is not enabled for an application is not allowed (throws 403, "Forbidden" error).

By default, these additional back-end connections are enabled for an application.

Back-end connection is displayed in the list.

3. Select the **Application-specific Connections** from the drop-down to show the back-end connections that are enabled for a particular application.

You can view the **Server-level Connections** and enable the connections for this application using the checkbox.

Note: Multiple back ends can be authenticated using various options of authenticating requests available in security configuration.

Defining Application Authentication

Assign a security profile to the selected application. The security profile defines parameters for how the server authenticates the user during onboarding, and request-response interactions.

Prerequisites

Note: You must configure security profiles for application authentication before you can complete this step.

Task

Security profiles are comprised of one or more authentication providers. These authentication providers can be shared across multiple security profiles, and can be modified in Management Cockpit. You can stack multiple providers to take advantage of features in the order you chose; the Control Flag must be set for each enabled security provider in the stack.

1. From Management Cockpit, select **Applications** > **Authentication**.
2. Click **Existing Profile**.

   Note: You can also create a new profile.
3. In Name, select a security profile name from the list. The name appears under Security Profile Properties, and one or more security providers appear under Authentication Providers.
4. Under Security Profile Properties, enter values.

Field	Value
Name	A unique name for the application authentication profile.
Check Impersonation	(Optional) In token-based authentication, whether to allow authentication to succeed when the user name presented cannot be matched against any of the user names validated in the login modules. To prevent the user authentication from succeeding in this scenario, the property is enabled by default.
5. Under Authentication Providers, you can select a security profile URL to view its settings. To change its settings, you must modify it through **Settings** > **Security Profiles**.

Kapsel Security Matrix

Use one of the supported security configurations to secure your applications.

Security Configuration	Implemented Using	Security Provider
Basic authentication with HTTP	Kapsel Logon plugin	No Authentication Challenge
Basic authentication with HTTPS	Kapsel Logon plugin	No Authentication Challenge
Mutual authentication with HTTPS using a certificate	Kapsel Logon plugin, Client Hub, Afaria	X.509 User Certificate
SiteMinder (non-network edge)	Kapsel Logon plugin	HTTP/HTTPS Authentication
SiteMinder network edge (reverse proxy)	Kapsel Logon plugin	Populate JAAS Subject From ClientHTTP/HTTPS Authentication

Security Configuration	Implemented Using	Security Provider
SSO2 token (HTTP and HTTPS)	Kapsel Logon plugin, Kapsel AuthProxy plugin	HTTP/HTTPS Authentication
SSO passcode with Client Hub	Kapsel Logon plugin, Client Hub	System Login (Admin Only)
User name and password using Client Hub	Kapsel Logon plugin, Client Hub	System Login (Admin Only)
Basic authentication with LDAP back end	Kapsel Logon plugin	Directory Service (LDAP/ AD)
Encrypted storage	Kapsel EncryptedStorage plugin	Any
Data Vault	Kapsel Logon plugin	Any

Creating an Apache Cordova Project

To create projects for use with Kapsel, use the Cordova command line tool.

Prerequisites

Set up your development environment.

Task

You must run the commands from a Windows command prompt, or a terminal window on iOS. See *http://cordova.apache.org/docs/en/3.0.0/guide_cli_index.md.html#The%20Command-line%20Interface*.

1. Create a folder to hold your Kapsel Cordova projects.

   For example, on Windows, `C:\Documents and Settings\<your_account>\Kapsel_Projects`, or on OS X, `~/Documents/Kapsel_Projects`.
2. Open a Windows command prompt or terminal and navigate into the project folder you created.
3. At the command prompt, enter:

   On Windows: `cordova -d create <Project_Folder> <Application_ID> <Application Name>`

   On Mac: `cordova -d create ~<Project_Folder> <Application_ID> <Application Name>`

   The -d flag indicates debug output and is optional.

This may take a few minutes to complete, as an initial download of the template project that is used is downloaded to `C:\Users\user\.cordova` on Windows, or `~/users/user/.cordova` on Mac.

The parameters are:

- (Required) <*Project_Folder*> – the directory to generate for the project.
- (Optional) <*Application_ID*> – must match the Application ID as configured on SAP Mobile Platform Server for the application, which is reverse-domain style, for example, com.sap.kapsel.

  Note: <Application_ID> cannot be too simple. For example, you can have "a.b" for an ID, but you cannot have "MyApplicationId." The ID is used as the package name (name space) for the application and it must be at least two pieces separated by a period, otherwise, you will get build errors.
- (Optional) <*Application_Name*> – name for the application.

In this example, you create a project folder named `LogonDemo` in the `Kapsel_Projects` directory. The Application ID is "com.mycompany.logon" and the application name is "LogonDemo." Running **cordova -d** allows you to see the progress of the project creation.

```
cordova -d create ~\Kapsel_Projects\LogonDemo
com.mycompany.logon LogonDemo
```

Your new project includes scripts to build, emulate, and deploy your application.

Note: All of the Cordova command line interface commands operate against the current folder. The **create** command creates a folder structure for your Cordova projects while the remaining commands must be issued from within the project folder created by create.

4. To add the platform, change to the folder you created in the previous step:

```
cd <~Project_Name>
```

This OS X example adds the Android and iOS platforms, creating both an Xcode project and an Android project.

```
cd ~\Kapsel_Projects\LogonDemo
cordova platform add ios android
```

Note: Android is supported on both Windows and OS X, but iOS is supported only on OS X.

Note: You must add the platform before you add any Kapsel plugins.

The project directory structure is similar to this:

```
LogonDemo/
|--.cordova/
|-- merges/
| |-- android/
| `-- ios/
|-- platforms/
| |-- android/
| `-- ios/
|-- plugins/
|-- www/
   -- config.xml
`
```

- `.cordova` – identifies the project as a Cordova project. The command line interface uses this folder for storing its lazy loaded files. The folder is located immediately under your user's home folder (On Windows, `c:\users\user_name\` , and on Macintosh, `/users/user_name/.cordova`).
- `merges` – contains your Web application assets, such as HTML, CSS, and JavaScript files within platform-specific subfolders. Files in this folder override matching files in the `www/` folder for each respective platform.
- `www` – this folder contains the main HTML, CSS, and JavaScript assets for your application. The `config.xml` file contains meta data and native application information needed to generate the application. The `index.html` file is the default page of the application. Once you finish editing your project's files, update the platform specific files using the **cordova -d -prepare** command.
- `platforms` – native application project structures are contained in subfolders for the platforms you added to your application.

5. (Optional) You can test your Cordova project by opening it in the respective development environment, for example, Xcode or Eclipse with the ADT plugins, and running it on the simulator or emulator.
6. Add the plugins. For example, to add the Cordova console plugin and the Kapsel Logon plugin on Windows, enter:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/
cordova-plugin-console.git
 cordova -d plugin add C:\SAP\MobileSDK3\KapselSDK\plugins\logon
```

Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).

7. Edit the Web application content in the project's `www` folder and use the **cordova prepare** command to copy that content into the Android and iOS project folders:

```
cordova -d prepare android
cordova -d prepare ios
```

Project Settings

To set application configuration parameters, use the Cordova platform-independent `config.xml` file.

To modify application metadata, edit the `config.xml` file. The `config.xml` file is located in the `www` directory in your project. For information about the project settings for each platform, see *http://cordova.apache.org/docs/en/3.0.0/config_ref_index.md.html#Configuration%20Reference*.

Kapsel Plugins

Developers use one or more Kapsel plugins in Cordova applications to add SAP Mobile Platform awareness and capabilities to the application. The plugins that you use vary depending on your application's requirements. As they are standard Cordova plugins, manage Kapsel plugins in a Cordova project using the standard Cordova CLI plugin commands.

Kapsel Plugin	Use
AppUpdate	(Required) As the Kapsel lifecycle management plugin, AppUpdate manages application update downloads and installs updates to the Kapsel application. The AppUpdate plugin initiates the check for an update when the application starts, and when it resumes after being suspended. You can also start an app update manually, if required. AppUpdate requires the Logon plugin; the two plugins are installed together.
Logon	Manages user onboarding and the authentication process for SAP Mobile Platform applications. Most other Kapsel plugins use capabilities that this plugin exposes. The plugin interfaces with the SAP Afaria® client as well as the Client Hub application to help manage authentication and single sign-on. You can install this plugin standalone, or it is automatically installed with AppUpdate.
AuthProxy	Provides capabilities that are used in certain security scenarios such as mutual authentication and in SiteMinder environments.
Logger	Lets you have an application write entries to a local log, which can be uploaded to the SAP Mobile Platform Server for analysis. The SAP Mobile Platform administrator can manage setting the application log remotely from the server and upload device logs to the server without user intervention.
Push	Manages the process of registering for push requests as well as exposes events that help you code an application to respond to push notifications. Once the push registration is completed, the plugin uses the Settings plugin to exchange application settings information with SAP Mobile Platform Server so it knows how to manage delivery of push notifications to the application.

Kapsel Plugin	Use
EncryptedStorage	Adds an encrypted persistent store (key/value pair) to a Cordova application, which allows you to build an application that securely stores application data while offline, or while the application is not running. Unlike the built-in local storage, EncryptedStorage is nonblocking.
Settings	Required if you are using the Push plugin. Manages the exchange of settings information between the Kapsel app and the SAP Mobile Platform server. Used by the Push plugin.

Using the Kapsel AppUpdate Plugin

The AppUpdate plugin provides server-based updates to the Web application content that is running in the Kapsel application.

AppUpdate Plugin Overview

The AppUpdate plugin lets an administrator remotely update the contents in the `www` folder of a deployed Kapsel application.

This means that updates to the Web application content only, which does not include application bundle contents outside the `www` folder, do not require corresponding updates to the native application bundle on the end-users’ devices.

Note: When you update Web content for applications that are distributed through a public app store, you must adhere to the policies of the app store provider, even though you do not need to go through the formal review process. Do not include updates to content that violates the terms of the app store content review policies, or change the functionality of the application.

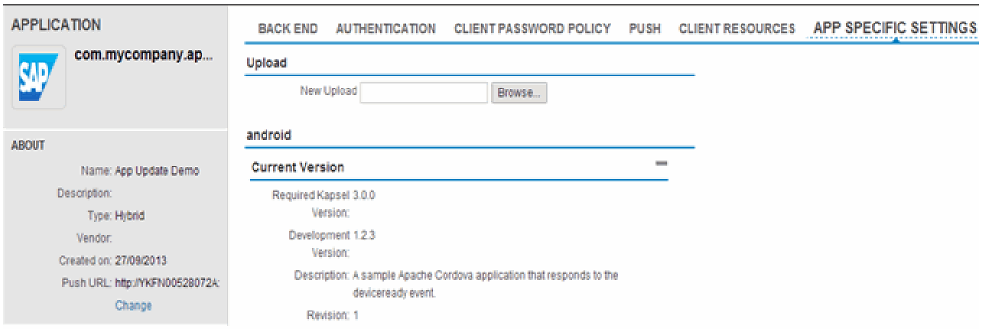
The AppUpdate plugin requires no developer programming, but includes a JavaScript API for customizing the way that application updates occur. The AppUpdate plugin operates in a default mode unless you handle the provided callback APIs.

Configuration Parameters

These configuration parameters are mapped between the Management Cockpit and the `www` folder's `config.xml` file. See *Managing Update Versions and Revisions* for information about usage.

Management Cockpit	config.xml File	Example Value
Revision	**hybridapprevision**	1

This shows an example of app-specific settings configuration for a sample app in Management Cockpit.



APPLICATION

com.mycompany.ap...

ABOUT

Name: App Update Demo
Description:
Type: Hybrid
Vendor:
Created on: 27/09/2013
Push URL: http://YKFN00528072A:
Change

BACK END | AUTHENTICATION | CLIENT PASSWORD POLICY | PUSH | CLIENT RESOURCES | APP SPECIFIC SETTINGS

Upload

New Upload [ ] Browse...

android

Current Version

Required Kapsel Version: 3.0.0
Development Version: 1.2.3
Description: A sample Apache Cordova application that responds to the deviceready event.
Revision: 1

The settings in Management Cockpit are mapped to: Sample <AppDirectory>/www/config.xml configuration<preference name="hybridapprevision" value="1" />

Note: The revision and development versions on SAP Mobile Platform Server are independent values. The Development Version is an optional value for the administrators' convenience, and is not used by the AppUpdate plugin. Revisions are auto incremented upon each update of the `www` folder archive on the server, regardless of whether the development version changes.

Update Flow

1. The administrator uploads a new archive of the `www` folder contents to SAP Mobile Platform Server, where he or she can update one or more platform versions of the `www` folder in an operation. The administrator specifies the minimum version of Kapsel required for the update, and the development version (for example, the build version). The SAP Mobile Platform Server auto increments the revision number when the administrator clicks **Deploy** or **Deploy All**.
   For details about these administrator tasks, as well as information on the underlying REST API that you can use to automate update uploads, see *administrator Guide > Application Administration > Deploying Applications > Defining Application-Specific Settings > Uploading and Deploying Hybrid Apps*.
2. The Kapsel application with the AppUpdate plugin checks with SAP Mobile Platform Server to see if there is a later revision of the `www` folder contents available. If the server has a revision that is greater than the currently downloaded revision, the updated `www` folder is downloaded. SAP Mobile Platform Server and the AppUpdate plugin support delta downloads between revision numbers for a development version of the `www` folder archive. See *Managing Update Versions and Revisions*.
3. If an update to the native Kapsel application bundle is distributed, the currently downloaded revisions of the `www` folder contents are retained through the update. When a newer revision is available on SAP Mobile Platform Server, the delta of the `www` folder contents between the on-device and server revision numbers are downloaded to the Kapsel application. For application bundle updates with very large changes to the `www` folder contents, you can specify a **hybridapprevision** parameter in the application bundle's `config.xml` matching that revision on SAP Mobile Platform Server, so that a delta

download takes place. The `www` folder contents in the Kapsel application bundle are then read, as if from a downloaded revision. Future revisions to the `www` folder contents uploaded to the SAP Mobile Platform Server are downloaded normally by the AppUpdate plugin. See *Managing Update Versions and Revisions*.

4. Once an update is downloaded by the AppUpdate plugin, there are a series of configurable behaviors for handling the end-user experience, and for when the update is applied.

   The default behavior is to display a modal alert to the user with options to accept or defer updates. If the end user accepts the update, the Web application session is restarted within the Kapsel application container, and the new version is loaded.

Example 1: User Accepts App Update

1. The `AppUpdate` function starts and triggers any required log on process.
2. Checking event is fired by AppUpdate.
3. AppUpdate finds that an update is available on the server, and the downloading event fires.
4. Updates finish downloading.
5. The `sap.AppUpdate.onupdateready` function is triggered.
6. A prompt asks the user to reload the application.
7. The user accepts the prompt.
8. The `sap.AppUpdate.reloadApp` function is called and the updated application loads.

Example 2: User Defers Update Action

1. The `AppUpdate` function starts and triggers any required log on process.
2. Checking event is fired by AppUpdate.
3. AppUpdate finds that an update is available on the server, and the downloading event is fired.
4. Updates finish downloading.
5. The `sap.AppUpdate.onupdateready` function is triggered.
6. A prompt asks the user to reload the application.
7. The user cancels the prompt.
8. The `sap.AppUpdate.onupdateready` function is triggered the next time the application is resumed or started.

Configuring the AppUpdate User Experience

You can modify the user experience of the update event by using the `onUpdateReady()` function in the JavaScript application code. These modifications include managing the UI that is shown to the user, text strings, look and feel, position of alert, and so on. You can also add behaviors such as storing a timestamp of the last time the end user was prompted for an update, then waiting for some fixed period of time, such as a week, before again prompting the user to update.

Note: Ensure that any code written for the `onUpdateReady()` function that defers, or otherwise overrides, default update life cycle includes an appropriate recovery method, and does not permanently turn off updates.

Example of Overriding Default Update Behavior

You can assign a custom function to the `onUpdateReady()` event to override default update behavior and force an update that does not ask the user to confirm it. It can either go immediately, or the Administrator can set a date by which it goes.

To do this, add a custom function to `onUpdateReady()`, for example:

```
sap.AppUpdate.onupdateready = myCustomAppUpdateFunction
```

Then, in that custom function, control the update process in whatever way you want. For example, to automatically load the update without first prompting the user for permission, you can add something similar to this:

```
function myCustomAppUpdateFunction = {
// No notification just reload
console.log("Applying application update…");
sap.AppUpdate.reloadApp();
}
```

To use your own custom prompt to warn the user that the app is ready to update, you can do something similar to this:

```
function myCustomAppUpdateFunction = (e){
   console.log("Confirming application update…");
  navigator.notification.confirm('Do you want to install the latest
application update?', doAppUpdateContinue, 'Please confirm', 'Yes,
No');
}

function doAppUpdateContinue(buttonNum){
  if (buttonNum==1) {
    console.log("Applying application update…");
   sap.AppUpdate.reloadApp();
}
};
```

Managing Update Versions and Revisions

SAP Mobile Platform Server with the AppUpdate plugin supports both full updates (a complete download of the `www` folder archive contents on the server) and delta updates (only changed files are downloaded to the device).

These rules govern how updates are downloaded to the device:

1. If the **hybridapprevision** parameter in `config.xml` = 0, or is omitted, the AppUpdate plugin downloads the complete `www` folder archive from the server the first time the device connects. There is no delta comparison between the server revision and the initial copy on the device—the full `www` folder is downloaded, and becomes **hybridapprevision**=*<current_server_revision_number>* on the device.

The initial copy from the application bundle functions normally, until the time that AppUpdate downloads the first revision from the server.

In other words, since the server's auto incremented Revision value starts at 1, a **hybridapprevision** value of 0, or an empty value in the `config.xml` tells the AppUpdate plugin that it is working with the application bundle copy.

2. If the **hybridapprevision** on the device (either set in `config.xml`, or managed by AppUpdate plugin) is greater than 0, and there is a newer revision on the server, then the AppUpdate plugin downloads only changed, new, or deleted resources—a delta update. The delta calculations are executed by SAP Mobile Platform Server before a request from the AppUpdate plugin, and are maintained for updating from any available historical revision on the server to the current revision.

This table shows an example of the update behavior. A valid update path is any distance to the right on the matrix.

Device hybridapprevision	Server Revision					
	1.2.3/1	1.2.3/2	1.2.3/3	1.3.0/4	1.5.1/5	2.0.0/6
0	Full	Full	Full	Full	Full	Full
1		Delta	Delta	Delta	Delta	Delta
2			Delta	Delta	Delta	Delta
3				Delta	Delta	Delta
4					Delta	Delta
5						Delta

Best Practices

- For most smaller Web applications, you should simply omit the **hybridapprevision** parameter from the `config.xml`. This ensures that the revision numbering on-device and on the server is correctly aligned. The only 'full' download occurs upon the Kapsel application bundle's installation and initialization—all subsequent downloads will be deltas.
- For large Web applications (tens of MBs or greater), setting the **hybridapprevision** parameter in the `config.xml` can greatly reduce the download volume. You should ensure that the value on-device matches the correct value for the server. Since the values on the server are auto incremented, it may be advisable when setting this parameter to complete the upload on the server before packaging and distributing the Kapsel application bundle. This ensures that the correct value is used.

Adding the AppUpdate Plugin

To install the AppUpdate plugin, use the Cordova command line interface.

Prerequisites

- Set up the development environment.
- Create your Cordova Project.
- Add your OS platforms.

Task

Note: The AppUpdate plugin has dependencies on the Logon plugin, as well as some Cordova plugins. These are automatically added to your project when you add the AppUpdate plugin.

1. Add the AppUpdate plugin by entering the following at the command prompt, or terminal:

   On Windows:

   ```
   cordova -d plugin add <SDK_HOME>\MobileSDK3\KapselSDK
   \plugins\appupdate
   ```

   On Mac:

   ```
   cordova -d plugin add ~<SDK_HOME>/MobileSDK3/KapselSDK/
   plugins/appupdate
   ```

   Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).
2. (Optional) To see a list of installed plugins in your Cordova project, open a command prompt or terminal window, navigate to your Cordova project folder, and enter:

   ```
   cordova plugins
   ```

   The Cordova command line interface returns a JSON array showing installed plugins, for example:

   ```
   [ 'org.apache.cordova.core.camera',
   'org.apache.cordova.core.device-motion',
   'org.apache.cordova.core.file' ]
   ```

   In this example, the Cordova project has the Cordova core Camera, Accelerator (device-motion), and File plugins installed.
3. Modify the files in the `www` folder for the project as necessary, then copy them to the platform directories by running:

   ```
   cordova -d prepare android
   cordova -d prepare ios
   ```
4. Use the Android IDE or Xcode to deploy and run the project.

Note: If you are using an iOS device, remember to add the "clienthubEntitlements" to the Keychain Groups in the Entitlement section in Xcode.

Kapsel AppUpdate API Reference

The Kapsel AppUpdate API Reference provides usage information for AppUpdate API classes and methods, as well as provides sample source code.

AppUpdate namespace

Used to provide server-based updates to the application content.

The AppUpdate plugin updates the contents of the www folder of deployed Kapsel applications. After an application successfully does a logon to an SAP Mobile Platform 3 server, the AppUpdate plugin is able to download an available update. See Uploading Hybrid Apps in user documentation for information on how to upload an update to SAP Mobile Platform 3 server.

After an update is completely downloaded, the application user is prompted to install the update and restart the application. They can decline if they wish.

Once an update is installed, the application's revision number is updated.

Adding and Removing the AppUpdate Plugin

The AppUpdate plugin is added and removed using the *Cordova CLI*.

To add the AppUpdate plugin to your project, use the following command:

```
cordova plugin add <path to directory containing Kapsel plugins>\appupdate
```

To remove the AppUpdate plugin from your project, use the following command:

```
cordova plugin rm com.sap.mp.cordova.plugins.appupdate
```

Hybrid App Revision Preference

This is an optional preference that tells the AppUdate plugin if the local assets are uploaded to the server, and at what number. If this preference is not provided, the default revision is 0. In your config.xml file you can add the following preference:

```
<preference name="hybridapprevision" value="1" />
```

This means that the local assets in your www folder are uploaded to the server and the server is reporting revision 1 for them. This allows the application to receive a delta update when revision 2 is available instead of a full update.

Caveats

It is important to test that your update has valid HTML, Javascript, and CSS. Otherwise, the update could prevent the application from functioning correctly, and may no longer be updateable. You can test the updated application in a separate simulator or additional test device. You can also validate your Javascript with tools like *JSLint*, or *JSHint*. You can validate CSS with *CSS Lint*.

Methods

Name	Description
addEventListener(eventname, f) on page 25	Add a listener for an AppUpdate event.
reloadApp() on page 26	Replaces the app resources with any newly downloaded resources.
removeEventListener(eventname, f) on page 26	Removes a listener for an AppUpdate event.

reset() on page 27	Removes all local updates and loads the original web assets bundled with the app.
update() on page 27	Force an update check.

Events

Name	Description
checking on page 27	Event fired when AppUpdate is checking for an update.
downloading on page 28	Event fired when AppUpdate has found an update and is starting the download.
error on page 28	Event fired when AppUpdate encounters an error while checking for an update or downloading an update. The status code and status message are provided with this event.
noupdate on page 29	Event fired when AppUpdate finds no available updates on server.
updateready on page 29	Event fired when AppUpdate has a newly downloaded update available.

Source

appupdate.js, line 84 on page 34.

addEventListener(eventname, f) method

Add a listener for an AppUpdate event.

See events for available event names.

Syntax

<static> `addEventListener`(*eventname*, *f*)

Parameters

Name	Type	Description
eventname	string	Name of the app update event.
f	function	Function to call when event is fired.

Example

```
sap.AppUpdate.addEventListener('checking', function(e) {
    console.log("Checking for update");
});
```

Source

appupdate.js, line 133 on page 35.

reloadApp() method

Replaces the app resources with any newly downloaded resources.

Syntax

<static> `reloadApp`()

Example

```
sap.AppUpdate.reloadApp();
```

Source

appupdate.js, line 108 on page 35.

removeEventListener(eventname, f) method

Removes a listener for an AppUpdate event.

See events for available event names.

Syntax

<static> `removeEventListener`(*eventname*, *f*)

Parameters

Name	Type	Description
eventname	string	Name of the app update event.
f	function	Function that was registered.

Example

```
// Adding the listener
var listener = function(e) {
    console.log("Checking for update");
});
sap.AppUpdate.addEventListener('checking', listener);

// Removing the listener
sap.AppUpdate.removeEventListener('checking', listener);
```

Source

appupdate.js, line 153 on page 36.

reset() method

Removes all local updates and loads the original web assets bundled with the app.

Call this after delete registration. Reset calls error callback if called during the update process.

Syntax

<static> `reset`()

Example

```
sap.Logon.core.deleteRegistration(function() {
    sap.AppUpdate.reset();
}, function() {});
```

Source

appupdate.js, line 120 on page 35.

update() method

Force an update check.

By default updates are done automatically during logon and resume. See events for what will be fired during this process.

Syntax

<static> `update`()

Example

```
sap.AppUpdate.update();
```

Source

appupdate.js, line 91 on page 34.

checking event

Event fired when AppUpdate is checking for an update.

Properties

Name	Type	Default	Description
type	string	undefined	The name of the event.Value will be checking.

Type
object

Example

```
sap.AppUpdate.addEventListener('checking', function(e) {
    console.log("Checking for update");
});
```

Source
appupdate.js, line 159 on page 36.

downloading event
Event fired when AppUpdate has found an update and is starting the download.

Properties

Name	Type	Default	Description
type	string	undefined	The name of the event.Value will be downloading.

Type
object

Example

```
sap.AppUpdate.addEventListener('downloading', function(e) {
    console.log("Downloading update");
});
```

Source
appupdate.js, line 163 on page 37.

error event
Event fired when AppUpdate encounters an error while checking for an update or downloading an update. The status code and status message are provided with this event.

Properties

Name	Type	Default	Description
type	string	undefined	The name of the event.Value will be error.

statusCode	int	undefined	The http error code.
statusMessage	string	undefined	The http status message.

Type
object

Example

```
sap.AppUpdate.addEventListener('error', function(e) {
    console.log("Error downloading update. statusCode: " +
e.statusCode + " statusMessage: " + e.statusMessage);
});
```

Source
appupdate.js, line 165 on page 37.

noupdate event

Event fired when AppUpdate finds no available updates on server.

Properties

Name	Type	Default	Description
type	string	undefined	The name of the event.Value will be noupdate.

Type
object

Example

```
sap.AppUpdate.addEventListener('noupdate', function(e) {
    console.log("No update");
});
```

Source
appupdate.js, line 161 on page 36.

updateready event

Event fired when AppUpdate has a newly downloaded update available.

A default handler is already added to sap.AppUpdate.onupdateready that will ask the user to reload the app. When handling this event you should call sap.AppUpdate.reloadApp() to apply the downloaded update.

Properties

Name	Type	Default	Description
type	string	undefined	The name of the event.Value will be up-dateready.
revision	int	undefined	The revision that was downloaded.

Type

object

Example

```
// This will listen for updateready event.
// Note: Use sap.AppUpdate.onupdateready if you want to override the
default handler.
sap.AppUpdate.addEventListener('updateready', function(e) {
    console.log("Update ready");
});

// Override default handler so that we automatically load the update
// without first prompting the user for permission,
sap.AppUpdate.onupdateready = function(e) {
    // No notification just reload
    console.log("Apply application update...");
    sap.AppUpdate.reloadApp();
};

// Override default handler with custom prompt to warn the user that
the
// application is ready to update.
sap.AppUpdate.onupdateready = function() {
    console.log("Confirming application update…");
    navigator.notification.confirm('Update Available',
       function(buttonIndex) {
           if (buttonIndex === 2) {
               console.log("Applying application update…");
               sap.AppUpdate.reloadApp();
           }
       },
       "Update", ["Later", "Relaunch Now"]);
};
```

Source

appupdate.js, line 167 on page 37.

Source code

appupdate.js

```
// ${project.version}
var exec = require('cordova/exec'),
    channel = require('cordova/channel'),
   logonFired = false, // Flag to determine if logon manager is done
    promptActive = false, // Flag to prevent prompt from displaying more than once
  bundle = null; // Internationalization. Loaded with device ready


// Event channels for AppUpdate
var channels = {
    'checking': channel.create('checking'),
    'noupdate': channel.create('noupdate'),
    'downloading': channel.create('downloading'),
    'error': channel.create('error'),
    'updateready': channel.create('updateready')
};

// Holds the dom 0 handlers that are registered for the channels
var domZeroHandlers = {};

// Private callback that plugin calls for events
var _eventHandler = function (event) {
    if (event.type) {
        if (event.type in channels) {
            channels[event.type].fire(event);
        }
    }
};
```

```

    /** @namespace sap */

    /**
     * Used to provide server-based updates to the application content.
     * <br/><br/>
    * The AppUpdate plugin updates the contents of the www folder of deployed Kapsel
     * applications.  After an application successfully does a logon to an SAP Mobile Platform 3
     * server, the AppUpdate plugin is able to download an available update. See Uploading Hybrid Apps in user documentation
     * for information on how to upload an update to SAP Mobile Platform 3 server.
     * <br/><br/>
     * After an update is completely downloaded, the application user is
     * prompted to install the update and restart the application.  They can decline
     * if they wish.
     * <br/><br/>
     * Once an update is installed, the application's revision number is updated.
     * <br/><br/>
     * <b>Adding and Removing the AppUpdate Plugin</b><br/>
     * The AppUpdate plugin is added and removed using the
     * <a href="http://cordova.apache.org/docs/en/edge/guide_cli_index.md.html#The%20Command-line%20Interface">Cordova CLI</a>.<br/>
     * <br/>
     * To add the AppUpdate plugin to your project, use the following command:<br/>
     * cordova plugin add <path to directory containing Kapsel plugins>\appupdate<br/>
     * <br/>
    * To remove the AppUpdate plugin from your project, use the following command:<br/>
     * cordova plugin rm com.sap.mp.cordova.plugins.appupdate
     * <br/><br/>
     *
     * <b>Hybrid App Revision Preference</b><br/>
     * This is an optional preference that tells the AppUpdate plugin if the local
    * assets are uploaded to the server, and at what number.  If this preference is
     * not provided, the default revision is 0.
     * In your config.xml file you can add the following preference:<br/>
     * <preference name="hybridapprevision" value="1" />
    <br/>
    <br/>
     * This means that the local assets in your www folder are uploaded to the server
     * and the server is reporting revision 1 for them.  This allows the application
     * to receive a delta update when revision 2 is available, instead of a full update.
     * <br/><br/>
     *
     * <b>Caveats</b><br/>
     * It is important to test that your update has valid HTML, JavaScript, and CSS.
     * Otherwise, the update could prevent the application from functioning correctly,
    * and may no longer be updateable.  You can test the updated application in a
     * separate simulator or other test device.  You can also validate your
     * JavaScript with tools like <a href="http://www.jslint.com">JSLint</a>, or
     * <a href="http://www.jshint.com">JSHint</a>.
     * You can validate CSS with <a href="http://csslint.net">CSS Lint</a>.
     * <br/><br/>
     *
     * @namespace
     * @alias AppUpdate
     * @memberof sap
     */
    module.exports = {
        /**
         * Force an update check.  By default, updates occur automatically during logon and resume.
         * See events for what is fired during this process.
         * @example
         * sap.AppUpdate.update();
         */
        update: function () {
            // Abort if logon event has not yet fired
            if (logonFired) {
                sap.Logon.unlock(function (connectionInfo) {
                    //Add application ID required for REST call
                    connectionInfo.applicationId = sap.Logon.applicationId;

                    exec(_eventHandler, null, 'AppUpdate', 'update', [connectionInfo]);
                });
            }
        },

        /**
         * Replaces the app resources with any newly downloaded resources.
         * @example
         * sap.AppUpdate.reloadApp();
         */
        reloadApp: function () {
            exec(null, null, 'AppUpdate', 'reloadApp', []);
        },

        /**
         * Removes all local updates and loads the original Web assets bundled with the app. Call this after deleteRegistration.
        * Reset calls error callback if it is called during the update process.
         * @example
         * sap.Logon.core.deleteRegistration(function() {
         *     sap.AppUpdate.reset();
         * }, function() {});
         */
        reset: function (successCallback, errorCallback) {
            exec(successCallback, errorCallback, 'AppUpdate', 'reset', []);
        },

        /**
        * Add a listener for an AppUpdate event.  See events for available event names.
         * @param {string} eventname Name of the app update event.
         * @param {function} f Function to call when event is fired.
         * @example
       * sap.AppUpdate.addEventListener('checking', function(e) {
         *     console.log("Checking for update");
         * });
         */
        addEventListener: function (eventname, f) {
            if (eventname in channels) {
                channels[eventname].subscribe(f);
            }
        },

        /**
        * Removes a listener for an AppUpdate event.  See events for available event names.
         * @param {string} eventname Name of the app update event.
         * @param {function} f Function that was registered.
         * @example
         * // Adding the listener
         * var listener = function(e) {
         *     console.log("Checking for update");
         * });
         * sap.AppUpdate.addEventListener('checking', listener);
         *
         * // Removing the listener
         * sap.AppUpdate.removeEventListener('checking', listener);
         */
        removeEventListener: function (eventname, f) {
            if (eventname in channels) {
                channels[eventname].unsubscribe(f);
            }
        }

        /**
         * Event fired when AppUpdate is checking for an update.
         *
         * @event sap.AppUpdate#checking
         * @type {object}
        * @property {string} type - The name of the event.  Value is 'checking.'
         * @example
        * sap.AppUpdate.addEventListener('checking', function(e) {
         *     console.log("Checking for update");
         * });
         */

        /**
         * Event fired when AppUpdate finds no available updates on the server.
         *
         * @event sap.AppUpdate#noupdate
         * @type {object}
        * @property {string} type - The name of the event.  Value is 'noupdate.'
         * @example
        * sap.AppUpdate.addEventListener('noupdate', function(e) {
         *     console.log("No update");
         * });
         */

        /**
         * Event fired when AppUpdate has found an update and is starting the download.
         *
         * @event sap.AppUpdate#downloading
         * @type {object}
        * @property {string} type - The name of the event.  Value is 'downloading.'
         * @example
         * sap.AppUpdate.addEventListener('downloading', function(e) {
         *     console.log("Downloading update");
         * });
         */

        /**
         * Event fired when AppUpdate encounters an error while checking for an update or while downloading an update.
         * The status code and status message are provided with this event.
         *
         * @event sap.AppUpdate#error
         * @type {object}
        * @property {string} type - The name of the event.  Value is 'error.'
         * @property {int} statusCode - The HTTP error code.
         * @property {string} statusMessage - The HTTP status message.
         * @example
         * sap.AppUpdate.addEventListener('error', function(e) {
        *    console.log("Error downloading update. statusCode: " + e.statusCode + " statusMessage: " + e.statusMessage);
         * });
         */

        /**
         * Event fired when AppUpdate has a newly downloaded update available.
         * A default handler is already added to sap.AppUpdate.onupdateready that will ask the user to reload the app.
         * When using this event, you should call sap.AppUpdate.reloadApp() to apply the downloaded update.
         *
         * @event sap.AppUpdate#updateready
         * @type {object}
         * @property {string} type - The name of the event.  Value is 'updateready.'
         * @property {int} revision - The revision that was downloaded.
         * @example
         *
         * // This listens for updateready event.
         * // Note: Use sap.AppUpdate.onupdateready if you want to override the default handler.
         * sap.AppUpdate.addEventListener('updateready', function(e) {
         *     console.log("Update ready");
         * });
         *
         * // Override the default handler so that the update is automatically loaded,
         * // without first prompting the user for permission.
         * sap.AppUpdate.onupdateready = function(e) {
         *     // No notification just reload
         *     console.log("Apply application update...");
         *     sap.AppUpdate.reloadApp();
         * };
         *
         * // Override the default handler with a custom prompt to notify the user that the
         * // application is ready to update.
         * sap.AppUpdate.onupdateready = function() {
         *     console.log("Confirming application updateâ€¦");
         *     navigator.notification.confirm('Update Available',
         *        function(buttonIndex) {
         *            if (buttonIndex === 2) {
         *                console.log("Applying application updateâ€¦");
         *                 sap.AppUpdate.reloadApp();
         *             }
         *         },
         *         "Update", ["Later", "Relaunch Now"]);
         * };
         */
    };

    // Add getter/setter for DOM0 style events
    for (var type in channels) {
        function defineSetGet(eventType) {
            module.exports.__defineGetter__("on" + eventType, function () {
                return domZeroHandlers[eventType];
            });

            module.exports.__defineSetter__("on" + eventType, function (val) {
                // Remove current handler
                if (domZeroHandlers[eventType]) {
                  module.exports.removeEventListener(eventType, domZeroHandlers[eventType]);
                }

                // Add new handler
                if (val) {
                    domZeroHandlers[eventType] = val;
                    module.exports.addEventListener(eventType, domZeroHandlers[eventType]);
                }
            });
        }

        defineSetGet(type);
    }

    // Add default update ready implementation
    module.exports.onupdateready = function () {
        if (!promptActive) {
            promptActive = true;

            var onConfirm = function (buttonIndex) {
                promptActive = false;
                if (buttonIndex === 2) {
                    // Only reload if we are unlocked
                    sap.Logon.unlock(function (connectionInfo) {
                        //Add application ID required for REST call
                        connectionInfo.applicationId = sap.Logon.applicationId;

                        module.exports.reloadApp();
                    });
                }
            }

            if (!bundle) {
                // Load required translations.
                var i18n = require('com.sap.mp.cordova.plugins.i18n.i18n');
                bundle = i18n.load({
                    path: "plugins/com.sap.mp.cordova.plugins.appupdate/www"
                });
            }

            window.navigator.notification.confirm(
                bundle.get("update_available"),
                onConfirm,
                bundle.get("update"), [bundle.get("later"), bundle.get("relaunch_now")]);
        }
    }

    // When logon is ready, an update check is started.
    document.addEventListener("onSapLogonSuccess", function () {
        logonFired = true;
        module.exports.update();
    }, false);

    document.addEventListener("onSapResumeSuccess", module.exports.update, false);
```

Using the Logon Plugin

The Logon plugin is a component of SAP Mobile Application Framework (MAF) that is exposed as a Cordova plugin and provides an interface to the SAP Afaria client and Client Hub.

Note: Before implementing the Logon plugin, you should thoroughly understand the Client Hub service with which the plugin is integrated to enable onboarding. If you are using an iOS device, you must add the "clienthubEntitlements" to the Keychain Groups in the Entitlement section in Xcode.

Logon Plugin Overview

The Logon plugin manages the application registration and authentication processes through SAP Mobile Platform Server.

Most of the Kapsel plugins rely upon the services provided by the Logon plugin. This plugin manages the process of onboarding applications with SAP Mobile Platform Server, authenticating users, and so on. The Logon plugin, where available, interfaces with Client Hub and pulls certificates from Afaria.

The Logon plugin provides a login screen where the user can enter the values needed to connect to SAP Mobile Platform server, and which stores those values in its own secure data vault. This data vault is separate from the one that is provided with the EncryptedStorage plugin. The Logon plugin also lets the user lock and unlock the application, to protect sensitive data.

Adding the Logon Plugin

To install the Logon plugin, use the Cordova command line interface.

Prerequisites

- Set up the development environment.
- Create your Cordova Project.
- Add your OS platforms.

Task

1. Add the plugin, by entering, at the command prompt:

   On Windows:

   ```
   cordova -d plugin add <SDK_HOME>\MobileSDK3\KapselSDK
   \plugins\logon
   ```

   On Mac:

   ```
   cordova -d plugin add ~<SDK_HOME>/MobileSDK3/KapselSDK/
   plugins/logon
   ```

   Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).

2. (Optional) To see a list of installed plugins in your Cordova project, open a command prompt or terminal window, navigate to your Cordova project folder, and enter:

   ```
   cordova plugins
   ```

   The Cordova command line interface returns a JSON array showing installed plugins, for example:

   ```
   [ 'com.sap.mp.cordova.plugins.corelibs',
   'com.sap.mp.cordova.plugins.logon',
   'org.apache.cordova.console',
   'org.apache.cordova.device',
   'org.apache.cordova.device-orientation',
   'org.apache.cordova.dialogs',
   'org.apache.cordova.inappbrowser' ]
   ```

   In this case, some core Cordova plugins were added, including corelibs, console, device, device-orientation, dialogs, and inAppBrowser. CoreLibs is a utility plugin that is automatically added to every Kapsel project by the command line interface, so you need never add the CoreLibs plugin to a project manually.
3. Configure the application in Management Cockpit.
4. Define a variable in the JavaScript code (typically, this is done in the `index.html` file of your Cordova application) to describe the app ID, for example:

   ```
   var appId = "com.sap.kapsel.mykapselapp";
   ```

Kapsel uses an app ID to tell the server which application definition on the server to use for this application. The app ID that is defined on the server must match what is entered here.

5. Define the connection to the server, for example:

```
var defaultContext = {
     "serverHost" : "192.168.254.159",
     "https" : "false",
     "serverPort" : "8080",
};
```

This prepopulates the fields in the registration dialog that is shown to users during the initialization process.

6. Make a call to the Logon plugin's `init` method as shown:

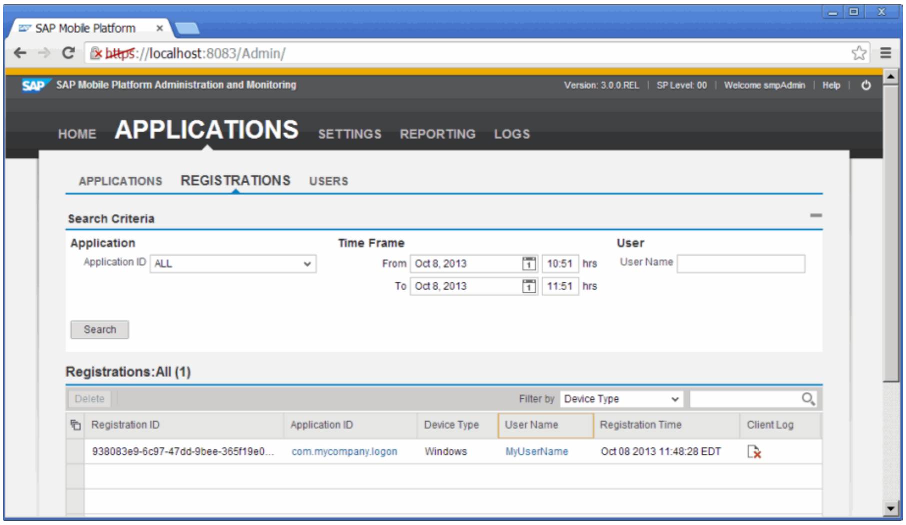
```
//Make call to Logon's Init method to get things registered and
all set up
sap.Logon.init(logonSuccess, logonError, appId, defaultContext);
```

The `init` method gathers information about the environment's security configuration by asking the Afaria client and Client Hub application, if available, sets up and configures the DataVault, connects to the server to register the application connection and authenticate the user. As part of this process, the appropriate screens are shown to gather user input and manage the entire process.

7. Verify the registration in Management Cockpit.

   a) Log in to Management Cockpit.
   b) Click **Applications**.
   c) Click **Registrations**.

   You can see the registration ID following a successful registration.


SAP Mobile Platform
~~https~~://localhost:8083/Admin/
SAP Mobile Platform Administration and Monitoring
Version: 3.0.0.REL | SP Level: 00 | Welcome smpAdmin | Help
HOME APPLICATIONS SETTINGS REPORTING LOGS
APPLICATIONS REGISTRATIONS USERS
Search Criteria
Search
Delete
Filter by

Application

Application ID: ALL

Time Frame

From: Oct 8, 2013 10:51 hrs

To: Oct 8, 2013 11:51 hrs

User

User Name:

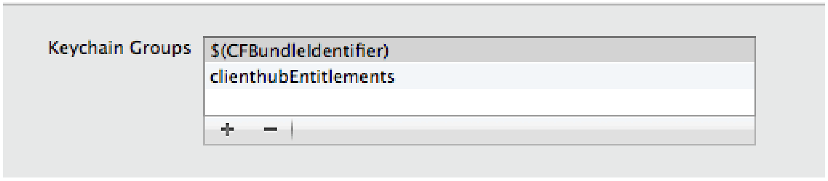
Registrations:All (1)

Filter by: Device Type

Registration ID	Application ID	Device Type	User Name	Registration Time	Client Log
938083e9-6c97-47dd-9bee-365f19e0...	com.mycompany.logon	Windows	MyUserName	Oct 08 2013 11:48:28 EDT	

8. Use the Android IDE or Xcode to deploy and run the project.

Note: If you are deploying to an iOS device, in Xcode, you must add the clienthubEntitlements and $(CFBundleIdentifier) to the keychain group in the Entitlements section as well as the bundle identifier.



Configuring Default Values

Add JavaScript to configure default logon settings.

1. Go to the `<Project Name>/www` folder and open the file where you want to add the JavaScript, for example, `index.html`.
2. Add your code, for example:

```
function logonSuccessCallback(context) {
                console.log("logonSuccessCallback " +
JSON.stringify(context));
            }

            function errorCallback(e) {
                alert("An error occurred");
                alert(JSON.stringify(e));
            }

        function deviceReady() {

              var appId = "theAppId"; // Change this to app id on
server

                // Optional initial connection context
                var context = {
                    "serverHost": "example.com",
                    "https": "false",
                    "serverPort": "8080",
                    "communicatorId": "REST",
                };
             sap.Logon.init(logonSuccessCallback, errorCallback,
appId, context);
            }

           document.addEventListener("deviceready", deviceReady,
false);
```

This example shows the call to the `sap.Logon.init` function, as well as the success and error callbacks that are passed to the `sap.Logon.init` function. It also shows how you can make sure the registration process is started as soon as possible by attaching a listener to the `deviceready` event. Inside the `deviceReady` function, the app ID and the context are defined.

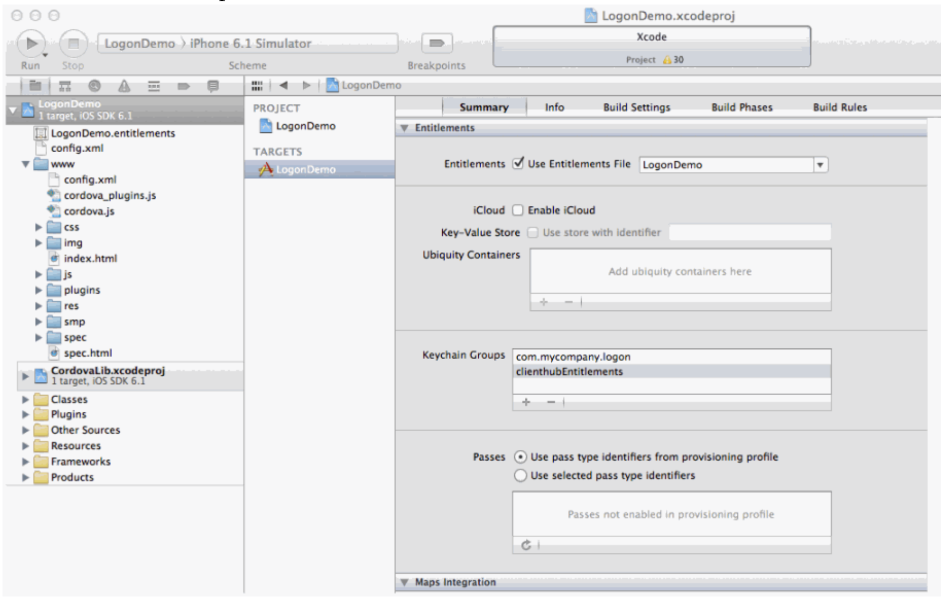
3. Save the file.

Running the Logon Application on iOS

Deploy and run the Logon project on iOS.

1. In a terminal window, make sure you are in the project folder and execute the command:
   ```
   cordova prepare ios
   ```
2. Open Xcode.
3. In a Finder window, browse to your Cordova project folder, `<Project Name>/platforms/ios`.
4. Add the **clienthubEntitlements** keychain group to the Entitlements section of the project.

   This shows an example:



Entitlements ✓ Use Entitlements File LogonDemo
iCloud ☐ Enable iCloud
Key-Value Store ☐ Use store with identifier
Ubiquity Containers
Keychain Groups com.mycompany.logon
clienthubEntitlements
Passes ● Use pass type identifiers from provisioning profile
○ Use selected pass type identifiers

5. Double-click the `<ProjectName>.xcodeproj` file to open the project in Xcode.
6. Select your Simulator type and click the **Run** button.

Removing Fields From the Registration Screen

If your application does not use a relay server, a reverse proxy server, or connect to an SAP Mobile Platform 2.x server, you can remove some of the fields from the registration screen, such as the URL Suffix, Company ID, and Security Config.

1. Open the `StaticScreens.js` file, which is located in SDK_HOME \MobileSDK3\KapselSDK\plugins\logon\www\common\modules.
2. Find the SCR_REGISTRATION screen and reorder, or hide and show fields using the visible:false options. You can also delete unneeded entries.
   For example:
   ```
   SCR_REGISTRATION': {
               id: 'SCR_REGISTRATION',
               fields: {
                           user : {
                               uiKey:'FLD_USER'
                           },
                           password : {
                               uiKey:'FLD_PASS',
                               type: 'password'
                           },
                           serverHost : {
                               uiKey:'FLD_HOST',
                               editable:true
                           },
                           serverPort : {
                               uiKey:'FLD_PORT',
                               type: 'number',
                               editable:true,
                               visible:true
                           },
                           communicatorId : {
                               uiKey: 'FLD_COMMUNICATORID',
                               'default':'REST',
                               visible:false
                           },
                           https: {
                               uiKey:'FLD_IS_HTTPS',
                               type: 'switch',
                               'default':false,
                               visible:false
                           },

               }
           },
   ```
3. Save the file.

Kapsel Logon API Reference

The Kapsel Logon API Reference provides usage information for Logon API classes and methods, as well as provides sample source code.

Logon namespace

The Logon plugin provides screen flows to register an app with an SAP Mobile Platform server.

The logon plugin is a component of the SAP Mobile Application Framework (MAF), exposed as a Cordova plugin. The basic idea is that it provides screen flows where the user can enter the values needed to connect to an SAP Mobile Platform 3.0 server and stores those values in its own secure data vault. This data vault is separate from the one provided with the encrypted storage plugin. In an OData based SAP Mobile Platform 3.0 application, a client must onboard or register with the SAP Mobile Platform 3.0 server to receive an application connection ID for a particular app. The application connection ID must be sent along with each request that is proxied through the SAP Mobile Platform 3.0 server to the OData producer.

Adding and Removing the Logon Plugin

The Logon plugin is added and removed using the *Cordova CLI*.

To add the Logon plugin to your project, use the following command:

cordova plugin add <full path to directory containing Kapsel plugins>\logon

To remove the Logon plugin from your project, use the following command:

cordova plugin rm com.sap.mp.cordova.plugins.logon

Methods

Name	Description
changePassword(onsuccess, onerror) on page 51	This method will launch the UI screen for application users to manage and update the back end passcode that Logon stores in the data vault that is used to authenticate the client to the server.
get(onsuccess, onerror, key) on page 51	Get an (JSON serializable) object from the DataVault for a given key.
init(successCallback, errorCallback, applicationId, [context], [logonView]) on page 52	Initialization method to set up the Logon plugin.
lock(onsuccess, onerror) on page 56	Locks the Logon plugin's secure data vault.

managePasscode(onsuccess, onerror) on page 56	This method will launch the UI screen for application users to manage and update the data vault passcode or, if the SMP server's Client Passcode Policy allows it, enable or disable the passcode to the data vault.
set(onsuccess, onerror, key, value) on page 57	Set an (JSON serializable) object in the DataVault.
showRegistrationData(onsuccess, onerror) on page 58	Calling this method will show a UI screen with values used for registrating application.
unlock(onsuccess, onerror) on page 59	

Type Definitions

Name	Description
errorCallback(errorObject) on page 59	Callback function that is invoked in case of an error.
getSuccessCallback(value) on page 60	Callback function that is invoked upon successfully retrieving an object from the DataVault.
successCallback(context) on page 60	Callback function that is invoked upon successfully registering or unlocking or retrieving the context.
successCallbackNoParameters on page 65	Callback function that will be invoked with no parameters.

Source

LogonController.js, line 1223 on page 108.

applicationId member

The application ID with which *sap.Logon.init* on page 52 was called. It is available here so it is easy to access later.

Syntax

<static> `applicationId`

Example

```
// After calling the init function
alert("The app ID for this app is: " + sap.Logon.applicationId);
```

Source
LogonController.js, line 1281 on page 110.

core member
Direct reference to the logon core object used by the Logon plugin.

This is needed to perform more complex operations that are not generally needed by applications.

There are several functions that can be accessed on the core object:

getState(successCallback,errorCallback) returns the state object of the application to the success callback in the form of a JavaScript object.

getContext(successCallback,errorCallback) returns the context object of the application to the success callback in the form of a JavaScript object.

deleteRegistration(successCallback,errorCallback) deletes the application's registration from the SAP Mobile Platform server and removes

application data on device.

Syntax
<static> `core`

Example

```
var successCallback = function(result){
    alert("Result: " + JSON.stringify(result));
}
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
sap.Logon.core.getState(successCallback,errorCallback);
sap.Logon.core.getContext(successCallback,errorCallback);
sap.Logon.core.deleteRegistration(successCallback,errorCallback);
```

Source
LogonController.js, line 1301 on page 111.

changePassword(onsuccess, onerror) method

This method will launch the UI screen for application users to manage and update the back end passcode that Logon stores in the data vault that is used to authenticate the client to the server.

Syntax

<static> `changePassword(` *onsuccess*, *onerror*)

Parameters

Name	Type	Description
onsuccess	*sap.Logon~successCallback-NoParameters* on page 65	The callback to call if the screen flow succeeds. onsuccess will be called without parameters for this method.
onerror	*sap.Logon~errorCallback* on page 59	The function that is invoked in case of an error.

Example

```
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
var successCallback = function(context){
    alert("Password successfully changed.");
}
sap.Logon.changePassword(successCallback,errorCallback);
```

Source

LogonController.js, line 1455 on page 117.

get(onsuccess, onerror, key) method

Get an (JSON serializable) object from the DataVault for a given key.

Syntax

<static> `get(` *onsuccess*, *onerror*, *key*)

Parameters

Name	Type	Description

onsuccess	*sap.Logon~getSuccessCallback* on page 60	The function that is invoked upon success.It is called with the resulting object as a single parameter. This can be null or undefined, if no object is defined for the given key.
onerror	*sap.Logon~errorCallback* on page 59	The function to invoke in case of error.
key	string	The key with which to query the DataVault.

Example

```
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
var getSuccess = function(value){
    alert("value retrieved from the store: " +
JSON.stringify(value));
}
var setSuccess = function(){
    sap.Logon.get(getSuccess,errorCallback,'someKey');
}
sap.Logon.set(setSuccess,errorCallback,'someKey', 'some string
(could also be an object).');
```

Source

LogonController.js, line 1323 on page 112.

init(successCallback, errorCallback, applicationId, [context], [logonView]) method

Initialization method to set up the Logon plugin.

This will register the application with the SMP server and also authenticate the user with servers on the network. This step must be done first prior to any attempt to communicate with the SMP server.

Syntax

<static> `init(` *successCallback*, *errorCallback*, *applicationId*, [*context*], [*logonView*])

Parameters

Name	Type	Argument	Default	Description

successCallback	*sap.Logon~successCallback* on page 60			The function that is invoked if initialization is successful.The current context is passed to this function as the parameter.
errorCallback	*sap.Logon~errorCallback* on page 59			The function that is invoked in case of an error.
applicationId	string			The unique ID of the application.Must match the application ID on the SAP Mobile Platform server.

context	object	(optional)		The context with default values for application registration.See *sap.Logon~successCallback* on page 60 for the structure of the context object. Note that all properties of the context object are optional, and you only need to specify the properties for which you want to provide default values for. The values will be presented to the application users during the registration process and given them a chance to override these values during runtime.

logonView	string	(optional)	"com/sap/mp/logon/iabui"	The cordova module ID of a custom renderer for the logon, implementing the [showScreen(), close()] interface.Please use the defaul module unless you are absolutely sure that you can provide your own custom implementation. Please refer to JavaScript files inside your Kapsel project's plugins\logon\www\common\modules\ folder as example.

Example

```
// a custom UI can be loaded here
var logonView = sap.logon.IabUi;

// The app ID
var applicationId = "someAppID";

// You only need to specify the fields for which you want to set the
default.   These values are optional because they will be
// used to prefill the fields on Logon's UI screen.
var defaultContext = {
 "serverHost" : "defaultServerHost.com"
\t"https" : false,
\t"serverPort" : "8080",
\t"user" : "user1",
\t"password" : "Zzzzzz123",
\t"communicatorId" : "REST",
\t"securityConfig" : "sec1",
\t"passcode" : "Aaaaaa123",
\t"unlockPasscode" : "Aaaaaa123"
};

var app_context;

var successCallback = function(context){
    app_context = context;
}

var errorCallback = function(errorInfo){
    alert("error: " + JSON.stringify(errorInfo));
}
sap.Logon.init(successCallback, errorCallback, applicationId,
defaultContext, logonView);
```

Source

LogonController.js, line 1273 on page 110.

lock(onsuccess, onerror) method

Locks the Logon plugin's secure data vault.

Syntax

<static> `lock`(*onsuccess*, *onerror*)

Parameters

Name	Type	Description
onsuccess	*sap.Logon~successCallback-NoParameters* on page 65	The function to invoke upon success.
onerror	*sap.Logon~errorCallback* on page 59	The function to invoke in case of error.

Example

```
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
var successCallback = function(){
    alert("Locked!");
}
sap.Logon.lock(successCallback,errorCallback);
```

Source

LogonController.js, line 1362 on page 113.

managePasscode(onsuccess, onerror) method

This method will launch the UI screen for application users to manage and update the data vault passcode or, if the SMP server's Client Passcode Policy allows it, enable or disable the passcode to the data vault.

Syntax

<static> `managePasscode`(*onsuccess*, *onerror*)

Parameters

Name	Type	Description
onsuccess	*sap.Logon~successCallback-NoParameters* on page 65	The function to invoke upon success.
onerror	*sap.Logon~errorCallback* on page 59	The function to invoke in case of error.

Example

```
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
var successCallback = function(context){
    alert("Passcode successfully managed.");
}
sap.Logon.managePasscode(successCallback,errorCallback);
```

Source

LogonController.js, line 1436 on page 116.

set(onsuccess, onerror, key, value) method

Set an (JSON serializable) object in the DataVault.

Syntax

<static> `set(` *onsuccess*, *onerror*, *key*, *value*)

Parameters

Name	Type	Description
onsuccess	*sap.Logon~successCallback-NoParameters* on page 65	The function to invoke upon success. onsuccess will be called without parameters for this method.
onerror	*sap.Logon~errorCallback* on page 59	The function to invoke in case of error.
key	string	The key to store the provided object on.
value	object	The object to be set on the given key.Must be JSON serializable (ie: cannot contain circular references).

Example

```
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
var getSuccess = function(value){
    alert("value retrieved from the store: " +
JSON.stringify(value));
}
var setSuccess = function(){
    sap.Logon.get(getSuccess,errorCallback,'someKey');
}
sap.Logon.set(setSuccess,errorCallback,'someKey', 'some string
(could also be an object).');
```

Source

LogonController.js, line 1346 on page 113.

showRegistrationData(onsuccess, onerror) method

Calling this method will show a UI screen with values used for registrating application.

Syntax

<static> `showRegistrationData(` *onsuccess*, *onerror* `)`

Parameters

Name	Type	Description
onsuccess	*sap.Logon~successCallback-NoParameters* on page 65	The callback to call if the screen flow succeeds. onsuccess will be called without parameters for this method.
onerror	*sap.Logon~errorCallback* on page 59	The function that is invoked in case of an error.

Example

```
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
var successCallback = function(context){
    alert("The showRegistrationData screenflow was successful.");
}
sap.Logon.showRegistrationData(successCallback,errorCallback);
```

Source

LogonController.js, line 1472 on page 118.

unlock(onsuccess, onerror) method

Unlock the Logon plugin's secure data vault if it has been locked (due to being inactive, or *sap.Logon.lock* on page 56 being called), then the user is prompted for the passcode to unlock the application.

If the application is already unlocked, then nothing will be done.

If the application has passcode disabled, then passcode prompt will not be necessary. In all cases if an error does not occur, the success callback is invoked with the current logon context as the parameter.

Syntax

<static> `unlock`(*onsuccess*, *onerror*)

Parameters

Name	Type	Description
onsuccess	*sap.Logon~successCallback* on page 60	The callback to call if the screen flow succeeds. onsuccess will be called with the current logon context as a single parameter.
onerror	*sap.Logon~errorCallback* on page 59	The callback to call if the screen flow fails.

Example

```
var errorCallback = function(errorInfo){
    alert("Error: " + JSON.stringify(errorInfo));
}
var successCallback = function(context){
    alert("Registered and unlocked.  Context: " +
JSON.stringify(context));
}
sap.Logon.unlock(successCallback,errorCallback);
```

Source

LogonController.js, line 1385 on page 114.

errorCallback(errorObject) type

Callback function that is invoked in case of an error.

Syntax
`errorCallback(` *errorObject* `)`

Parameters

Name	Type	Description
errorObject	Object	An object containing properties: 'errorCode', 'errorMessage', and 'errorDomain'. The 'errorCode' is just a number uniquely identifying the error.The 'errorMessage' property is a string with more detailed information of what went wrong. The 'errorDomain' property specifies the domain that the error occurred in.

Source
LogonController.js, line 1476 on page 118.

getSuccessCallback(value) type
Callback function that is invoked upon successfully retrieving an object from the DataVault.

Syntax
`getSuccessCallback(` *value* `)`

Parameters

Name	Type	Description
value	Object	The object that was stored with the given key.Can be null or undefined if no object was stored with the given key.

Source
LogonController.js, line 1482 on page 118.

successCallback(context) type
Callback function that is invoked upon successfully registering or unlocking or retrieving the context.

Syntax

`successCallback(` *context* `)`

Parameters

Name	Type	Description

context	Object	An object containing the current logon context.Two properties of particular importance are applicationEndpointURL, and applicationConnectionId. The context object contains the following properties: "registrationContext": { "serverHost": Host of the server. "domain": Domain for server. Can be used in case of SAP Mobile Platform communication. "resourcePath": Resource path on the server. The path is used mainly for path based reverse proxy but can contain a custom relay server path as well. "https": Marks whether the server should be accessed in a secure way. "serverPort": Port of the server. "user": Username in the backend. "password": Password for the backend user.

		"farmId": FarmId of the server. Can be nil. Used in case of Relay server or SiteMinder. "communicatorId": Id of the communicator manager that will be used for performing the logon. Possible values: IMO / GATEWAY / REST "securityConfig": Security configuration. If nil, the default configuration is used. "mobileUser": Mobile User. Used in case of IMO manual user creation. "activationCode": Activation Code. Used in case of IMO manual user creation. "gatewayClient": The key string that identifies the client on the gateway. Used in Gateway only registration mode. The value will be used as adding the parameter: sap-client=<gateway client> "gatewayPingPath": The custom path of the ping URL on the gateway. Used in case of Gateway only registration mode. }

		"applicationEndpointURL": Contains the application end-point URL after a successful registration. "applicationConnectionId": ID to get after a successful SUP REST registration. Needs to be set in the download request header with key X-SUP-APP-CID "afariaRegistration": manual / automatic / certificate "policyContext": Contains the password policy for the secure store { "alwaysOn": "alwaysOff": "defaultOn": "hasDigits": "hasLowerCaseLetters": "hasSpecialLetters": "hasUpperCaseLetters": "defaultAllowed":

		"expirationDays": "lockTimeout": "minLength": "minUniqueChars": "retryLimit": } "registrationReadOnly": specifies whether context values are coming from clientHub / afaria "policyReadOnly": specifies whether passcode policy is coming from afaria "credentialsByClientHub": specifies whether credentials are coming from clientHub

Source
LogonController.js, line 1478 on page 118.

successCallbackNoParameters type
Callback function that will be invoked with no parameters.

Syntax
`successCallbackNoParameters`()

Source
LogonController.js, line 1480 on page 118.

Source code

LogonController.js

```


var utils = sap.logon.Utils;
var TIMEOUT = 2000;

var _oLogonCore;
var _oLogonView;
var _hasLogonSuccessEventFired = false;

var _providedContext;

var init = function (successCallback, errorCallback, applicationId, context, customView) {


    document.addEventListener("resume",
        function(){
            resume(
            function() { fireEvent('onSapResumeSuccess', arguments);},
             function() { fireEvent('onSapResumeError', arguments);}
            );
        },
        false);

    // The success callback used for the call to _oLogonCore.initLogon(...)
    var initSuccess = function(){
     utils.log('LogonController: LogonCore successfully initialized.');

```

```
              // Now that Logon is initialized, registerOrUnlock is automatically called.
                registerOrUnlock( successCallback, errorCallback );
            }

            var initError = function(error){
               // If a parameter describing the error is given, pass it along.
             // Otherwise, construct something to call the error callback with.
                if( error ) {
                    errorCallback( error );
                } else {
                    errorCallback( utils.Error('ERR_INIT_FAILED') );
                }
            }

            utils.log('LogonController.init enter');
            utils.log(applicationId);
            module.exports.applicationId = applicationId;

            // Make note of the context given (if any)
            if( context ){
                _providedContext = context;
            }

            _oLogonView = customView;
            if (!_oLogonView) {
                _oLogonView = sap.logon.IabUi;
            }

            // coLogonCore.cordova.require("com.sap.mp.cordova.plugins.logon.LogonCore");
            _oLogonCore = sap.logon.Core;
            _oLogonCore.initLogon(initSuccess, initError, applicationId);

            //update exports definition
            module.exports.core = _oLogonCore;
        }

        var fireEvent = function (eventId, args) {
            if (typeof eventId === 'string') {
                //var event = document.createEvent('Events');
                //event.initEvent(eventId, false, false);

                if (!window.CustomEvent) {
                    window.CustomEvent = function(type, eventInitDict) {
                        var newEvent = document.createEvent('CustomEvent');
                        newEvent.initCustomEvent(
                            type,
                            !!(eventInitDict && eventInitDict.bubbles),
                            !!(eventInitDict && eventInitDict.cancelable),
                        (eventInitDict ? eventInitDict.detail : null));
                        return newEvent;
                    };
                }

                var event = new CustomEvent(eventId, { 'detail': { 'id': eventId, 'args': args }});

                setTimeout(function() {
                    document.dispatchEvent(event);
                }, 0);
            } else {
                throw 'Invalid eventId: ' + JSON.stringify(event);
            }
        }

        var FlowRunner = function(onsuccess, onerror, pLogonView, pLogonCore) {

            var onFlowSuccess;
            var onFlowError;
            var onFlowCancel;

            var logonView;
            var logonCore;
            var flow;



            logonView = pLogonView;
            logonCore = pLogonCore;

            onFlowSuccess = function onFlowSuccess() {
                utils.logJSON('onFlowSuccess');
                logonView.close();
                onsuccess.apply(this, arguments);
            }

            onFlowError = function onFlowError() {
                utils.logJSON('onFlowError');
                logonView.close();
                onerror.apply(this, arguments);
            }

            onFlowCancel = function onFlowCancel(){
                utils.logJSON('onFlowCancel');
                //logonView.close();
                onFlowError(new utils.Error('ERR_USER_CANCELLED'));
            }

            var handleCoreStateOnly = function(currentState){
                handleCoreResult(null, currentState);
            }

            var handleCoreResult = function (currentContext, currentState) {
                if (typeof currentContext === undefined) currentContext = null;

                //workaround for defaultPasscodeAllowed
                if (currentState) {
                    if (currentContext && currentContext.policyContext && currentContext.policyContext.defaultAllowed){
                        currentState.defaultPasscodeAllowed = true;
                    }
                    else {
                        currentState.defaultPasscodeAllowed = false;
                    }
                }

```

```
utils.logJSON(currentContext, 'handleCoreResult currentContext');
utils.logJSON(currentState, 'handleCoreResult currentState');


utils.logJSON(flow.name);
var matchFound = false;
var rules = flow.stateTransitions;


ruleMatching:
for (key in rules){

    var rule = flow.stateTransitions[key];
    //utils.logJSON(rule, 'rule');

    //utils.logJSON(rule.condition, 'rule.condition');
    if (typeof rule.condition === 'undefined') {
        throw 'undefined condition in state transition rule';
    }


    if (rule.condition.state === null) {
        if (currentState)
        {
        continue ruleMatching; // non-null state (and rule) mismatch
        }
        //else {
        //    // match:
        //    // rule.condition.state === null &&
        //    // (typeof currentState === 'undefined') // null or undefined
        //}
    }
else if (rule.condition.state !== 'undefined' && currentState){
        utils.log('stateMatching');

        stateMatching:
        for (field in rule.condition.state) {
            utils.log(field);
            if (rule.condition.state[field] === currentState[field])
            {
                utils.log('field matching ' + field);
               continue stateMatching; // state field match
            }
            else {
               utils.log('field mismatching ' + field);
           continue ruleMatching; // state field (and rule) mismatch
            };
        }
    }

    if (rule.condition.context === null) {
        if (currentContext)
        {
            continue ruleMatching; // non-null context (and rule) mismatch
        }
```

```
          //else {
          //    // match:
         //    // rule.condition.context === null &&
          //    // (typeof currentContext === 'undefined') // null or undefined
          //}
      }
  else if (rule.condition.context !== 'undefined' && currentContext){

          utils.log('contextMatching');
          contextMatching:
          for (field in rule.condition.context) {
              utils.log(field);
            if (rule.condition.context[field] === currentContext[field])
              {
                  utils.log('field matching ' + field);
            continue contextMatching;  // context field match
              }
              else {
                 utils.log('field mismatching ' + field);
               continue ruleMatching;  // context field (and rule) mismatch
              };
          }
      }
      utils.log('match found');
      utils.logJSON(rule, 'rule');

      if (typeof rule.action === 'function') {
          rule.action(currentContext);
      }
      else if (typeof rule.action === 'string') {
          // the action is a screenId
          var screenId = rule.action;
          utils.log('handleCoreResult: ' + screenId);
          utils.logKeys(flow.screenEvents[screenId]);
          if(!currentContext){
              currentContext = {};
          }

       if( !currentContext.registrationContext && _providedContext ){
            // The current registrationContext is null, and we have been given a context when initialized,
              // so use the one we were given.
             currentContext.registrationContext = _providedContext;
          } else if (currentContext.registrationContext && _providedContext && !currentContext.registrationReadOnly){
              for (key in _providedContext) {
                  //if (!currentContext.registrationContext[key]){
                  currentContext.registrationContext[key] = _providedContext[key];
                  //}
              }
          }

          logonView.showScreen(screenId, flow.screenEvents[screenId], currentContext);

      }
```

```
            else {
                onFlowError(new utils.Error('ERR_INVALID_ACTION'));
            }

            matchFound = true;
            break ruleMatching;
        }

        if (!matchFound) {
            onFlowError(new utils.Error('ERR_INVALID_STATE'));
        }
    }


    this.run = function(FlowClass) {
        utils.log('FlowRunner.run '  + FlowClass.name);
        flow = new FlowClass(logonCore, logonView, handleCoreResult, onFlowSuccess, onFlowError, onFlowCancel);
        utils.logKeys(flow , 'new flow ');
        logonCore.getState(handleCoreStateOnly, onFlowError);
    }

}


  var MockFlow = function MockFlow(logonCore, logonView, onCoreResult, onFlowSuccess, onFlowError, onFlowCancel) {
   //wrapped into a function to defer evaluation of the references to flow callbacks
        //var flow = {};

```

```
                    this.name = 'mockFlowBuilder';

                    this.stateTransitions = [
                    {
                        condition: {
                            state: {
                                secureStoreOpen: false,
                            }
                        },
                        action: 'SCR_MOCKSCREEN'
                    },
                    {
                        condition: {
                            state: {
                                secureStoreOpen: true,
                            }
                        },
                        action: 'SCR_MOCKSCREEN'
                    },

                    ];

                    this.screenEvents = {
                        'SCR_TURN_PASSCODE_ON': {
                            onsubmit: onFlowSuccess,
                            oncancel: onFlowCancel,
                            onerror: onFlowError,
                        }
                    };

                    utils.log('flow constructor return');
                    //return flow;
```

```
            }

            var RegistrationFlow = function RegistrationFlow(logonCore, logonView, onCoreResult, onFlowSuccess, onFlowError, onFlowCancel) {
            //wrapped into a function to defer evaluation of the references to flow callbacks

                this.name = 'registrationFlowBuilder';

                var registrationInProgress = false;

                var onCancelSSOPin = function() {
onFlowError(errorWithDomainCodeDescription("MAFLogon","0","SSO Passcode set screen was cancelled"));
                }

                var onCancelRegistration = function() {
onFlowError(errorWithDomainCodeDescription("MAFLogon","1","Registration screen was cancelled"));
                }

                // internal methods
                var showScreen = function(screenId) {
                    return function(coreContext) {
                        logonView.showScreen(screenId, this.screenEvents[screenId], coreContext);
                    }.bind(this);
                }.bind(this);

                var onUnlockSubmit = function(context){
                    utils.logJSON(context, 'logonCore.unlockSecureStore');
```

```
        logonCore.unlockSecureStore(onCoreResult, onUnlockError, context)
    }

    var onUnlockError = function(error) {
        utils.logJSON("onUnlockError: " + JSON.stringify(error));

        // TODO switch case according to the error codes
        logonView.showNotification("ERR_UNLOCK_FAILED");
    }

    var noOp = function() { }

    var onRegistrationBackButton = function() {
        if (registrationInProgress == true) {
        utils.log('back button pushed, no operation is required as registration is running');
        }
        else {
            onCancelRegistration();
        }
    }

    var onUnlockVaultWithDefaultPasscode = function() {
        utils.log('logonCore.unlockSecureStore - default passcode');
        var unlockContext = {"unlockPasscode":null};
        logonCore.unlockSecureStore(onCoreResult, onFlowError, unlockContext)
    }

```

```
                var onRegSucceeded = function(context, state) {
                    onCoreResult(context, state);
                    registrationInProgress = false;
                }

                var onRegError = function(error){
                    utils.logJSON(error, 'registration failed');
logonView.showNotification(getRegistrationErrorText(error));
                    registrationInProgress = false;
                }

                var onRegSubmit = function(context){
                    utils.logJSON(context, 'logonCore.startRegistration');
                    registrationInProgress = true;
                    logonCore.startRegistration(onRegSucceeded, onRegError, context)
                }

                var onCreatePasscodeSubmit = function(context){
                    utils.logJSON(context, 'logonCore.persistRegistration');
                    logonCore.persistRegistration(onCoreResult, onCreatePasscodeError, context);
                }

                var onCancelRegistrationError = function(error){
                  utils.logJSON("onCancelRegistrationError: " + JSON.stringify(error));
logonView.showNotification(getRegistrationCancelError(error));
                }

```

```
var onCreatePasscodeError = function(error) {
    utils.logJSON("onCreatePasscodeError: " + JSON.stringify(error));
    logonView.showNotification(getSecureStoreErrorText(error));
}

var onSSOPasscodeSetError = function(error) {
    utils.logJSON("onSSOPasscodeSetError: " + JSON.stringify(error));
    logonView.showNotification(getSSOPasscodeSetErrorText(error));
}

var callGetContext = function(){
    utils.log('logonCore.getContext');
    logonCore.getContext(onCoreResult, onFlowError);
}

 var onFullRegistered = function()
 {
     var getContextSuccessCallback = function(result){

         if(!_hasLogonSuccessEventFired) {
            fireEvent("onSapLogonSuccess", arguments);
             _hasLogonSuccessEventFired = true;
         }

         onFlowSuccess(result);
     }
     utils.log('logonCore.getContext');
```

```
        logonCore.getContext(getContextSuccessCallback, onFlowError);
     }

    var onForgotAppPasscode = function(){
        utils.log('logonCore.deleteRegistration');
        logonCore.deleteRegistration(onFlowError, onFlowError);
    }

    var onForgotSsoPin = function(){
        utils.log('forgotSSOPin');
        logonView.showNotification("ERR_FORGOT_SSO_PIN");
    }

    var onSkipSsoPin = function(){
        utils.logJSON('logonCore.skipClientHub');
        logonCore.skipClientHub(onCoreResult, onFlowError);
    }

    var callPersistWithDefaultPasscode = function(context){
        utils.logJSON(context, 'logonCore.persistRegistration');
        context.passcode = null;
        logonCore.persistRegistration(
                onCoreResult,
                onFlowError,
                context)
    }

    // exported properties
```

```
                this.stateTransitions = [
                {
                    condition: {
                        state: {
                            secureStoreOpen: false,
                            status: 'fullRegistered',
                            defaultPasscodeUsed: true
                        }
                    },
                    action: onUnlockVaultWithDefaultPasscode
                },

                {
                    condition: {
                        state: {
                            secureStoreOpen: false,
                            status: 'fullRegistered'
                        }
                    },
                    action: 'SCR_UNLOCK'
                },


                {
                    condition: {
                        state: {
                            //secureStoreOpen: false, //TODO clarify
                            status: 'fullRegistered',
                            stateClientHub: 'availableNoSSOPin'
                        }
                    },
                    action: 'SCR_SSOPIN_SET'
                },
                {
                    condition: {
                        state: {
                            secureStoreOpen: false,
                            status: 'new'
                        },
                        context: null
                    },
                    action: callGetContext
                },

                {
                    condition: {
                        state: {
                            secureStoreOpen: false,
                            status: 'new',
                            stateClientHub: 'availableNoSSOPin'
                        }
                    },
                    action: 'SCR_SSOPIN_SET'
                },

                {
                    condition: {
                        state: {
                            secureStoreOpen: false,
                            status: 'new',
                            stateClientHub: 'availableInvalidSSOPin'
                        }
```

```
        },
        action: 'SCR_SSOPIN_SET'
    },

    {
        condition: {
            state: {
                secureStoreOpen: false,
                status: 'new',
                stateClientHub: 'availableValidSSOPin'
            },
            context : {
                credentialsByClientHub : true,
                registrationReadOnly : true
            }
        },
        action: function(context){
            utils.logJSON(context, 'logonCore.startRegistration');
          logonCore.startRegistration(onCoreResult, onFlowError, context.registrationContext);
        }
    },
    {
        condition: {
            state: {
                secureStoreOpen: false,
                status: 'new',
                stateClientHub: 'availableValidSSOPin',
                stateAfaria: 'initializationSuccessful'
            },
            context : {
                registrationReadOnly : true,
                afariaRegistration: 'certificate'
            }
        },
        action: function(context){
            utils.logJSON(context, 'logonCore.startRegistration');
          logonCore.startRegistration(onCoreResult, onFlowError, context.registrationContext);
        }
    },
    {
        condition: {
            state: {
                secureStoreOpen: false,
                status: 'new',
                stateClientHub: 'availableValidSSOPin'
            },
            context : {
                registrationReadOnly :true,
                credentialsByClientHub : false
            }
        },
        action: 'SCR_ENTER_CREDENTIALS'
    },


    {
        condition: {
            state: {
                secureStoreOpen: false,
                status: 'new',
                //stateClientHub: 'notAvailable' | 'availableValidSSOPin' | 'skipped' | 'error'
                stateAfaria: 'initializationFailed'
            }
        },
        action: 'SCR_REGISTRATION'
    },

    {
        condition: {
            state: {
                secureStoreOpen: false,
                status: 'new',
                //stateClientHub: 'notAvailable' | 'availableValidSSOPin' | 'skipped' | 'error'
            }
        },
        action: 'SCR_REGISTRATION'
    },

    {
        condition: {
            state: {
                secureStoreOpen: false,
                status: 'new',
                //stateClientHub: 'notAvailable' | 'availableValidSSOPin' | 'skipped' | 'error'
                stateAfaria: 'initializationFailed'
            }
        },
        action: 'SCR_REGISTRATION'
    },

{
    condition: {
        state: {
            secureStoreOpen: false,
            status: 'registered',
            defaultPasscodeUsed: true,
//                        defaultPasscodeAllowed: true,
        }
    },
    action: 'SCR_SET_PASSCODE_OPT_OFF'
},
{
    condition: {
        state: {
            secureStoreOpen: false,
            status: 'registered',
            defaultPasscodeUsed: false,
            defaultPasscodeAllowed: true,
        }
    },
    action: 'SCR_SET_PASSCODE_OPT_ON'
},
{
    condition: {
        state: {
            secureStoreOpen: false,
            status: 'registered',
//                        defaultPasscodeAllowed: false,
        }
    },
    action: 'SCR_SET_PASSCODE_MANDATORY'
},


{
    condition: {
        state: {
            //secureStoreOpen: false, //TODO clarify
            status: 'fullRegistered',
            stateClientHub: 'availableInvalidSSOPin'
        }
    },
    action: 'SCR_SSOPIN_CHANGE'
},
{
    condition: {
        state: {
            secureStoreOpen: true,
            status: 'fullRegistered',
            stateClientHub: 'notAvailable'
        }
    },
    action: onFullRegistered
},
{
    condition: {
        state: {
            secureStoreOpen: true,
            status: 'fullRegistered',
            stateClientHub: 'availableValidSSOPin'
        }
```

```
            },
            action: onFullRegistered
        },
        {
            condition: {
                state: {
                    secureStoreOpen: true,
                    status: 'fullRegistered',
                    stateClientHub: 'skipped'
                }
            },
            action: onFullRegistered
        },



        ];

        this.screenEvents = {
            'SCR_SSOPIN_SET': {
                onsubmit: function(context){
                    utils.logJSON(context, 'logonCore.setSSOPasscode');
                     logonCore.setSSOPasscode(onCoreResult, onSSOPasscodeSetError, context);
                },
                oncancel: onCancelSSOPin,
                onerror: onFlowError,
                onforgot: onForgotSsoPin,
                onskip: onSkipSsoPin
            },

            'SCR_SSOPIN_CHANGE': {
```

```
            onsubmit: function(context){
                utils.logJSON(context, 'logonCore.setSSOPasscode');
             logonCore.setSSOPasscode(onCoreResult, onSSOPasscodeSetError, context);
            },
            oncancel: onSkipSsoPin,
            onerror: onFlowError,
            onforgot: onForgotSsoPin
        },

        'SCR_UNLOCK': {
            onsubmit: onUnlockSubmit,
            oncancel: noOp,
            onerror: onFlowError,
            onforgot: onForgotAppPasscode,
            onerrorack: noOp
        },

        'SCR_REGISTRATION':  {
            onsubmit: onRegSubmit,
            oncancel: onCancelRegistration,
            onerror: onFlowError,
            onbackbutton: onRegistrationBackButton
        },

        'SCR_ENTER_CREDENTIALS' : {
            onsubmit: onRegSubmit,
            oncancel: onCancelRegistration,
            onerror: onFlowError
        },
        'SCR_SET_PASSCODE_OPT_ON': {
            onsubmit: onCreatePasscodeSubmit,
            oncancel: noOp,
            onerror: onFlowError,
            ondisable: showScreen('SCR_SET_PASSCODE_OPT_OFF'),
            onerrorack: noOp
        },
        'SCR_SET_PASSCODE_OPT_OFF': {
            onsubmit: callPersistWithDefaultPasscode,
            oncancel: noOp,
            onerror: onFlowError,
            onenable: showScreen('SCR_SET_PASSCODE_OPT_ON'),
            onerrorack: noOp
        },
        'SCR_SET_PASSCODE_MANDATORY': {
            onsubmit: onCreatePasscodeSubmit,
            oncancel: noOp,
            onerror: onFlowError,
            onerrorack: noOp
        },



    };


    utils.log('flow constructor return');
}



```

```
    var ChangePasswordFlow = function ChangePasswordFlow(logonCore, logonView, onCoreResult, onFlowSuccess, onFlowError, onFlowCancel) {
    //wrapped into a function to defer evaluation of the references to flow callbacks

        this.name = 'changePasswordFlowBuilder';


        // internal methods

        var callUnlockFlow = function(){
                utils.log(this.name + ' triggered unlock');
            registerOrUnlock(onCoreResult,onFlowError);
        }

        var onChangePasswordSubmit = function(context){
            utils.logJSON(context, 'logonCore.changePassword');
            // this logonCore call does not return with context
            logonCore.changePassword(onPasswordChanged, onFlowError, context);
        }


        var onPasswordChanged = function(){
            utils.log('onPasswordChanged');
            logonCore.getContext(onFlowSuccess, onFlowError);
        }

        // exported properties
        this.stateTransitions = [
        {
            condition: {
                state: {
                    secureStoreOpen: false,
                }
            },
            action: callUnlockFlow,
        },
        {
            condition: {
                state: {
                    secureStoreOpen: true,
                }
            },
            action: 'SCR_CHANGE_PASSWORD'
        },

        ];

        this.screenEvents = {
            'SCR_CHANGE_PASSWORD': {
                onsubmit: onChangePasswordSubmit,
                oncancel: onFlowCancel,
                onerror: onFlowError
            }
        };


        utils.log('flow constructor return');
    }

```

```
 var ManagePasscodeFlow = function ManagePasscodeFlow(logonCore, logonView, onCoreResult, onFlowSuccess, onFlowError, onFlowCancel) {
//wrapped into a function to defer evaluation of the references to flow callbacks

     this.name = 'managePasscodeFlowBuilder';

     // internal methods
     var showScreen = function(screenId) {
         return function(coreContext) {
             logonView.showScreen(screenId, this.screenEvents[screenId], coreContext);
         }.bind(this);
     }.bind(this);


     var callChangePasscode = function(context){
             utils.logJSON(context, 'logonCore.changePasscode');
             logonCore.changePasscode(
                 onCoreResult,
                 onChangePasscodeError,
                 context)
     }

     var onChangePasscodeError = function(error) {
         utils.logJSON("onChangePasscodeError: " + JSON.stringify(error));
         logonView.showNotification(getSecureStoreErrorText(error));
     }

     var noOp = function() { }

```

```
    var callDisablePasscode = function(context){
        utils.logJSON(context, 'logonCore.disablePasscode');
        context.passcode = null;
        logonCore.changePasscode(
                onCoreResult,
                onFlowError,
                context)
    }

    var callGetContext = function(){
        utils.log('logonCore.getContext');
        logonCore.getContext(onCoreResult, onFlowError);
    }

    var onPasscodeEnable = function(context){
        utils.logJSON(context, this.name + ' onPasscodeEnable: ');
        //logonCore.changePasscode(onFlowSuccess, onFlowError, context);
        onFlowError();
    }

    // exported properties
    this.stateTransitions = [
    {
        condition: {
            state: {
                secureStoreOpen: true,
            },
            context: null
        },
        action: callGetContext
```

```
                },
                {
                    condition: {
                        state: {
                            secureStoreOpen: false,
                        }
                    },
                    action: onFlowError
                },
                {
                    condition: {
                        state: {
                            secureStoreOpen: true,
                            defaultPasscodeUsed: true,
    //                        defaultPasscodeAllowed: true,
                        }
                    },
                    action: 'SCR_MANAGE_PASSCODE_OPT_OFF'
                },
                {
                    condition: {
                        state: {
                            secureStoreOpen: true,
                            defaultPasscodeUsed: false,
                            defaultPasscodeAllowed: true,
                        }
                    },
                    action: 'SCR_MANAGE_PASSCODE_OPT_ON'
                },
                {
                    condition: {
                        state: {
```

```
                secureStoreOpen: true,
                //defaultPasscodeUsed: [DONTCARE],
                defaultPasscodeAllowed: false,
            }
        },
        action: 'SCR_MANAGE_PASSCODE_MANDATORY'
    },


    ];

    this.screenEvents = {
        'SCR_MANAGE_PASSCODE_OPT_ON': {
            onsubmit: onFlowSuccess,
            oncancel: onFlowSuccess,
            onerror: onFlowError,
            ondisable: showScreen('SCR_CHANGE_PASSCODE_OPT_OFF'),
            onchange: showScreen('SCR_CHANGE_PASSCODE_OPT_ON')
        },
        'SCR_MANAGE_PASSCODE_OPT_OFF': {
            onsubmit: onFlowSuccess,
            oncancel: onFlowSuccess,
            onerror: onFlowError,
            onenable: showScreen('SCR_SET_PASSCODE_OPT_ON')
        },
        'SCR_MANAGE_PASSCODE_MANDATORY': {
            onsubmit: onFlowSuccess,
            oncancel: onFlowSuccess,
            onerror: onFlowError,
            onchange: showScreen('SCR_CHANGE_PASSCODE_MANDATORY')
```

```
                    },


                    'SCR_SET_PASSCODE_OPT_ON': {
                        onsubmit: callChangePasscode,
                        oncancel: onFlowCancel,
                        onerror: onFlowError,
                        ondisable: showScreen('SCR_SET_PASSCODE_OPT_OFF'),
                        onerrorack: noOp
                    },
                    'SCR_SET_PASSCODE_OPT_OFF': {
                        onsubmit: callDisablePasscode,
                        oncancel: onFlowCancel,
                        onerror: onFlowError,
                        onenable: showScreen('SCR_SET_PASSCODE_OPT_ON'),
                        onerrorack: noOp
                    },
                    'SCR_CHANGE_PASSCODE_OPT_ON': {
                        onsubmit: callChangePasscode,
                        oncancel: onFlowCancel,
                        onerror: onFlowError,
                        ondisable: showScreen('SCR_CHANGE_PASSCODE_OPT_OFF'),
                        onerrorack: noOp
                    },
                    'SCR_CHANGE_PASSCODE_OPT_OFF': {
                        onsubmit: callDisablePasscode,
                        oncancel: onFlowCancel,
                        onerror: onFlowError,
                        onenable: showScreen('SCR_CHANGE_PASSCODE_OPT_ON'),
                        onerrorack: noOp
```

```
            },
            'SCR_CHANGE_PASSCODE_MANDATORY': {
                onsubmit: callChangePasscode,
                oncancel: onFlowCancel,
                onerror: onFlowError,
                onerrorack: noOp
            },

        };


        utils.log('flow constructor return');
    }

    var ShowRegistrationFlow = function ShowRegistrationFlow(logonCore, logonView, onCoreResult, onFlowSuccess, onFlowError, onFlowCancel) {
   //wrapped into a function to defer evaluation of the references to flow callbacks

        this.name = 'showRegistrationFlowBuilder';

        var showRegistrationInfo = function(context) {
          logonView.showScreen('SCR_SHOW_REGISTRATION', this.screenEvents['SCR_SHOW_REGISTRATION'], context);
        }.bind(this);

        var callGetContext = function(){
            utils.log('logonCore.getContext');
            logonCore.getContext(onCoreResult, onFlowError);
        }

        // exported properties
        this.stateTransitions = [
        {
            condition: {
                state: {
                    secureStoreOpen: true,

                },
                context: null
            },
            action: callGetContext
        },
        {
            condition: {
                secureStoreOpen: true,
            },
            action: showRegistrationInfo
        }

        ];

        this.screenEvents = {
            'SCR_SHOW_REGISTRATION': {
                oncancel: onFlowSuccess,
                onerror: onFlowError
            }
        };


        utils.log('flow constructor return');
    }

```

```
    // === flow launcher methods =====================================


            var resume = function (onsuccess, onerror) {

                var onUnlockSuccess = function(){
                    _oLogonCore.onEvent(onsuccess, onerror, 'RESUME');
                }

                var onGetStateSuccess = function(state) {
                //call registration flow only if the status is fullregistered in case of resume, so logon screen will not loose its input values
                    if (state.status == 'fullRegistered') {
                        registerOrUnlock(onUnlockSuccess, onerror);
                    }
                }

                getState(onGetStateSuccess, onerror);
            }


            var get = function (onsuccess, onerror, key) {

                var onUnlockSuccess = function(){
                    _oLogonCore.getSecureStoreObject(onsuccess, onerror, key);
                }

                registerOrUnlock(onUnlockSuccess, onerror);
            }



            var set = function (onsuccess, onerror, key, value)
{

                var onUnlockSuccess = function(){
                    _oLogonCore.setSecureStoreObject(onsuccess,
onerror, key, value);
                }

                registerOrUnlock(onUnlockSuccess, onerror);
            }



            var lock = function (onsuccess, onerror) {
                _oLogonCore.lockSecureStore(onsuccess,
onerror);
            }

            var getState = function (onsuccess, onerror) {
                _oLogonCore.getState(onsuccess, onerror);
            }


            var registerOrUnlock = function(onsuccess, onerror)
{
              var flowRunner = new FlowRunner(onsuccess, onerror,
_oLogonView, _oLogonCore);
                flowRunner.run(RegistrationFlow);
            }

            var changePassword = function(onsuccess, onerror) {

                var onUnlockSuccess = function(){
                 var innerFlowRunner = new FlowRunner(onsuccess, onerror, _oLogonView, _oLogonCore);
                    innerFlowRunner.run(ChangePasswordFlow);
                }

                registerOrUnlock(onUnlockSuccess, onerror);
            }


            var forgottenPasscode = function(onsuccess, onerror) {

                var onUnlockSuccess = function(){
                 var innerFlowRunner = new FlowRunner(onsuccess, onerror, _oLogonView, _oLogonCore);
                    innerFlowRunner.run(MockFlow);
                }

                registerOrUnlock(onUnlockSuccess, onerror);
            }

            var managePasscode = function(onsuccess, onerror) {

                var onUnlockSuccess = function(){
                 var innerFlowRunner = new FlowRunner(onsuccess, onerror, _oLogonView, _oLogonCore);
                    innerFlowRunner.run(ManagePasscodeFlow);
                }

                registerOrUnlock(onUnlockSuccess, onerror);
            }

```

```
        var showRegistrationData = function(onsuccess, onerror) {
            var onUnlockSuccess = function(){
            var innerFlowRunner = new FlowRunner(onsuccess, onerror, _oLogonView, _oLogonCore);
                innerFlowRunner.run(ShowRegistrationFlow);
            }

            registerOrUnlock(onUnlockSuccess, onerror);
        }

        var getSecureStoreErrorText = function(error) {
utils.logJSON('LogonController.getSecureStoreErrorText: ' + JSON.stringify(error));

            var errorText;

           if(error.errorCode === '14' && error.errorDomain === 'MAFSecureStoreManagerErrorDomain')
                errorText = "ERR_PASSCODE_TOO_SHORT";
            else if(error.errorCode === '10' && error.errorDomain === 'MAFSecureStoreManagerErrorDomain')
                errorText = "ERR_PASSCODE_REQUIRES_DIGIT";
            else if(error.errorCode === '13' && error.errorDomain === 'MAFSecureStoreManagerErrorDomain')
                errorText = "ERR_PASSCODE_REQUIRES_UPPER";
            else if(error.errorCode === '11' && error.errorDomain === 'MAFSecureStoreManagerErrorDomain')
                errorText = "ERR_PASSCODE_REQUIRES_LOWER";
            else if(error.errorCode === '12' && error.errorDomain === 'MAFSecureStoreManagerErrorDomain')
                errorText = "ERR_PASSCODE_REQUIRES_SPECIAL";
            else if(error.errorCode === '15' && error.errorDomain === 'MAFSecureStoreManagerErrorDomain')
```

```
                errorText = "ERR_PASSCODE_UNDER_MIN_UNIQUE_CHARS";
            else {
                errorText = "ERR_SETPASSCODE_FAILED";
            }

            return errorText;
        }

        var getSSOPasscodeSetErrorText = function(error) {
            utils.logJSON('LogonController.getSSOPasscodeSetErrorText: ' + JSON.stringify(error));

            var errorText;

            if (error.errorDomain === 'MAFLogonCoreErrorDomain') {
                if (error.errorCode === '16') {
                    errorText = "ERR_SSO_PASSCODE_SET_ERROR";
                }
            }

            return errorText;
        }

        var getRegistrationErrorText = function(error) {
            utils.logJSON('LogonController.getRegistrationErrorText: ' + JSON.stringify(error));

            var errorText;

            if (error.errorDomain === 'MAFLogonCoreErrorDomain') {
                if (error.errorCode === '80003') {
                    errorText = "ERR_REG_FAILED_WRONG_SERVER";
                }
                //in case of wrong application id
                else if (error.errorCode === '404') {
                    errorText = "ERR_REG_FAILED";
                }
                else if (error.errorCode === '401') {
                    errorText = "ERR_REG_FAILED_UNATHORIZED";
                }
                else {
                    errorText = "ERR_REG_FAILED";
                }
            }

            return errorText;
        }

        var getRegistrationCancelError = function(error) {
utils.logJSON('LogonController.getRegistrationCancelError: ' + JSON.stringify(error));

            var errorText;

            errorText = "ERR_REGISTRATION_CANCEL";

            return errorText;
        }

```

```
    var errorWithDomainCodeDescription = function(domain, code, description) {
        var error = {
            errorDomain: domain,
            errorCode: code,
            errorMessage: description
        };

        return error;
    }




// =================== exported (public) members ====================

    /**
     * The Logon plugin provides screen flows to register an app with an SAP Mobile Platform server.<br/>
     * <br/>
     * The logon plugin is a component of the SAP Mobile Application Framework (MAF), exposed as a Cordova plugin. The basic
     * idea is that it provides screen flows where the user can enter the values needed to connect to an SAP Mobile Platform 3.0 server and
     * stores those values in its own secure data vault. This data vault is separate from the one provided with the
     * encrypted storage plugin. In an OData based SAP Mobile Platform 3.0 application, a client must onboard or register with the SAP Mobile Platform 3.0
     * server to receive an application connection ID for a particular app. The application connection ID must be sent
     * along with each request that is proxied through the SAP Mobile Platform 3.0 server to the OData producer.<br/>
     * <br/>
     * <b>Adding and Removing the Logon Plugin</b><br/>
```

```
         * The Logon plugin is added and removed using the
         * <a href="http://cordova.apache.org/docs/en/edge/guide_cli_index.md.html#The%20Command-line%20Interface">Cordova CLI</a>.<br/>
         * <br/>
         * To add the Logon plugin to your project, use the following command:<br/>
        * cordova plugin add <full path to directory containing Kapsel plugins>\logon<br/>
         * <br/>
        * To remove the Logon plugin from your project, use the following command:<br/>
         * cordova plugin rm com.sap.mp.cordova.plugins.logon
         *
         * @namespace
          * @alias Logon
          * @memberof sap
          */
        module.exports = {

        /**
       * Initialization method to set up the Logon plugin.  This will register the application with the SMP server and also authenticate the user
         * with servers on the network.  This step must be done first prior to any attempt to communicate with the SMP server.
         *
         * @method
       * @param {sap.Logon~successCallback} successCallback The function that is invoked if initialization is successful.  The current
         * context is passed to this function as the parameter.
         * @param {sap.Logon~errorCallback} errorCallback The function that is invoked in case of an error.
         * @param {string} applicationId The unique ID of the application.  Must match the application ID on the SAP Mobile Platform server.
         * @param {object} [context] The context with default values for application registration.  See {@link sap.Logon~successCallback} for the structure
        * of the context object.  Note that all properties of the context object are optional, and you only need to specify the properties
        * for which you want to provide default values for.  The values will be presented to the application users during the registration process and given them
         * a chance to override these values during runtime.
         * @param {string} [logonView="com/sap/mp/logon/iabui"] The cordova module ID of a custom renderer for the logon,
         * implementing the [showScreen(), close()] interface.  Please use the defaul module unless you are absolutely sure that you can provide your own
         * custom implementation.  Please refer to JavaScript files inside your Kapsel project's plugins\logon\www\common\modules\ folder as example.
         * @example
         * // a custom UI can be loaded here
         * var logonView = sap.logon.IabUi;
         *
         * // The app ID
         * var applicationId = "someAppID";
         *
         * // You only need to specify the fields for which you want to set the default.   These values are optional because they will be
         * // used to prefill the fields on Logon's UI screen.
         * var defaultContext = {
         *  "serverHost" : "defaultServerHost.com"
         *    "https" : false,
         *    "serverPort" : "8080",
         *    "user" : "user1",
         *    "password" : "Zzzzzz123",
         *    "communicatorId" : "REST",
         *    "securityConfig" : "sec1",
         *    "passcode" : "Aaaaaa123",
         *    "unlockPasscode" : "Aaaaaa123"
         * };
         *
         * var app_context;
         *
         * var successCallback = function(context){
         *     app_context = context;
         * }
         *
         * var errorCallback = function(errorInfo){
         *     alert("error: " + JSON.stringify(errorInfo));
         * }
         * sap.Logon.init(successCallback, errorCallback, applicationId, defaultContext, logonView);
         */
           init: init,

           /**
          * The application ID with which {@link sap.Logon.init} was called.  It is available here so it is easy to access later.
            * @example
            * // After calling the init function
            * alert("The app ID for this app is: " + sap.Logon.applicationId);
            */
           applicationId: null,
        /**
        * Direct reference to the logon core object used by the Logon plugin.  This is needed to perform more complex operations that
         * are not generally needed by applications. <br/>
       * There are several functions that can be accessed on the core object:<br/>
```

```
         * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; getState(successCallback,errorCallback) returns the state object of the application to the success callback in the form of a JavaScript object.<br/>
         * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; getContext(successCallback,errorCallback) returns the context object of the application to the success callback in the form of a JavaScript object.<br/>
         * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; deleteRegistration(successCallback,errorCallback) deletes the application's registration from the SAP Mobile Platform server and removes<br/>
         * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp; application data on device.<br/>
         * @example
         * var successCallback = function(result){
         *     alert("Result: " + JSON.stringify(result));
         * }
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * sap.Logon.core.getState(successCallback,errorCallback);
         * sap.Logon.core.getContext(successCallback,errorCallback);
         * sap.Logon.core.deleteRegistration(successCallback,errorCallback);
         */
                core: _oLogonCore, //Must be updated after init

        /**
         * Get an  (JSON serializable) object from the DataVault for a given key.
         * @method
         * @param {sap.Logon~getSuccessCallback} onsuccess The function that is invoked
       * upon success.  It is called with the resulting object as a single parameter.
```

```
        * This can be null or undefined, if no object is defined for the given key.
        * @param {sap.Logon~errorCallback} onerror The function to invoke in case of error.
         * @param {string} key The key with which to query the DataVault.
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var getSuccess = function(value){
         *     alert("value retrieved from the store: " + JSON.stringify(value));
         * }
         * var setSuccess = function(){
         * sap.Logon.get(getSuccess,errorCallback,'someKey');
         * }
         * sap.Logon.set(setSuccess,errorCallback,'someKey', 'some string (could also be an object).');
         */
                get: get,

        /**
         * Set an (JSON serializable) object in the DataVault.
         * @method
         * @param {sap.Logon~successCallbackNoParameters} onsuccess The function to invoke upon success.
         * onsuccess will be called without parameters for this method.
        * @param {sap.Logon~errorCallback} onerror The function to invoke in case of error.
         * @param {string} key The key to store the provided object on.
        * @param {object} value The object to be set on the given key.  Must be JSON serializable (ie:
         * cannot contain circular references).
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var getSuccess = function(value){
         *     alert("value retrieved from the store: " + JSON.stringify(value));
         * }
         * var setSuccess = function(){
         * sap.Logon.get(getSuccess,errorCallback,'someKey');
         * }
         * sap.Logon.set(setSuccess,errorCallback,'someKey', 'some string (could also be an object).');
         */
                set: set,

        /**
         * Locks the Logon plugin's secure data vault.
         * @method
         * @param {sap.Logon~successCallbackNoParameters} onsuccess The function to invoke upon success.
        * @param {sap.Logon~errorCallback} onerror The function to invoke in case of error.
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var successCallback = function(){
         *     alert("Locked!");
         * }
         * sap.Logon.lock(successCallback,errorCallback);
         */
                lock: lock,

        /**
         * Unlock the Logon plugin's secure data vault if it has been locked (due to being inactive, or
         * {@link sap.Logon.lock} being called), then the user is prompted for the passcode to unlock the
         * application.<br/>
         * If the application is already unlocked, then nothing will be done.<br/>
         * If the application has passcode disabled, then passcode prompt will not be necessary.
         * In all cases if an error does not occur, the success callback is invoked with the current logon context
         * as the parameter.
         * @method
         * @param {sap.Logon~successCallback} onsuccess - The callback to call if the screen flow succeeds.
         * onsuccess will be called with the current logon context as a single parameter.
         * @param {sap.Logon~errorCallback} onerror - The callback to call if the screen flow fails.
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var successCallback = function(context){
         *     alert("Registered and unlocked.  Context: " + JSON.stringify(context));
         * }
         * sap.Logon.unlock(successCallback,errorCallback);
         */
            unlock: registerOrUnlock,

        /**
         * This is an alias for registerOrUnlock.  Calling this function is equivalent
        * to calling {@link sap.Logon.unlock} since both of them are alias to registerOrUnlock.
         * @method
         * @private
         */
                registerUser: registerOrUnlock,

        /**
         * This function registers the user and creates a new unlocked DataVault to store the registration
         * information.<br/>
         * If the user has already been registered, but the application is locked (due to being inactive, or
        * {@link sap.Logon.lock} being called), then the user is prompted for the passcode to unlock the
         * application.<br/>
         * If the application is already unlocked, then nothing will be done.<br/>
         * In all cases if an error does not occur, the success callback is invoked with the current logon context
         * as the parameter.
         * @method
         * @param {sap.Logon~successCallback} onsuccess - The callback to call if the screen flow succeeds.
        * onsuccess will be called with the current logon context as a single parameter.
        * @param {sap.Logon~errorCallback} onerror - The callback to call if the screen flow fails.
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var successCallback = function(context){
         *     alert("Registered and unlocked.  Context: " + JSON.stringify(context));
         * }
         *
sap.Logon.registerOrUnlock(successCallback,errorCallback);
         * @private
         */
            registerOrUnlock: registerOrUnlock,

        /**
         * This method will launch the UI screen for application
users to manage and update the data vault passcode or,
         * if the SMP server's Client Passcode Policy allows it,
enable or disable the passcode to the data vault.
         *
         * @method
         * @param {sap.Logon~successCallbackNoParameters}
onsuccess - The function to invoke upon success.
         * @param {sap.Logon~errorCallback} onerror - The function
to invoke in case of error.
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var successCallback = function(context){
         *     alert("Passcode successfully managed.");
         * }
         *
sap.Logon.managePasscode(successCallback,errorCallback);
         */
            managePasscode: managePasscode,

        /**
         * This method will launch the UI screen for application
users to manage and update the back end passcode that Logon stores in
the
         * data vault that is used to authenticate the client to
the server.
         *
         * @method
         * @param {sap.Logon~successCallbackNoParameters}
onsuccess - The callback to call if the screen flow succeeds.
         * onsuccess will be called without parameters for this
method.
        * @param {sap.Logon~errorCallback} onerror The function
that is invoked in case of an error.
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var successCallback = function(context){
         *     alert("Password successfully changed.");
         * }
         *
sap.Logon.changePassword(successCallback,errorCallback);
         */
                changePassword: changePassword,

        /**
        * Calling this method will show a UI screen with values
used for registrating application.
         * @method
         * @param {sap.Logon~successCallbackNoParameters}
onsuccess - The callback to call if the screen flow succeeds.
         * onsuccess will be called without parameters for this
method.
        * @param {sap.Logon~errorCallback} onerror The function
that is invoked in case of an error.
         * @example
         * var errorCallback = function(errorInfo){
         *     alert("Error: " + JSON.stringify(errorInfo));
         * }
         * var successCallback = function(context){
         *     alert("The showRegistrationData screenflow was
successful.");
```

```
         * }
         * sap.Logon.showRegistrationData(successCallback,errorCallback);
         */
                showRegistrationData: showRegistrationData,

            };

    /**
     * Callback function that is invoked in case of an error.
     *
     * @callback sap.Logon~errorCallback
     *
     * @param {Object} errorObject An object containing properties: 'errorCode', 'errorMessage', and 'errorDomain'.
     * The 'errorCode' is just a number uniquely identifying the error.  The 'errorMessage'
     * property is a string with more detailed information of what went wrong.  The 'errorDomain' property specifies
     * the domain that the error occurred in.
     */

    /**
     * Callback function that is invoked upon successfully registering or unlocking or retrieving the context.
     *
     * @callback sap.Logon~successCallback
     *
     * @param {Object} context An object containing the current logon context.  Two properties of particular importance
     * are applicationEndpointURL, and applicationConnectionId.
     * The context object contains the following properties:<br/>
     * "registrationContext": {<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"serverHost": Host of the server.<br/>
    * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"domain": Domain for server. Can be used in case of SAP Mobile Platform communication.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"resourcePath": Resource path on the server. The path is used mainly for path based reverse proxy but can contain a custom relay server path as well.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"https": Marks whether the server should be accessed in a secure way.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"serverPort": Port of the server.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"user": Username in the backend.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"password": Password for the backend user.<br/>
    * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"farmId": FarmId of the server. Can be nil. Used in case of Relay server or SiteMinder.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"communicatorId": Id of the communicator manager that will be used for performing the logon. Possible values: IMO / GATEWAY / REST<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"securityConfig": Security configuration. If nil, the default configuration is used.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"mobileUser": Mobile User. Used in case of IMO manual user creation.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"activationCode": Activation Code. Used in case of IMO manual user creation.<br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"gatewayClient": The key string that identifies the client on the gateway. Used in Gateway only registration mode. The value will be used as adding the parameter: sap-client=<gateway client><br/>
     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"gatewayPingPath":
```

```
The custom path of the ping URL on the gateway. Used in case of Gateway only registration mode.<br/>

     * }<br/>

     * "applicationEndpointURL": Contains the application endpoint URL after a successful registration.<br/>

     * "applicationConnectionId": ID to get after a successful SUP REST registration. Needs to be set in the download request header with key X-SUP-APPCID<br/>

    * "afariaRegistration": manual / automatic / certificate<br/>

     * "policyContext": Contains the password policy for the secure store {<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"alwaysOn":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"alwaysOff":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"defaultOn":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"hasDigits":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"hasLowerCaseLetters":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"hasSpecialLetters":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"hasUpperCaseLetters":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"defaultAllowed":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"expirationDays":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"lockTimeout":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"minLength":<br/>

     * &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"minUniqueChars":<br/>
* &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;"retryLimit":<br/>
     * }<br/>
     * "registrationReadOnly": specifies whether context values are coming from clientHub / afaria<br/>
     * "policyReadOnly": specifies whether passcode policy is coming from afaria<br/>
     * "credentialsByClientHub": specifies whether credentials are coming from clientHub
     */

     /**
      * Callback function that will be invoked with no parameters.
      *
      * @callback sap.Logon~successCallbackNoParameters
      */

     /**
      * Callback function that is invoked upon successfully retrieving an object from the DataVault.
      *
      * @callback sap.Logon~getSuccessCallback
      *
     * @param {Object} value The object that was stored with the given key.  Can be null or undefined if no object was stored
      * with the given key.
      */


```

Using the Kapsel AuthProxy Plugin

The Kapsel AuthProxy plugin automates the process of accepting SSL certificates returned by a call to a Web resource.

AuthProxy Plugin Overview

The AuthProxy plugin provides the ability to make HTTPS requests with mutual authentication.

The AuthProxy plugin allows you to specify a certificate to include in an HTTPS request that identifies the client to the server, which allows the server to verify the identity of the client. An example of where you might need mutual authenticaion is in the onboarding process, when you register with an application, or, to access an OData producer. You can make HTTPS requests with no authentication, with basic authentication, or by using certificates. Supported certificate sources include file, system key manager, and Afaria.

Sending Requests

There are two functions for sending requests:

- `get` = function (url, header, successCB, errorCB, user, password, timeout, certSource). This is a convenience function and provides no additional functionality compared to the `sendRequest` function. It just calls the `sendRequest` function with the method set to GET and no `requestBody`.
- `sendRequest` = function (method, url, header, requestBody, successCB, errorCB, user, password, timeout, certSource).

Constructor Functions

There are three constructor functions to make objects that you can use for certificates:

- `CertificateFromFile` = function (Path, Password, CertificateKey)
- `CertificateFromLogonManager` = function(AppID)
- `CertificateFromStore` = function (CertificateKey)

Note: The `success` callback is called upon any response from the server, so be sure to check the status on the response.

Adding the AuthProxy Plugin

Use the Cordova command line interface to install the AuthProxy plugin.

Prerequisites

- Set up the development environment.
- Create your Cordova Project.
- Add your OS platforms.

- On Android these permissions are required:
  - • <uses-permission android:name="android.permission.INTERNET" />
    - <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

        - <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
- On iOS:
    - The plugin depends on `afariaSLL.a`
    - Requires the link flag of "-lstdc++," if not yet included.

Task

1. Add the AuthProxy plugin by entering the following at the command prompt, or terminal:

   On Windows:

   ```
   cordova -d plugin add <SDK_HOME>\MobileSDK3\KapselSDK
   \plugins\authproxy
   ```

   On Mac:

   ```
   cordova -d plugin add ~<SDK_HOME>/MobileSDK3/KapselSDK/
   plugins/authproxy
   ```

   Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).
2. (Optional) To see a list of installed plugins in your Cordova project, open a command prompt or terminal window, navigate to your Cordova project folder, and enter:

   ```
   cordova plugins
   ```

   The Cordova command line interface returns a JSON array showing installed plugins, for example:

   ```
   [ 'org.apache.cordova.core.camera',
   'org.apache.cordova.core.device-motion',
   'org.apache.cordova.core.file' ]
   ```

   In this example, the Cordova project has the Cordova core Camera, Accelerator (device-motion), and File plugins installed.
3. Modify the files in the `www` folder for the project as necessary, then copy them to the platform directories by running:

   ```
   cordova -d prepare android
   cordova -d prepare ios
   ```
4. Use the Android IDE or Xcode to deploy and run the project.
5. (Optional) For iOS, if the application uses Afaria mutual certificate authentication, or if multiple applications on the devices need to share the credentials, you must first build and deploy Client Hub to the device, and then add the "clienthubEntitlements" and "$(CFBundleIdentifier)" items to the shared keychain groups in the application's project settings in Xcode.

Adding User Permissions to the Android Manifest File

Add user permissions to the Android project.

1. In the Android IDE, open the AndroidManifest.xml file.
2. Add user permissions in the `AndroidManifest.xml` file, for example:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/
android"
    package="smp.tutorial.android"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="15" />
<uses-permission android:name="android.permission.INTERNET">
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE">
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE">
 <application>

        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>
  </application>
</manifest>
```

3. Select **File** > **Save**.

Adding Cookies to a Request

To add cookies to a request for authentication, use the header object that is passed to the get/sendRequest functions.

Only the cookie name and value should be set this way. The other pieces of the cookie (domain, path, and so on) are set automatically based on the URL the request is made against. The cookie is treated as a session cookie and sent on future requests as appropriate. The API examples below show an example of how to set a cookie with the header object.

```
var successCallback = function( result ){
     if( result.status === 200 ) {
          alert("success\!
        Response text: " + result.responseText );
     } else {
          alert("Not success, response status:
        " + result.status);
     }
}
```

```
var failureCallback = function( error ) {
     alert("Error! Code: " + error.errorCode + "\n" +
error.description + "\nNative error code: " +
error.nativeErrorCode );
}

// setting a cookie with a request
var header = {cookie:
"customCookieName=customCookieValue;anotherName=AnotherValue"};

sap.AuthProxy.sendRequest("POST", "http://www.example.com/stuff/
etc", header, null, successCallback, failureCallback);
```

Using the AuthProxy Plugin to Register With SAP Mobile Platform Server

This example procedure demonstrates how to use the AuthProxy plugin to register with the SAP Mobile Platform Server using a client certificate.

This example does not use the Logon plugin to perform the registration. You can test certificates on an Android device or emulator, or an iOS device. The server certificate must be installed on the device's system store, so for iOS, the actual device is required.

1. Use the keytool utility to create the server and client certificates.

   The SAP Mobile Platform Server stores its certificates in a file named `smp_keystore.jks`.
2. Download the certificate and generate a certificate signing request (CSR).
3. Import the signed certificate into the keystore.
4. Copy the client's public key to `smp_keystore.jks` so that the server can authenticate the client.
5. Create a security profile in Management Cockpit
6. Import the public and private key of the client certificate to the mobile device using the PKCS12 format.

   Both the client certificate (stored in the keystore client.p12 containing the public and private keys) and the certificate authority's certificate, must be added to the mobile device. You should add the certificate authority's certificate to the device's trust store. The client certificate in this example for Android is placed in a location the application can access it from.

   ```
   adb push SAPServerCA.cer /mnt/sdcard/
   adb push client.p12 /mnt/sdcard/
   adb shell
   cd /mnt/sdcard
   ls
   exit
   ```

   For an iOS device, both certificates can be installed into the device's trusted store by sending them through an e-mail, opening the device browser to a Web page that contains the links to the certificates, or by using the iPhone Configuration Utility. See *http://support.apple.com/kb/DL1465*.

On the iOS device, the certificates can be viewed and uninstalled under **Settings** > **General** > **Profiles**.

In addition to accessing the certificate from the file system and the device's secure store, the client certificate can be provisioned to the device using Afaria and then accessed from Afaria using the Logon plugin using the method `sap.AuthProxy.CertificateFromLogonManager("clientKey").`

7. Create a new Cordova project to perform mutual authentication to the SAP Mobile Platform Server.
8. Add the AuthProxy plugin.
9. Create a new security provider and add an x.509 User Certificate authentication provider.
10. Copy the files to the platform directory by running the **prepare** command.
11. Use the Android IDE or Xcode to deploy and run the project.

Generating Certificates and Keys

Use a PKI system and a trusted CA to generate production-ready certificates and keys that encrypt communication among different SAP Mobile Platform components. You can then use the **keytool** utility to import and export certificate to the keystore.

Note: Any changes to the keystore require the server to be restarted.

Kapsel AuthProxy API Reference

The Kapsel AuthProxy API Reference provides usage information for AuthProxy API classes and methods, as well as provides sample source code.

AuthProxy namespace

The AuthProxy plugin provides the ability to make HTTPS requests with mutual authentication.

The regular XMLHttpRequest does not support mutual authentication. The AuthProxy plugin allows you to specify a certificate to include in an HTTPS request to identify the client to the server. This allows the server to verify the identity of the client. An example of where you might need mutual authenticaion is the onboarding process to register with an application, or, to access an OData producer. This occurs mostly in Business to Business (B2B) applications. This is different from most business to consumer (B2C) web sites where it is only the server that authenticates itself to the client with a certificate.

Adding and Removing the AuthProxy Plugin

The AuthProxy plugin is added and removed using the *Cordova CLI*.

To add the AuthProxy plugin to your project, use the following command:

```
cordova plugin add <path to directory containing Kapsel plugins>\authproxy
```

To remove the AuthProxy plugin from your project, use the following command:

```
cordova plugin rm com.sap.mp.cordova.plugins.authproxy
```

Classes

Name	Description
sap.AuthProxy.CertificateFromFile on page 129	Create certificate source description object for a certificate from a keystore file.
sap.AuthProxy.CertificateFromLogonManager on page 131	Create a certificate source description object for certificates from logon manager.
sap.AuthProxy.CertificateFromStore on page 132	Create a certificate source description object for certificates from the system keystore.

Members

Name	Description
ERR_CERTIFICATE_ALIAS_NOT_FOUND on page 132	Constant indicating the certificate with the given alias could not be found.
ERR_CERTIFICATE_FILE_NOT_EXIST on page 133	Constant indicating the certificate file could not be found.
ERR_CERTIFICATE_INVALID_FILE_FORMAT on page 133	Constant indicating incorrect certificate file format.
ERR_CLIENT_CERTIFICATE_ VALIDATION on page 133	Constant indicating the provided certificate failed validation on the server side.
ERR_FILE_CERTIFICATE_SOURCE_UNSUPPORTED on page 133	Constant indicating the certificate from file is not supported on the current platform.

ERR_GET_CERTIFICATE_FAILED on page 134	Constant indicating failure in getting the certificate.
ERR_HTTP_TIMEOUT on page 134	Constant indicating timeout error while connecting to the server.
ERR_INVALID_PARAMETER_VALUE on page 134	Constant indicating the operation failed due to an invalid parameter (for example, a string was passed where a number was required).
ERR_LOGON_MANAGER_CERTIFICATE_METHOD_NOT_AVAILABLE on page 134	Constant indicating the logon manager certifciate method is not available.
ERR_LOGON_MANAGER_CORE_NOT_AVAILABLE on page 135	Constant indicating the logon manager core library is not available.
ERR_MISSING_PARAMETER on page 135	Constant indicating the operation failed because of a missing parameter.
ERR_NO_SUCH_ACTION on page 135	Constant indicating there is no such Cordova action for the current service.
ERR_SERVER_CERTIFICATE_VALIDATION on page 136	Constant indicating the server certificate failed validation on the client side.
ERR_SERVER_REQUEST_FAILED on page 136	Constant indicating the server request failed.
ERR_SYSTEM_CERTIFICATE_SOURCE_UNSUPPORTED on page 136	Constant indicating the certificate from the system keystore is not supported on the current platform.
ERR_UNKNOWN on page 136	Constant indicating the operation failed with unknown error.

Methods

Name	Description
deleteCertificateFromStore(successCB, [errorCB], certificateKey) on page 137	Delete a cached certificate from the keychain.
generateODataHttpClient() on page 138	Generates an OData client that uses the AuthProxy plugin to make requests.
get(url, header, successCB, errorCB, [user], [password], [timeout], [certSource]) on page 139	Send an HTTP(S) GET request to a remote server.

sendRequest(method, url, header, requestBody, successCB, errorCB, [user], [password], [timeout], [certSource]) on page 141	Send an HTTP(S) request to a remote server.

Type Definitions

Name	Description
deleteCertificateSuccessCallback on page 143	Callback function that is invoked upon successfully deleting a certificate from the store.
errorCallback(errorObject) on page 143	Callback function that is invoked in case of an error.
successCallback(serverResponse) on page 144	Callback function that is invoked upon a response from the server.

Source

authproxy.js, line 27 on page 146.

sap.AuthProxy.CertificateFromFile class

Create certificate source description object for a certificate from a keystore file.

The keystore file must be of type PKCS12 (usually a .p12 extention) since that is the only certificate file type that can contain a private key (a private key is needed to authenticate the client to the server). You might want to use this method if you know the desired certificate resides in a file on the filesystem.

Syntax

new `CertificateFromFile`(*Path*, *Password*, *CertificateKey*)

Parameters

Name	Type	Description

Path	string	The Path of the keystore file. For iOS clients, it first tries to load the relative file path from the application's Documents folder. If it fails, it then tries to load the file path from application's main bundle. In addition, before trying to load the certificate from the file system, the iOS client first checks whether the specified certificate key already exists in the key store. If it does, it loads the existing certificate from key store, instead of loading the certificate from file system. For Android clients, the filepath is first treated as an absolute path. If the certificate is not found, then the filepath is treated as relative to the root of the sdcard.
Password	string	The password of the keystore.
CertificateKey	string	A unique key (aka: alias) that is used to locate the certificate.

Example

```
// Create the certificate source description object.
var fileCert = new sap.AuthProxy.CertificateFromFile("directory/
certificateName.p12", "certificatePassword", "certificateKey");
// callbacks
var successCB = function(serverResponse){
    alert("Status: " + JSON.stringify(serverResponse.status));
    alert("Headers: " + JSON.stringify(serverResponse.headers));
    alert("Response: " + JSON.stringify(serverResponse.response));
}
var errorCB = function(errorObject){
    alert("Error making request: " + JSON.stringify(errorObject));
}
// Make the request with the certificate source description object.
sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS
IS THE BODY", successCB, errorCB, null, null, 0, fileCert);
```

Source

authproxy.js, line 224 on page 154.

sap.AuthProxy.CertificateFromLogonManager class

Create a certificate source description object for certificates from logon manager.

Using the resulting certificate source description object on subsequent calls to AuthProxy.sendRequest or AuthProxy.get will cause AuthProxy to retrieve a certificate from Logon Manager to use for client authentication. The appID parameter is used to indicate which application's certificate to use.

Note that to use a certificate from Logon Manager, the application must have already registered with the server using a certificate from Afaria.

Syntax

new `CertificateFromLogonManager`(*appID*)

Parameters

Name	Type	Description
appID	string	application identifier

Example

```
// Create the certificate source description object.
var logonCert = new
sap.AuthProxy.CertificateFromLogonManager("applicationID");
// callbacks
var successCB = function(serverResponse){
    alert("Status: " + JSON.stringify(serverResponse.status));
    alert("Headers: " + JSON.stringify(serverResponse.headers));
    alert("Response: " + JSON.stringify(serverResponse.response));
}
var errorCB = function(errorObject){
    alert("Error making request: " + JSON.stringify(errorObject));
}
// Make the request with the certificate source description object.
sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS
IS THE BODY", successCB, errorCB, null, null, 0, logonCert);
```

Source

authproxy.js, line 281 on page 156.

sap.AuthProxy.CertificateFromStore class

Create a certificate source description object for certificates from the system keystore.

You might want to use a certificate from the system keystore if you know the user's device will have the desired certificate installed on it.

On Android, sending a request with a certificate from the system store results in UI being shown for the user to pick the certificate to use (the certificate with the alias matching the given CertificateKey is pre-selected).

Syntax

new `CertificateFromStore`(*CertificateKey*)

Parameters

Name	Type	Description
CertificateKey	string	A unique key (aka: alias) that is used to locate the certificate.

Example

```
// Create the certificate source description object.
var systemCert = new
sap.AuthProxy.CertificateFromStore("certificatekey");
// callbacks
var successCB = function(serverResponse){
    alert("Status: " + JSON.stringify(serverResponse.status));
    alert("Headers: " + JSON.stringify(serverResponse.headers));
    alert("Response: " + JSON.stringify(serverResponse.response));
}
var errorCB = function(errorObject){
    alert("Error making request: " + JSON.stringify(errorObject));
}
// Make the request with the certificate source description object.
sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS
IS THE BODY", successCB, errorCB, null, null, 0, systemCert);
```

Source

authproxy.js, line 253 on page 155.

ERR_CERTIFICATE_ALIAS_NOT_FOUND member

Constant indicating the certificate with the given alias could not be found.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax

<constant> `ERR_CERTIFICATE_ALIAS_NOT_FOUND` : number

Source
authproxy.js, line 90 on page 149.

ERR_CERTIFICATE_FILE_NOT_EXIST member
Constant indicating the certificate file could not be found.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_CERTIFICATE_FILE_NOT_EXIST` : number

Source
authproxy.js, line 98 on page 149.

ERR_CERTIFICATE_INVALID_FILE_FORMAT member
Constant indicating incorrect certificate file format.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_CERTIFICATE_INVALID_FILE_FORMAT` : number

Source
authproxy.js, line 106 on page 149.

ERR_CLIENT_CERTIFICATE_VALIDATION member
Constant indicating the provided certificate failed validation on the server side.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_CLIENT_CERTIFICATE_VALIDATION` : number

Source
authproxy.js, line 122 on page 150.

ERR_FILE_CERTIFICATE_SOURCE_UNSUPPORTED member
Constant indicating the certificate from file is not supported on the current platform.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_FILE_CERTIFICATE_SOURCE_UNSUPPORTED` : number

Source
authproxy.js, line 74 on page 148.

ERR_GET_CERTIFICATE_FAILED member
Constant indicating failure in getting the certificate.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_GET_CERTIFICATE_FAILED` : number

Source
authproxy.js, line 114 on page 149.

ERR_HTTP_TIMEOUT member
Constant indicating timeout error while connecting to the server.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_HTTP_TIMEOUT` : number

Source
authproxy.js, line 164 on page 151.

ERR_INVALID_PARAMETER_VALUE member
Constant indicating the operation failed due to an invalid parameter (for example, a string was passed where a number was required).

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_INVALID_PARAMETER_VALUE` : number

Source
authproxy.js, line 48 on page 147.

ERR_LOGON_MANAGER_CERTIFICATE_METHOD_NOT_AVAILABLE member
Constant indicating the logon manager certifciate method is not available.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_LOGON_MANAGER_CERTIFICATE_METHOD_NOT_AVAILABLE` : number

Source
authproxy.js, line 156 on page 151.

ERR_LOGON_MANAGER_CORE_NOT_AVAILABLE member

Constant indicating the logon manager core library is not available.

Getting this error code means you tried to use Logon plugin features (for example, a certificate from Logon) without adding the Logon plugin to the app. A possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_LOGON_MANAGER_CORE_NOT_AVAILABLE` : number

Source
authproxy.js, line 148 on page 151.

ERR_MISSING_PARAMETER member

Constant indicating the operation failed because of a missing parameter.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_MISSING_PARAMETER` : number

Source
authproxy.js, line 56 on page 147.

ERR_NO_SUCH_ACTION member

Constant indicating there is no such Cordova action for the current service.

When a Cordova plugin calls into native code it specifies an action to perform. If the action provided by the JavaScript is unknown to the native code this error occurs. This error should not occur as long as authproxy.js is unmodified. Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_NO_SUCH_ACTION` : number

Source
authproxy.js, line 66 on page 148.

ERR_SERVER_CERTIFICATE_VALIDATION member
Constant indicating the server certificate failed validation on the client side.

This is likely because the server certificate is self-signed, or not signed by a well-known certificate authority. This constant is used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_SERVER_CERTIFICATE_VALIDATION` : number

Source
authproxy.js, line 131 on page 150.

ERR_SERVER_REQUEST_FAILED member
Constant indicating the server request failed.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_SERVER_REQUEST_FAILED` : number

Source
authproxy.js, line 139 on page 150.

ERR_SYSTEM_CERTIFICATE_SOURCE_UNSUPPORTED member
Constant indicating the certificate from the system keystore is not supported on the current platform.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_SYSTEM_CERTIFICATE_SOURCE_UNSUPPORTED` : number

Source
authproxy.js, line 82 on page 148.

ERR_UNKNOWN member
Constant indicating the operation failed with unknown error.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 143.

Syntax
<constant> `ERR_UNKNOWN` : number

Source
authproxy.js, line 40 on page 147.

deleteCertificateFromStore(successCB, [errorCB], certificateKey) method

Delete a cached certificate from the keychain.

iOS clients always checks the cached certificate first to see if it is available before loading the certificate from the file system. If the cached certificate is no longer valid, use this method to delete it from the keychain.

Only supported on iOS platform, NOT Android.

Syntax
`deleteCertificateFromStore(` *successCB*, [*errorCB*], *certificateKey*)

Parameters

Name	Type	Argument	Description
successCB	*sap.AuthProxy~deleteCertificateSuccessCallback* on page 143		Callback method upon success.
errorCB	*sap.AuthProxy~errorCallback* on page 143	(optional)	Callback method upon failure.
certificateKey	string		The key of the certificate to be deleted.

Example

```
var successCB = function(){
    alert("certificate successfully deleted.");
}
var errorCB = function(error){
    alert("error deleting certificate: " + JSON.stringify(error));
}
sap.AuthProxy.deleteCertificateFromStore(successCB, errorCB,
"certificateKeyToDelete");
```

Source
authproxy.js, line 636 on page 170.

generateODataHttpClient() method

Generates an OData client that uses the AuthProxy plugin to make requests.

This is useful if you are using Datajs, but want to make use of the certificate features of AuthProxy. Datajs is a javascript library useful for accessing OData services. Datajs has a concept of an HttpClient, which does the work of making the request. This function generates an HttpClient that you can specify to Datajs so you can provide client certificates for requests. If you want to use the generated HTTP client for all future Datajs requests, you can do that by setting the OData.defaultHttpClient property to the return value of this function. Once that is done, then doing OData stuff with Datajs is almost exactly the same, but you can add a certificateSource to a request.

Syntax

`generateODataHttpClient`()

Example

```
OData.defaultHttpClient = sap.AuthProxy.generateODataHttpClient();

// Using a certificate from file, for example.
fileCert = new sap.AuthProxy.CertificateFromFile("mnt/sdcard/
cert.p12", "password", "certKey");

// This is the same request object you would have created if you were
just using Datajs, but now
// you can add the extra 'certificateSource' property.
var createRequest = {
    requestUri: "http://www.example.com/stuff/etc/example.svc",
    certificateSource : fileCert,
    user : "username",
    password : "password",
    method : "POST",
    data:
    {
         Description: "Created Record",
         CategoryName: "Created Category"
    }
}

// Use Datajs to send the request.
OData.request( createRequest, successCallback, failureCallback );
```

Source

authproxy.js, line 733 on page 173.

get(url, header, successCB, errorCB, [user], [password], [timeout], [certSource]) method

Send an HTTP(S) GET request to a remote server.

This is a convenience function that simply calls *sap.AuthProxy#sendRequest* on page 141 with "GET" as the method and null for the request body. All given parameters are passed as-is to sap.AuthProxy.sendRequest. The success callback is invoked upon any response from the server. Even responses not generally considered to be successful (such as 404 or 500 status codes) will result in the success callback being invoked. The error callback is reserved for problems that prevent the AuthProxy from creating the request or contacting the server. It is, therefore, important to always check the status property on the object given to the success callback.

Syntax

`get(` *url*, *header*, *successCB*, *errorCB*, [*user*], [*password*], [*timeout*], [*certSource*]) → {function}

Parameters

Name	Type	Argument	Description
url	string		The URL against which to make the request.
header	Object		HTTP header to send to the server.This is an Object. Can be null.
successCB	*sap.AuthProxy~successCallback* on page 144		Callback method invoked upon a response from the server.
errorCB	*sap.AuthProxy~errorCallback* on page 143		Callback method invoked in case of failure.
user	string	(optional)	User ID for basic authentication.
password	string	(optional)	User password for basic authentication.
timeout	number	(optional)	Timeout setting in seconds.Default value (0) means there is no timeout.

certSource	Object	(optional)	Certificate description object.It can be one of *sap.AuthProxy#CertificateFromFile* on page 129, *sap.AuthProxy#CertificateFromStore* on page 132, or *sap.AuthProxy#CertificateFromLogonManager* on page 131.

Returns

A JavaScript function object to abort the operation. Calling the abort function results in neither the success or error callback being invoked for the original request (excepting the case where the success or error callback was invoked before calling the abort functino). Note that the request itself cannot be unsent, and the server will still receive the request - the JavaScript will just not know the results of that request.

Type:

function

Example

```
var successCB = function(serverResponse){
    alert("Status: " + JSON.stringify(serverResponse.status));
    alert("Headers: " + JSON.stringify(serverResponse.headers));
    if (serverResponse.responseText){
        alert("Response: " +
JSON.stringify(serverResponse.responseText));
    }
}
var errorCB = function(errorObject){
    alert("Error making request: " + JSON.stringify(errorObject));
}
// To send a GET request to server, call the method
var abortFunction = sap.AuthProxy.get("http://www.example.com",
null, successCB, errorCB);
// An example of aborting the request
abortFunction();
// To send a GET request to the server with headers, call the method
sap.AuthProxy.get("http://www.example.com", {HeaderName : "Header
value"}, successCB, errorCB);
// To send a GET request to the server with basic authentication,
call the method
sap.AuthProxy.get("https://www.example.com", headers, successCB,
errorCB, "username", "password");
// To send a GET request to the server with mutual authentication,
call the method
```

```
sap.AuthProxy.get("https://www.example.com", headers, successCB,
errorCB, null, null, 0,
    new sap.AuthProxy.CertificateFromLogonManager("theAppId"));
```

Source

authproxy.js, line 616 on page 169.

sendRequest(method, url, header, requestBody, successCB, errorCB, [user], [password], [timeout], [certSource]) method

Send an HTTP(S) request to a remote server.

This function is the centerpiece of the AuthProxy plugin. It will handle mutual authentication if a certificate source is provided. The success callback is invoked upon any response from the server. Even responses not generally considered to be successful (such as 404 or 500 status codes) will result in the success callback being invoked. The error callback is reserved for problems that prevent the AuthProxy from creating the request or contacting the server. It is therefore important to always check the status property on the object given to the success callback.

Syntax

`sendRequest(` *method*, *url*, *header*, *requestBody*, *successCB*, *errorCB*, [*user*], [*password*], [*timeout*], [*certSource*]) → {function}

Parameters

Name	Type	Argument	Description
method	string		Standard HTTP request method name.
url	string		The HTTP URL with format http(s):// [user:password]@hostname[:port]/path.
header	Object		HTTP header to send to the server.This is an Object. Can be null.
requestBody	string		Data to send to the server with the request.Can be null.
successCB	*sap.AuthProxy~successCallback* on page 144		Callback method invoked upon a response from the server.

errorCB	*sap.AuthProxy~errorCallback* on page 143		Callback method invoked in case of failure.
user	string	(optional)	User ID for basic authentication.
password	string	(optional)	User password for basic authentication.
timeout	number	(optional)	Timeout setting in seconds.Default value (0) means there is no timeout.
certSource	Object	(optional)	Certificate description object.It can be one of *sap.AuthProxy#CertificateFromFile* on page 129, *sap.AuthProxy#CertificateFromStore* on page 132, or *sap.AuthProxy#CertificateFromLogonManager* on page 131.

Returns

A JavaScript function object to abort the operation. Calling the abort function results in neither the success or error callback being invoked for the original request (excepting the case where the success or error callback was invoked before calling the abort function). Note that the request itself cannot be unsent, and the server will still receive the request - the JavaScript will just not know the results of that request.

Type:

function

Example

```
// callbacks
var successCB = function(serverResponse){
    alert("Status: " + JSON.stringify(serverResponse.status));
    alert("Headers: " + JSON.stringify(serverResponse.headers));
    alert("Response: " + JSON.stringify(serverResponse.response));
}
var errorCB = function(errorObject){
    alert("Error making request: " + JSON.stringify(errorObject));
}
// To send a post request to the server, call the method
```

```
var abortFunction = sap.AuthProxy.sendRequest("POST", "http://
www.google.com", null, "THIS IS THE BODY", successCB, errorCB);
// An example of aborting the request
abortFunction();

// To send a post request to the server with headers, call the method
sap.AuthProxy.sendRequest("POST", url, {HeaderName : "Header
value"}, "THIS IS THE BODY", successCB, errorCB);

// To send a post request to the server with basic authentication,
call the method
sap.AuthProxy.sendRequest("POST", url, headers, "THIS IS THE BODY",
successCB, errorCB, "username", "password");

// To send a post request to the server with mutual authentication,
call the method
sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS
IS THE BODY", successCB, errorCB, null,
    null, 0, new
sap.AuthProxy.CertificateFromLogonManager("theAppId"));
```

Source
authproxy.js, line 466 on page 163.

deleteCertificateSuccessCallback type
Callback function that is invoked upon successfully deleting a certificate from the store.

Syntax
`deleteCertificateSuccessCallback`()

Source
authproxy.js, line 819 on page 176.

errorCallback(errorObject) type
Callback function that is invoked in case of an error.

Syntax
`errorCallback`(*errorObject*)

Parameters

Name	Type	Description

errorObject	Object	An object containing two properties: 'errorCode' and 'description.' The 'errorCode' property corresponds to one of the *sap.AuthProxy* on page 126 constants. The 'description' property is a string with more detailed information of what went wrong.

Example

```
function errorCallback(errCode) {
   //Set the default error message. Used if an invalid code is passed
to the
   //function (just in case) but also to cover the
   //sap.AuthProxy.ERR_UNKNOWN case as well.
   var msg = "Unkown Error";
   switch (errCode) {
      case sap.AuthProxy.ERR_INVALID_PARAMETER_VALUE:
         msg = "Invalid parameter passed to method";
         break;
      case sap.AuthProxy.ERR_MISSING_PARAMETER:
         msg = "A required parameter was missing";
         break;
      case sap.AuthProxy.ERR_HTTP_TIMEOUT:
         msg = "The request timed out";
         break;
   };
   //Write the error to the log
   console.error(msg);
   //Let the user know what happened
   navigator.notification.alert(msg, null, "AuthProxy Error", "OK");
};
```

Source

authproxy.js, line 815 on page 176.

successCallback(serverResponse) type

Callback function that is invoked upon a response from the server.

Syntax

`successCallback(` *serverResponse*)

Parameters

Name	Type	Description

serverResponse	Object	An object containing the response from the server.Contains a 'headers' property, a 'status' property, and a 'responseText' property. 'headers' is an object containing all the headers in the response. 'status' is an integer corresponding to the HTTP status code of the response. It is important to check the status of the response, since **this success callback is invoked upon any response from the server** - including responses that are not normally thought of as successes (for example, the status code could be 404 or 500). 'responseText' is a string containing the body of the response.

Source
authproxy.js, line 817 on page 176.

Source code

authproxy.js

```
// ${project.version}
var exec = require('cordova/exec');

/**
 * The AuthProxy plugin provides the ability to make HTTPS requests with mutual authentication.<br/>
 * <br/>
 * The regular XMLHttpRequest does not
 * support mutual authentication.  The AuthProxy plugin allows you to specify a certificate to include in an HTTPS request
 * to identify the client to the server.  This allows the server to verify the identity of the client.  An example of where you
 * might need mutual authentication is the onboarding process to register with an application, or, to access an
 * OData producer. This occurs mostly in Business to Business (B2B) applications. This is different from most business to
 * consumer (B2C) Web sites, where it is only the server that authenticates itself to the client with a certificate.<br/>
 * <br/>
 * <b>Adding and Removing the AuthProxy Plugin</b><br/>
 * Add or remove the AuthProxy plugin using the
 * <a href="http://cordova.apache.org/docs/en/edge/guide_cli_index.md.html#The%20Command-line%20Interface">Cordova CLI</a>.<br/>
 * <br/>
 * To add the AuthProxy plugin to your project, use the following command:<br/>
 * cordova plugin add <path to directory containing Kapsel plugins>\authproxy<br/>
 * <br/>
 * To remove the AuthProxy plugin from your project, use the following command:<br/>
 * cordova plugin rm com.sap.mp.cordova.plugins.authproxy
 * @namespace
 * @alias AuthProxy
 * @memberof sap
 */
var AuthProxy = function () {};


/**
 * Constant definitions for registration methods
 */

```

```
    /**
     * Constant indicating the operation failed with unknown error. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_UNKNOWN = -1;

    /**
    * Constant indicating the operation failed due to an invalid parameter (for example, a string was passed where a number was
     * required). Used as a possible value for the errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE = -2;

    /**
     * Constant indicating the operation failed because of a missing parameter. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_MISSING_PARAMETER = -3;

    /**
    * Constant indicating there is no such Cordova action for the current service.  When a Cordova plugin calls into native
     * code it specifies an action to perform.  If the action provided by the JavaScript is unknown to the native code this
     * error occurs.  This error should not occur as long as authproxy.js is unmodified. Used as a possible
     * value for the errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_NO_SUCH_ACTION = -100;

    /**
     * Constant indicating the certificate from file is not supported on the current platform. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_FILE_CERTIFICATE_SOURCE_UNSUPPORTED = -101;

    /**
     * Constant indicating the certificate from the system keystore is not supported on the current platform. Used as a possible value
     * for the errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
AuthProxy.prototype.ERR_SYSTEM_CERTIFICATE_SOURCE_UNSUPPORTED = -102;

    /**
     * Constant indicating the certificate with the given alias could not be found. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_CERTIFICATE_ALIAS_NOT_FOUND = -104;

    /**
     * Constant indicating the certificate file could not be found. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_CERTIFICATE_FILE_NOT_EXIST = -105;

    /**
     * Constant indicating incorrect certificate file format. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_CERTIFICATE_INVALID_FILE_FORMAT = -106;

    /**
     * Constant indicating failure in getting the certificate. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_GET_CERTIFICATE_FAILED = -107;

    /**
     * Constant indicating the provided certificate failed validation on the server side. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_CLIENT_CERTIFICATE_VALIDATION = -108;

    /**
     * Constant indicating the server certificate failed validation on the client side.  This is likely because the server certificate
    * is self-signed, or not signed by a well-known certificate authority.  This constant is used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_SERVER_CERTIFICATE_VALIDATION = -109;

    /**
     * Constant indicating the server request failed. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_SERVER_REQUEST_FAILED = -110;

    /**
    * Constant indicating the Logon Manager core library is not available.  Getting this error code means you tried
     * to use Logon plugin features (for example, a certificate from Logon) without adding the Logon plugin to the app.
     * A possible value for the errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_LOGON_MANAGER_CORE_NOT_AVAILABLE = -111;

    /**
    * Constant indicating the Logon Manager certifciate method is not available. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
AuthProxy.prototype.ERR_LOGON_MANAGER_CERTIFICATE_METHOD_NOT_AVAILABLE = -112;

    /**
     * Constant indicating timeout error while connecting to the server. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */
    AuthProxy.prototype.ERR_HTTP_TIMEOUT = -120;

    /**
     * Constant indicating timeout error while connecting to the server. Used as a possible value for the
     * errorCode in {@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type number
     */

    /**
    * Constant indicating a missing required parameter message. Used as a possible value for the description
     * in (@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type string
     * @private
     */
    AuthProxy.prototype.MSG_MISSING_PARAMETER = "Missing a required parameter: ";

    /**
    * Constant indicating invalid parameter value message.  Used as a possible value for the description
     * in (@link sap.AuthProxy~errorCallback}.
     * @constant
     * @type string
     * @private
     */
    AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE = "Invalid Parameter Value for parameter: ";

    /**
     * Create certificate source description object for a certificate from a keystore file.  The keystore file must be of type PKCS12
     * (usually a .p12 extension) since that is the only certificate file type that can contain a private key (a private key is needed
    * to authenticate the client to the server).  You might want to use this method if you know the desired certificate resides in a
     * file on the filesystem.
     * @class
    * @param {string} Path The Path of the keystore file.<br/>For iOS clients, it first tries to load the
    *                 relative file path from the application's Documents folder. If it fails, it then tries
     *               to load the file path from the application's main bundle. In addition, before trying
     *               to load the certificate from the file system, the iOS client first checks whether the
     *               specified certificate key already exists in the key store. If it does, it loads
     *               the existing certificate from the key store, instead of loading the certificate from the
     *               file system.<br/>
     *               For Android clients, the file path is first treated as an absolute path. If the certificate
     *               is not found, the file path is treated as relative to the root of the SD card.
     * @param {string} Password The password of the keystore.
     * @param {string} CertificateKey A unique key (alias) that is used to locate the certificate.
     * @example
     * // Create the certificate source description object.
     * var fileCert = new sap.AuthProxy.CertificateFromFile("directory/certificateName.p12", "certificatePassword", "certificateKey");
     * // callbacks
     * var successCB = function(serverResponse){
     *     alert("Status: " + JSON.stringify(serverResponse.status));
     *     alert("Headers: " + JSON.stringify(serverResponse.headers));
     *     alert("Response: " + JSON.stringify(serverResponse.response));
     * }
     * var errorCB = function(errorObject){
     *     alert("Error making request: " + JSON.stringify(errorObject));
     * }
     * // Make the request with the certificate source description object.
     * sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS IS THE BODY", successCB, errorCB, null, null, 0, fileCert);
     *
     */
    AuthProxy.prototype.CertificateFromFile = function (Path, Password, CertificateKey) {
        this.Source = "FILE";
        this.Path = Path;
        this.Password = Password;
        this.CertificateKey = CertificateKey;
    };

    /**
     * Create a certificate source description object for certificates from the system keystore.  You might want to use a certificate
     * from the system keystore if you know the user's device will have the desired certificate installed on it.<br/>
     * On Android, sending a request with a certificate from the system store results in UI being shown, where the user can pick
     * the certificate to use (the certificate with the alias matching the given CertificateKey is pre-selected).
     * @class
     * @param {string} CertificateKey A unique key (alias) that is used to locate the certificate.
     * @example
     * // Create the certificate source description object.
     * var systemCert = new sap.AuthProxy.CertificateFromStore("certificatekey");
     * // callbacks
     * var successCB = function(serverResponse){
     *     alert("Status: " + JSON.stringify(serverResponse.status));
     *     alert("Headers: " + JSON.stringify(serverResponse.headers));
     *     alert("Response: " + JSON.stringify(serverResponse.response));
     * }
     * var errorCB = function(errorObject){
```

```
     *     alert("Error making request: " + JSON.stringify(errorObject));
     * }
    * // Make the request with the certificate source description object.
     * sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS IS THE BODY", successCB, errorCB, null, null, 0, systemCert);
     */
    AuthProxy.prototype.CertificateFromStore = function (CertificateKey) {
        this.Source = "SYSTEM";
        this.CertificateKey = CertificateKey;
    };


    /**
     * Create a certificate source description object for certificates from Logon Manager.  Using the resulting certificate source description
     * object on subsequent calls to AuthProxy.sendRequest or AuthProxy.get causes AuthProxy to retrieve a certificate from Logon Manager
     * to use for client authentication. The appID parameter is used to indicate which application's certificate to use.<br/>
     * Note: To use a certificate from Logon Manager, the application must have already registered with the server using a certificate from Afaria.
     * @class
     * @param {string} appID application identifier
     * @example
     * // Create the certificate source description object.
     * var logonCert = new sap.AuthProxy.CertificateFromLogonManager("applicationID");
     * // callbacks
     * var successCB = function(serverResponse){
     *     alert("Status: " + JSON.stringify(serverResponse.status));
```

```
 *     alert("Headers: " + JSON.stringify(serverResponse.headers));
 *     alert("Response: " + JSON.stringify(serverResponse.response));
 * }
 * var errorCB = function(errorObject){
 *     alert("Error making request: " + JSON.stringify(errorObject));
 * }
 * // Make the request with the certificate source description object.
 * sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS IS THE BODY", successCB, errorCB, null, null, 0, logonCert);
 */
AuthProxy.prototype.CertificateFromLogonManager = function (appID) {
    this.Source = "LOGON";
    this.AppID = appID;
};


/**
 * Verifies that a certificate source description object (created with {@link sap.AuthProxy#CertificateFromFile},
 * {@link sap.AuthProxy#CertificateFromStore}, or {@link sap.AuthProxy#CertificateFromLogonManager}) has all the required fields and that the values
 * for those fields are the correct type.  This function verifies only the certificate description object, not the certificate itself.  So, for example,
 * if the certificate source description object was created with {@link sap.AuthProxy#CertificateFromFile} and has a string for the file path and a
 * string for the key/alias, <b>this function considers it valid even if no certificate actually exists on the file system</b>. If the certificate
 * source description object is valid but the certificate itself is not, then an error occurs during the call to {@link sap.AuthProxy#get} or
```

```
     * {@link sap.AuthProxy#sendRequest}.
     * @param {object} certSource The certificate source object.
     * @param {sap.AuthProxy~errorCallback} errorCB The error callback invoked if the certificate source is not valid.  Will have an object with 'errorCode'
     * and 'description' properties.
     * @example
     * var notValidCert = {};
     * var errorCallback = function(error){
     *     alert("certificate not valid!\nError code: " + error.errorCode + "\ndescription: " + error.description);
     * }
     * var isCertValid = sap.AuthProxy.validateCertSource(notValidCert, errorCallback);
     * if( isCertValid ){
     *     // do stuff with the valid certificate source description object
     * } else {
     *     // at this point we know the cert is not valid, and the error callback is invoked with extra information.
     * }
     *
     *
     * Developers are not expected to call this function.
     * @private
     */
    AuthProxy.prototype.validateCertSource = function (certSource, errorCB) {
        if (!certSource) {
            // The certificate is not present, so just ignore it.
            return true;
        }

        // errorCB required.
        // First check this one. We may need it to return errors
        if (errorCB && (typeof errorCB !== "function")) {
            console.log("AuthProxy Error: errorCB is not a function");
            return false;
        }

        try {
            // First check whether it is an object
            if (typeof certSource !== "object") {
                errorCB({
                    errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                    description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certSource"
                });
                return false;
            }

            if (certSource.Source === "FILE") {
                if (!certSource.Path) {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_MISSING_PARAMETER + "keystore path"
                    });
                    return false;
                }

                if (typeof certSource.Path !== "string") {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "keystore path"
                    });
                    return false;
                }

                if (!certSource.CertificateKey) {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_MISSING_PARAMETER + "certificate key"
                    });
                    return false;
                }

                if (typeof certSource.CertificateKey !== "string") {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certificate key"
                    });
                    return false;
                }
            } else if (certSource.Source === "SYSTEM") {
                if (!certSource.CertificateKey) {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_MISSING_PARAMETER + "certificate key"
                    });
                    return false;
                }

              if (typeof certSource.CertificateKey !== "string") {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certificate key"
                    });
                    return false;
                }
            } else if (certSource.Source === "LOGON") {
                if (!certSource.AppID) {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_MISSING_PARAMETER + "AppID"
                    });
                    return false;
                }

                if (typeof certSource.AppID !== "string") {
                    errorCB({
                        errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                        description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "AppID"
                    });
                    return false;
                }
            } else {
                errorCB({
                    errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                    description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certSource"
                });
                return false;
            }

            return true;
        } catch (ex) {
            errorCB({
                errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certSource"
            });
        }
    };


    /**
     * Send an HTTP(S) request to a remote server.  This function is the centerpiece of the AuthProxy plugin.  It handles
     * mutual authentication if a certificate source is provided.
     * The success callback is invoked upon any response from the server, even responses not generally considered to be
     * successful (such as 404 or 500 status codes) result in the success callback being invoked.  The error callback
     * is reserved for problems that prevent the AuthProxy from creating the request or contacting the server. It is, therefore,
     * important to always check the status property on the object given to the success callback.
     * @param {string} method Standard HTTP request method name.
     * @param {string} url The HTTP URL with format http(s)://[user:password]@hostname[:port]/path.
     * @param {Object} header HTTP header to send to the server. This is an Object. Can be null.
     * @param {string} requestBody Data to send to the server with the request. Can be null.
     * @param {sap.AuthProxy~successCallback} successCB Callback method invoked upon a response from the server.
     * @param {sap.AuthProxy~errorCallback} errorCB Callback method invoked in case of failure.
     * @param {string} [user] User ID for basic authentication.
     * @param {string} [password] User password for basic authentication.
     * @param {number} [timeout] Timeout setting in seconds. Default value (0) means there is no timeout.
     * @param {Object} [certSource] Certificate description object. It can be one of {@link sap.AuthProxy#CertificateFromFile},
     * {@link sap.AuthProxy#CertificateFromStore}, or {@link sap.AuthProxy#CertificateFromLogonManager}.
     * @return {function} A JavaScript function object to abort the operation.  Calling the abort function results in neither the success or error
     * callback being invoked for the original request (excepting the case where the success or error callback was invoked before calling the
     * abort function).  Note: The request itself cannot be unsent, and the server will still receive the request, but the JavaScript will
     * not know the results of that request.
     * @example
     * // callbacks
     * var successCB = function(serverResponse){
     *     alert("Status: " + JSON.stringify(serverResponse.status));
     *     alert("Headers: " + JSON.stringify(serverResponse.headers));
     *     alert("Response: " + JSON.stringify(serverResponse.response));
     * }
     * var errorCB = function(errorObject){
     *     alert("Error making request: " + JSON.stringify(errorObject));
     * }
     * // To send a post request to the server, call the method
     * var abortFunction = sap.AuthProxy.sendRequest("POST", "http://www.google.com", null, "THIS IS THE BODY", successCB, errorCB);
     * // An example of aborting the request
     * abortFunction();
     *
     * // To send a post request to the server with headers, call the method
     * sap.AuthProxy.sendRequest("POST", url, {HeaderName : "Header value"}, "THIS IS THE BODY", successCB, errorCB);
     *
     * // To send a post request to the server with basic authentication, call the method
     * sap.AuthProxy.sendRequest("POST", url, headers, "THIS IS THE BODY", successCB, errorCB, "username", "password");
     *
     * // To send a post request to the server with mutual authentication, call the method
     * sap.AuthProxy.sendRequest("POST", "https://hostname", headers, "THIS IS THE BODY", successCB, errorCB, null,
     *     null, 0, new sap.AuthProxy.CertificateFromLogonManager("theAppId"));
     */
    AuthProxy.prototype.sendRequest = function (method, url, header, requestBody, successCB, errorCB, user, password, timeout, certSource) {

        // errorCB required.
        // First check this one. We may need it to return errors
        if (!errorCB || (typeof errorCB !== "function")) {
            console.log("AuthProxy Error: errorCB is not a function");
            // if error callback is invalid, throw an exception to notify the caller
            throw new Error("AuthProxy Error: errorCB is not a function");
        }

        // method required
        if (!method) {
            console.log("AuthProxy Error: method is required");
            errorCB({
                errorCode: AuthProxy.prototype.ERR_MISSING_PARAMETER,
                description: AuthProxy.prototype.MSG_MISSING_PARAMETER + "method"
            });
            return;
        }


        // We only support GET, POST, HEAD, PUT, DELETE method
        if (method !== "GET" && method !== "POST" && method !== "HEAD" && method !== "PUT" && method !== "DELETE") {
          console.log("Invalid Parameter Value for parameter: " + method);
            errorCB({
                errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "method"
            });
            return;
        }


        // url required
        if (!url) {
            console.log("AuthProxy Error: url is required");
            errorCB({
                errorCode: AuthProxy.prototype.ERR_MISSING_PARAMETER,
                description: AuthProxy.prototype.MSG_MISSING_PARAMETER + "url"
            });
            return;
        }


        // successCB required
        if (!successCB) {
            console.log("AuthProxy Error: successCB is required");
            errorCB({
                errorCode: AuthProxy.prototype.ERR_MISSING_PARAMETER,
                description: AuthProxy.prototype.MSG_MISSING_PARAMETER + "successCB"
            });
            return;
        }


        if (typeof successCB !== "function") {
            console.log("AuthProxy Error: successCB is not a function");
            errorCB({
                errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "successCB"
            });
            return;
        }


        if (user && typeof user !== "string") {
            errorCB({
                errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "user"
            });
            return;
        }


        if (password && typeof password !== "string") {
            errorCB({
                errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "password"
            });
            return;
        }


        if (timeout && typeof timeout !== "number") {
            errorCB({
                errorCode: AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
                description: AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "timeout"
            });
            return;
        }

        if (!this.validateCertSource(certSource, errorCB)) {
            return;
        }


        try {
            var client = new Client(method, url, header, requestBody, successCB, errorCB, user, password, timeout, certSource);
            return client.send();
        } catch (ex) {
            errorCB({
                errorCode: AuthProxy.prototype.ERR_UNKNOWN,
                description: ex.message
            });
        }

    };

    /**
     * Send an HTTP(S) GET request to a remote server.  This is a convenience function that simply calls {@link sap.AuthProxy#sendRequest}
     * with "GET" as the method and null for the request body. All given parameters are passed as-is to sap.AuthProxy.sendRequest.
     * The success callback is invoked upon any response from the server, even responses not generally considered to be
     * successful (such as 404 or 500 status codes) result in the success callback being invoked. The error callback
     * is reserved for problems that prevent the AuthProxy from creating the request or contacting the server.  It is, therefore,
     * important to always check the status property on the object given to the success callback.
     * @param {string} url The URL against which to make the request.
     * @param {Object} header HTTP header to send to the server. This is an Object. Can be null.
     * @param {sap.AuthProxy~successCallback} successCB Callback
method invoked upon a response from the server.
     * @param {sap.AuthProxy~errorCallback} errorCB Callback
method invoked in case of failure.
     * @param {string} [user] User ID for basic authentication.
     * @param {string} [password] User password for basic
authentication.
     * @param {number} [timeout] Timeout setting in seconds.
Default value (0) means there is no timeout.
     * @param {Object} [certSource] Certificate description
object. It can be one of {@link sap.AuthProxy#CertificateFromFile},
     * {@link sap.AuthProxy#CertificateFromStore}, or {@link
sap.AuthProxy#CertificateFromLogonManager}.
     * @return {function} A JavaScript function object to abort
the operation.  Calling the abort function results in neither the
success or error
     * callback being invoked for the original request (excepting
the case where the success or error callback was invoked before
calling the
     * abort function).  Note: The request itself cannot be
unsent, and the server will still receive the request, but the
JavaScript will
     * not know the results of that request.
     * @example
     * var successCB = function(serverResponse){
     *     alert("Status: " +
JSON.stringify(serverResponse.status));
     *     alert("Headers: " +
JSON.stringify(serverResponse.headers));
     *     if (serverResponse.responseText){
     *         alert("Response: " +
JSON.stringify(serverResponse.responseText));
     *     }
     * }
     * var errorCB = function(errorObject){
     *     alert("Error making request: " +
JSON.stringify(errorObject));
     * }
     * // To send a GET request to server, call the method
     * var abortFunction = sap.AuthProxy.get("http://www.example.com", null, successCB, errorCB);
     * // An example of aborting the request
     * abortFunction();
     * // To send a GET request to the server with headers, call the method
     * sap.AuthProxy.get("http://www.example.com", {HeaderName : "Header value"}, successCB, errorCB);
     * // To send a GET request to the server with basic authentication, call the method
     * sap.AuthProxy.get("https://www.example.com", headers, successCB, errorCB, "username", "password");
     * // To send a GET request to the server with mutual authentication, call the method
     * sap.AuthProxy.get("https://www.example.com", headers, successCB, errorCB, null, null, 0,
     *     new sap.AuthProxy.CertificateFromLogonManager("theAppId"));
     */
    AuthProxy.prototype.get = function (url, header, successCB, errorCB, user, password, timeout, certSource) {
        return this.sendRequest("GET", url, header, null, successCB, errorCB, user, password, timeout, certSource);
    };

    /**
     * Delete a cached certificate from the keychain. iOS clients always check the cached certificate first to see if it is available before
     * loading the certificate from the file system. If the cached certificate is no longer valid, use this method to delete it from the keychain.
     * <br/><b>This function is supported only on iOS.</b>
     * @param {sap.AuthProxy~deleteCertificateSuccessCallback} successCB Callback method upon success.
     * @param {sap.AuthProxy~errorCallback} [errorCB] Callback method upon failure.
     * @param {string} certificateKey The key of the certificate to delete.
     * @example
     * var successCB = function(){
     *     alert("certificate successfully deleted.");
     * }
     * var errorCB = function(error){
     *     alert("error deleting certificate: " + JSON.stringify(error));
     * }
     * sap.AuthProxy.deleteCertificateFromStore(successCB, errorCB, "certificateKeyToDelete");
     */
    AuthProxy.prototype.deleteCertificateFromStore = function (successCB, errorCB, certificateKey) {
        cordova.exec(successCB, errorCB, "AuthProxy", "deleteCertificateFromStore", [certificateKey]);
    };

    /**
     * @private
     */
    var Client = function (method, url, header, requestBody, successCB, errorCB, user, password, timeout, certSource) {

        //ios plugin parameter does not support object type, convert Header and CertSource to JSON string
        if (device.platform === "iOS" || (device.platform && device.platform.indexOf("iP") === 0)) {
            if (header) {
                header = JSON.stringify(header);
            }
            if (certSource) {
                certSource = JSON.stringify(certSource);
            }
        }

        this.Method = method;
        this.Url = url;
        this.Header = header;
        this.RequestBody = requestBody;
        this.SuccessCB = successCB;
        this.ErrorCB = errorCB;
        this.User = user;
        this.Password = password;
        this.Timeout = timeout;
        this.CertSource = certSource;
        this.IsAbort = false;

        this.abort = function () {
            this.IsAbort = true;
        };


        this.send = function () {

            var args = [this.Method, this.Url, this.Header,
this.RequestBody, this.User, this.Password, this.Timeout,
this.CertSource];

            var me = this;

            var successCallBack = function (data) {
                if (me.IsAbort === true) {
                    return;
                }

                successCB(data);
            };

            var errorCallBack = function (data) {
                if (me.IsAbort === true) {
                    return;
                }

                errorCB(data);
            };

            exec(successCallBack, errorCallBack, "AuthProxy", "sendRequest", args);

            return this.abort;
        };
    };

    /**
     * Generates an OData client that uses the AuthProxy plugin to make requests.  This is useful if you are using Datajs, but want
     * to make use of the certificate features of AuthProxy. Datajs is a JavaScript library useful for accessing OData services.
     * Datajs has a concept of an HttpClient, which does the work of making the request.  This function generates an HttpClient that
     * you can specify to Datajs so you can provide client certificates for requests.  If you want to use the generated HTTP client
     * for all future Datajs requests, you can do that by setting the OData.defaultHttpClient property to the return value of this
     * function.  Once that is done, then doing OData stuff with Datajs is almost exactly the same, but you can add a
     * certificateSource to a request.
     * @example
     * OData.defaultHttpClient = sap.AuthProxy.generateODataHttpClient();
     *
     * // Using a certificate from file, for example.
     * fileCert = new sap.AuthProxy.CertificateFromFile("mnt/sdcard/cert.p12", "password", "certKey");
     *
     * // This is the same request object you would have created if you were just using Datajs, but now
     * // you can add the extra 'certificateSource' property.
     * var createRequest = {
     *     requestUri: "http://www.example.com/stuff/etc/example.svc",
     *     certificateSource : fileCert,
     *     user : "username",
     *     password : "password",
     *     method : "POST",
     *     data:
     *     {
     *          Description: "Created Record",
     *          CategoryName: "Created Category"
     *     }
     * }
     *
     * // Use Datajs to send the request.
     * OData.request( createRequest, successCallback, failureCallback );
     *
     */
    AuthProxy.prototype.generateODataHttpClient = function () {
        var httpClient = {
            request: function (request, success, error) {
              var url, requestHeaders, requestBody, statusCode, statusText, responseHeaders;
             var responseBody, requestTimeout, requestUserName, requestPassword, requestCertificate;
                var client, result;

            url = request.requestUri;
            requestHeaders = request.headers;
            requestBody = request.body;

            var successCB = function (data) {
                var response = {
                    requestUri: url,
                    statusCode: data.status,
                    statusText: data.status,
                    headers: data.headers,
                    body: (data.responseText ? data.responseText : data.responseBase64)
                };

                if (response.statusCode >= 200 && response.statusCode <= 299) {
                    if (success) {
                        success(response);
                    }
                } else {
                    if (error) {
                        error({
                            message: "HTTP request failed",
                            request: request,
                            response: response
                        });
                    }
                }
            };

            var errorCB = function (data) {
                if (error) {
                    error({
                        message: data
                    });
                }
            };

            if (request.timeoutMS) {
                requestTimeout = request.timeoutMS / 1000;
            }

            if (request.certificateSource) {
                requestCertificate = request.certificateSource;
            }

            if (request.user) {
                requestUserName = request.user;
            }

            if (request.password) {
                requestPassword = request.password;
            }

            client = AuthProxy.prototype.sendRequest(request.method || "GET", url, requestHeaders, requestBody, successCB, errorCB, requestUserName, requestPassword, requestTimeout, requestCertificate);

            result = {};
            result.abort = function () {
                client.abort();

                if (error) {
                    error({
                        message: "Request aborted"
                        });
                    }
                };
                return result;
            }
        };
        return httpClient;
    };

    var AuthProxyPlugin = new AuthProxy();

    module.exports = AuthProxyPlugin;


    /**
     * Callback function that is invoked in case of an error.
     *
     * @callback sap.AuthProxy~errorCallback
     *
     * @param {Object} errorObject An object containing two 
properties: 'errorCode' and 'description.'
    * The 'errorCode' property corresponds to one of the {@link 
sap.AuthProxy} constants.  The 'description'
   * property is a string with more detailed information of what 
went wrong.
     *
     * @example
     * function errorCallback(errCode) {
   *    //Set the default error message. Used if an invalid code 
is passed to the
     *    //function (just in case) but also to cover the
     *    //sap.AuthProxy.ERR_UNKNOWN case as well.
     *    var msg = "Unkown Error";
     *    switch (errCode) {
```

```
     *       case sap.AuthProxy.ERR_INVALID_PARAMETER_VALUE:
     *          msg = "Invalid parameter passed to method";
     *          break;
     *       case sap.AuthProxy.ERR_MISSING_PARAMETER:
     *          msg = "A required parameter was missing";
     *          break;
     *       case sap.AuthProxy.ERR_HTTP_TIMEOUT:
     *          msg = "The request timed out";
     *          break;
     *    };
     *    //Write the error to the log
     *    console.error(msg);
     *    //Let the user know what happened
     *    navigator.notification.alert(msg, null, "AuthProxy Error", "OK");
     * };
     */

    /**
    * Callback function that is invoked upon a response from the server.
     *
     * @callback sap.AuthProxy~successCallback
     *
     * @param {Object} serverResponse An object containing the response from the server.  Contains a 'headers' property,
     * a 'status' property, and a 'responseText' property.<br/>
     * 'headers' is an object containing all the headers in the response.<br/>
     * 'status' is an integer corresponding to the HTTP status code of the response.  It is important to check the status of
     * the response, since <b>this success callback is invoked upon any response from the server</b> - including responses that are
     * not normally thought of as successes (for example, the status code could be 404 or 500).<br/>
     * 'responseText' is a string containing the body of the response.
     */

    /**
     * Callback function that is invoked upon successfully deleting a certificate from the store.
     *
     * @callback sap.AuthProxy~deleteCertificateSuccessCallback
     */
```

Using the Logger Plugin

The Logger plugin includes client-side APIs that you can use for logging the activities of your application.

Logger Plugin Overview

The Logger plugin allows you to log information to trace bugs or other issues in your application for analysis.

Note: To upload log files successfully with the Logger plugin, these conditions must be met:

- In Management Cockpit, the **Log Upload** check box must be selected.
- The `sap.Logger.upload()` must be called.

With the Logger plugin, you can enable an application to write log entries that can then be automatically uploaded to SAP Mobile Platform Server for analysis by using the `sap.Logger.upload()` method. If you add the Settings plugin to your project files, `sap.Logger.upload()` is called with a logon success event (for example, when the application is launched or resumed and logon is successful) so the log file is uploaded automatically. If you do not use the Settings plugin, you can upload log files only by calling the `sap.Logger.upload()` method manually.

You can build in support for logging so that an administrator can remotely set the appropriate log level from SAP Mobile Platform Server. The Kapsel Logger plugin can define each log message with specific levels, such as Debug and Error, which enables you to filter the log message by priority level. The Kapsel Logger plugin mirrors the OData logger library so that it can collect all of the logging data produced by the OData library. The Kapsel plugins use OData libraries in several places so that it can help see and trace the plugins' logging data.

Using the provided `sap.Logger.upload()` method allows you to log events that occur on the device and send them to SAP Mobile Platform Server, where an Administrator can view them and remotely set the appropriate log level to control the amount of information that is written to the log.

This shows the index.html file for a sample app, which has the appID of "com.mycompany.logger" with the server connection information. This information allows the app to register with the appID on SAP Mobile Platform Server. This sample app logs messages with the log level and uploads a log file to SAP Mobile Platform Server. For example, to log messages with DEBUG log level, you can call the `sap.Logger.debug(...)` method. You can also use other methods for logging with other log levels (INFO, WARN and ERROR).

```
<html>
     <head>
        <script type="text/javascript" charset="utf-8"
src="cordova.js"></script>
        <script>
            logonView = null;
            logon = null;
            applicationContext = null;
            function init() {
              var appId = "com.mycompany.logger";  // Change this to
app id on server
             // Optional initial connection context var context = {
                  "serverHost": "server.sap.corp", //Place your SAP
Mobile Platform server 3.0 name here
                   "https": "false",
                   "serverPort": "8080",
                "user": "user", //Place your user name for the OData
Endpoint here
                "password": "xxxxxxx",  //Place your password for the
OData Endpoint here
                   "communicatorId": "REST",
                   "passcode": "password",
                   "unlockPasscode": "password"
               };
            sap.Logon.init(function() { }, function() {alert("Logon
Failed"); }, appId, context, sap.logon.IabUi);
              sap.Logger.setLogLevel(sap.Logger.DEBUG);
}
          function logMessage() {
               var employee = {name: "Dan", location : "Waterloo"};
               console.log("The value of employee is " +
JSON.stringify(employee));
}
          function logMessage2() {
               sap.Logger.debug("Debug log message");
               sap.Logger.info("Info log message");
               sap.Logger.warn("Warn log message");
               sap.Logger.error("Error log message");
}
          function uploadLog() {
               sap.Logger.upload(function() {
                alert("Upload Successful");
               }, function(e) {
                alert("Upload Failed. Status: " + e.statusCode + ", 
Message: " + e.statusMessage);
});
}
          document.addEventListener("deviceready", init, false);
       </script>
</head>
     <body>
          <h1>Logger Sample</h1>
          <button id="log" onclick="logMessage()">Log Message with 
console</button><br>
          <button id="log" onclick="logMessage2()">Log Message with 
Logging Plugin</button><br>
          <button id="upload" onclick="uploadLog()">Upload Log</
button>
     </body>
</html>
```

Setting the Log Level

You can manually set the Kapsel log level in the `init(…)` function by adding code, for example:

```
sap.Logger.setLogLevel(sap.Logger.INFO,
    function(logLevel) {console.log("Log level set");},
    function() {console.log("Failed to set log level");});
```

Log levels are:

- ERROR
- WARN
- INFO
- DEBUG

By default, only error level logs are captured. Use the `setLogLevel` to capture other levels. If the log level is DEBUG, all log level messages are stored. If it is WARN, the uploaded log contains WARN and ERROR messages.

On iOS, if the log level is ERROR, then only ERROR level messages are displayed in the console, even if other log level messages are generated. But if the current log level is DEBUG, INFO, or WARN, all generated log messages, regardless of log level, are displayed in the console.On Android, all generated log messages, regardless of log level, are shown in the Android log cat view (console).

To upload the log to the server, in the `logMessageInfoToSMP(…)` function, enter:

```
sap.Logger.setLogLevel(sap.Logger.INFO,
    function(logLevel) {console.log("Log level set");},
    function() {console.log("Failed to set log level");});
```

Limitations

On Android, the maximum for log entries is 10,000. The oldest 200 log entries are removed if the 10,000 maximum is reached. This applies to both the device and emulator.

On iOS simulators, the Logger plugin may behave in unpredictable ways, as it is intended for use with a device. On iOS devices, there is no explicit maximum for log entries, however, old messages are removed from the device after a time.

Adding the Logger Plugin

Install the Logger plugin using the Cordova command line interface.

Prerequisites

- Set up the development environment.
- Create your Cordova Project.
- Add your OS platforms.

Task

1. Add the Logger plugin by entering the following at the command prompt, or terminal:

   On Windows:

   ```
   cordova -d plugin add <SDK_HOME>\MobileSDK3\KapselSDK
   \plugins\logger
   ```

   On Mac:

   ```
   cordova -d plugin add ~<SDK_HOME>/MobileSDK3/KapselSDK/
   plugins/logger
   ```

   Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).

2. (Optional) To see a list of installed plugins in your Cordova project, open a command prompt or terminal window, navigate to your Cordova project folder, and enter:

   ```
   cordova plugins
   ```

   The Cordova command line interface returns a JSON array showing installed plugins, for example:

   ```
   [ 'org.apache.cordova.core.camera',
   'org.apache.cordova.core.device-motion',
   'org.apache.cordova.core.file' ]
   ```

   In this example, the Cordova project has the Cordova core Camera, Accelerator (device-motion), and File plugins installed.

3. Modify the files in the `www` folder for the project as necessary, then copy them to the platform directories by running:

   ```
   cordova -d prepare android
   cordova -d prepare ios
   ```

4. Use the Android IDE or Xcode to deploy and run the project.

Note: If you are using an iOS device, remember to add the "clienthubEntitlements" to the Keychain Groups in the Entitlement section in Xcode.

Viewing Client Logs

(Applies only to hybrid) Download and view a client log associated with the selected application registration. The developer must have implemented Logger code in the application code, and the application must be registered and collecting data. The log content varies by device type and operating system.

1. From Management Cockpit, select **Registrations** on the Home screen to view application connections. Alternatively, in the **Applications** tab, click the **Registrations** tab.

   Information for up to 200 registered applications appears.
2. Use the search, sorting, and filtering options to locate the registration in which you are interested:
3. Click the red client log icon to display the Client Logging dialog. In some cases, a list of client logs appears.
   a) Click **Enable Log Upload**.
   b) Select the log level in **Log Type**.
   c) Click **Save** to save the modified setting of whether to allow uploading of client logs, and the level at which the client should log.
      The red client log icon turns green.
   d) Click a log file name to download the log and open it in your selected viewer.

Client Logs

(Applies only to hybrid) If client logging has been enabled for a hybrid application and log data is available, you can view the client long for the selected application registration.

Note: The log format varies by device type and operating system. Following are example log excerpts for Android and iOS.

Hybrid Client Log - Android Example

```
1377125811306
Debug Tag
Debug Message
null (com.sap.mp.cordova.plugins.logger.Logger:execute:54)
1893

1377125813056
Info Tag
Info Message
null (com.sap.mp.cordova.plugins.logger.Logger:execute:61)
1893

1377125814165
Warn Tag
Warn Message
null (com.sap.mp.cordova.plugins.logger.Logger:execute:68)
```

```
1893

1377125815157
Error Tag
Error Message
null (com.sap.mp.cordova.plugins.logger.Logger:execute:75)
1893
```

Hybrid Client Log - iOS Example

```

ASLMessageID = 142697
Time = Aug 20, 2013, 4:37:22 PM
TimeNanoSec = 274834000
Level = 3
PID = 4823
UID = 966313393
GID = 1824234391
ReadGID = 80
Host = PALM00545086A
Sender = KAPSEL326
Facility = com.sap.sdmlogger
Message = MAFLogon MCIM is available: NO -[MAFMCIMManager
isAvailable] Line:48 thread:<NSThread: 0x9d57c10>{name = (null), num
= 1}

ASLMessageID = 142696
Time = Aug 20, 2013, 4:37:22 PM
TimeNanoSec = 234589000
Level = 4
PID = 4823
UID = 966313393
GID = 1824234391
ReadUID = 966313393
Host = PALM00545086A
Sender = KAPSEL326
Facility = com.sap.kapsel326
Message = Finished load of: file:///Users/i834381/Library/
Application%20Support/iPhone%20Simulator/6.1/Applications/9EC4E7F3-
C156-476F-850B-56EE001EEAB2/KAPSEL326.app/www/index.html
CFLog Local Time = 2013-08-20 16:37:22.234
CFLog Thread = c07

ASLMessageID = 142695
Time = Aug 20, 2013, 4:37:22 PM
TimeNanoSec = 194786000
Level = 4
PID = 4823
UID = 966313393
GID = 1824234391
ReadUID = 966313393
Host = PALM00545086A
Sender = KAPSEL326
Facility = com.sap.kapsel326
Message = Resetting plugins due to page load.
CFLog Local Time = 2013-08-20 16:37:22.194
CFLog Thread = c07

ASLMessageID = 142694
Time = Aug 20, 2013, 4:37:22 PM
TimeNanoSec = 152145000
Level = 4
PID = 4823
UID = 966313393
GID = 1824234391
ReadUID = 966313393
Host = PALM00545086A
Sender = KAPSEL326
Facility = com.sap.kapsel326
Message = Multi-tasking -> Device: YES, App: YES
CFLog Local Time = 2013-08-20 16:37:22.151
CFLog Thread = c07
```

Testing Logging

The log file is located in `SMP_HOME\Server\log\clientlogs\<application_id>\<application_registration_id>\Log.txt`.

1. Run your project with the Android IDE or Xcode.
2. In Management Cockpit, enable the upload log function.

   Note: For the call to `sap.Logger.upload()` to succeed, the **Log Upload** checkbox on the registration ID in the Management Cockpit must be checked.
3. View the uploaded logs in the Management Cockpit.

Kapsel Logger API Reference

The Kapsel Logger API Reference provides usage information for Logger API classes and methods, as well as provides sample source code.

Logger namespace

The Kapsel Logger plugin provides a Cordova plugin wrapper around the SAP Mobile Platform client logging API.

It has ERROR, WARN, INFO and DEBUG log level and log messages are captured based on the selected log level. Android and iOS logger default log level is ERROR. By default only ERROR level logs are captured. The sap.Logger.setLogLevel() method is used to set other levels. To get log messages for all log levels, you must set the log level to DEBUG. (DEBUG < INFO < WARN < ERROR)

Using the provided sap.Logger.upload() method allows you to upload a log file to SAP Mobile Platform Server, where an Administrator can view them and remotely set the appropriate log level to control the amount of information that is written to the log. When the sap.Logger.upload() method is triggered, a log file (not multiple) is uploaded. If the **Log Upload** check box is enabled in the Management Cockpit, the client can upload a log file by calling sap.Logger.upload(). If the **Log Upload** check box is disabled on the server, the client cannot upload without getting an HTTP/1.1 403 Forbidden error. For the Logger plugin, you must call sap.Logger.upload() to upload the log file and implement an upload button or something that does the upload from within the app to the server.

To upload a log file with the Logger plugin, these conditions must be met: 1) Log Upload check box is enabled on the server 2)The sap.Logger.upload() method is called by developer.

This is the expected work flow with the current architecture.

1) If a user has an issue, such as an exception error, he/she reports it by email or a call to customer center.

2) The Administrator (or developer) enables the app on the server. Then the Administrator lets the user know that he/she can upload the log file.

3) The user uploads a log file to the server. The Administrator gets the uploaded log file in the Management Cockpit.

4) The Administrator sends the file to the app's developer to debug.

On iOS, if the current log level is ERROR(default level), only ERROR level messages are displayed in the console even if other log level messages are generated. But if the current log level is DEBUG, INFO or WARN, all generated log messages, regardless of log level are displayed in the console.

On Android, all generated log messages, regardless of log level, are displayed in the Android log cat view (console).

When you use the Settings plugin, it 1) Gets the log level from the server, 2) Sets it into Logger on the client and 3) Calls sap.Logger.upload() upon a logon success event (when the app is launched or resumed and logon is successful). This means that the setting plugin retrieves the selected log level(type) from the Management Cockpit on the server, sets the log level to the Logger plugin and then automatically uploads a log file to the server. If the Settings plugin is not added in the project, a log file can be uploaded only by calling the sap.Logger.upload() method manually by developer. In the Management Cockpit (client logging dialog box), the Log Upload check box can enable and disable and you can choose the log type(level). The uploaded log files are listed in the Management Cockpit. There are seven log types (NONE, FATAL, ERROR, WARNING, INFO, DEBUG and PATH) on the server. Since the Kapsel Logger plugin supports only DEBUG, INFO, WARN and ERROR, the Logger plugin implicitly matches FATAL to ERROR and PATH to DEBUG. If NONE is set on server admin UI, logger sets it to default log level.

Adding and Removing the Logger Plugin

The Logger plugin is added and removed using the *Cordova CLI*.

To add the Logger plugin to your project, use the following command:

```
Cordova plugin add <path to directory containing Kapsel plugins>\logger
```

To remove the Logger plugin from your project, use the following command:

```
cordova plugin rm com.sap.mp.cordova.plugins.logger
```

Members

Name	Description
Logger#DEBUG on page 187	Constant variable for Debug log level.
Logger#ERROR on page 187	Constant variable for Error log level.
Logger#INFO on page 188	Constant variable for Information log level.
Logger#WARN on page 188	Constant variable for Warning log level.

Methods

Name	Description
debug(message, [tag], [successCallback], [errorCallback]) on page 188	Add a debug message to the log. This function logs messages with 'DEBUG' log level.
error(message, [tag], [successCallback], [errorCallback]) on page 189	Add an error message to the log. This function logs messages with 'ERROR' log level.
getLogLevel(successCallback, [errorCallback]) on page 191	Get log level.
info(message, [tag], [successCallback], [errorCallback]) on page 192	Add an info message to the log. This function logs messages with 'INFO' log level.
setLogLevel(level, [successCallback], [errorCallback]) on page 193	Set log level. This function sets the log level for logging.
upload(successCallback, errorCallback) on page 195	Upload a log file that contains log entries to SAP Mobile Platform server.
warn(message, [tag], [successCallback], [errorCallback]) on page 196	Add a warning message to the log. This function logs messages with 'WARN' log level.

Source
logger.js, line 61 on page 200.

Logger#DEBUG member
Constant variable for Debug log level.

It contains "DEBUG" string.

Syntax
<static, constant> `Logger#DEBUG` : String

Example

```
sap.Logger.setLogLevel(sap.Logger.DEBUG);
```

Source
logger.js, line 331 on page 210.

Logger#ERROR member
Constant variable for Error log level.

It contains "ERROR" string.

Syntax
<static, constant> `Logger#ERROR` : String

Example

```
sap.Logger.setLogLevel(sap.Logger.ERROR);
```

Source
logger.js, line 301 on page 209.

Logger#INFO member
Constant variable for Information log level.

It contains "INFO" string.

Syntax
<static, constant> `Logger#INFO` : String

Example

```
sap.Logger.setLogLevel(sap.Logger.INFO);
```

Source
logger.js, line 321 on page 210.

Logger#WARN member
Constant variable for Warning log level.

It contains "WARN" string.

Syntax
<static, constant> `Logger#WARN` : String

Example

```
sap.Logger.setLogLevel(sap.Logger.WARN);
```

Source
logger.js, line 311 on page 210.

debug(message, [tag], [successCallback], [errorCallback]) method
Add a debug message to the log. This function logs messages with 'DEBUG' log level.

Syntax
<static> `debug(` *message*, [*tag*], [*successCallback*], [*errorCallback*])

Parameters

Name	Type	Argument	Description
message	String		log message to be logged
tag	String	(optional)	Tag can indicate what this message is for.(ex. SMP_LOGGER, SMP_AUTHPROXY) It can be appended to messages at front.
successCallback	function	(optional)	success callback method upon success state.When a debug message is successfully logged, it is fired. No object will be passed to success callback.
errorCallback	function	(optional)	error callback method upon error state. For this method, since the Kapsel Logger native code always calls success callback after executing the API of the oData logger library, it is very unusual to fire an error callback.(If Cordova caused a system exception, the error callback could be fired by Cordova.)

Example

```
sap.Logger.debug("debug message", "DEBUG_TAG");
```

Source
logger.js, line 75 on page 201.

error(message, [tag], [successCallback], [errorCallback]) method
Add an error message to the log. This function logs messages with 'ERROR' log level.

Syntax

<static> `error`(*message*, [*tag*], [*successCallback*], [*errorCallback*])

Parameters

Name	Type	Argument	Description
message	String		log message to be logged
tag	String	(optional)	Tag can indicate what this message is for.(ex. SMP_LOGGER, SMP_AUTHPROXY) It can be appended to messages at front.
successCallback	function	(optional)	success callback method upon success state.When a debug message is successfully logged, it is fired. No object will be passed to success callback.
errorCallback	function	(optional)	error callback method upon error state. For this method, since the Kapsel Logger native code always calls success callback after executing the API of the oData logger library, it is very unusual to fire error callback.(If Cordova caused a system exception, the error callback could be fired by Cordova.)

Example

```
sap.Logger.error("error message", "ERROR_TAG");
```

Source

logger.js, line 150 on page 203.

getLogLevel(successCallback, [errorCallback]) method

Get log level.

This function gets current log level. Using this function, you can find out what kind of log level messages can be generated and affected at the current log level.

Syntax

<static> `getLogLevel(` *successCallback*, [*errorCallback*])

Parameters

Name	Type	Argument	Description
successCallback	function		success callback method upon success state.When current log level is successfully retrieved, it is fired with the current log level. [DEBUG, INFO, WARN, ERROR] Log level of String type will be passed to success callback. Default log level is ERROR.
errorCallback	function	(optional)	error callback method upon error state.For this method, error callback is optional. Since logger native code always passes log level to success callback, it is very unusual to fire error callback. (If cordova caused system exception, error callback could be fired by cordova)

Example

```
sap.Logger.getLogLevel(function(logLevel) {
  alert("Log level is " + logLevel);
}, function() {
  alert("Failed to get log level");
});
```

Source
logger.js, line 224 on page 206.

info(message, [tag], [successCallback], [errorCallback]) method
Add an info message to the log. This function logs messages with 'INFO' log level.

Syntax
<static> `info`(*message*, [*tag*], [*successCallback*], [*errorCallback*])

Parameters

Name	Type	Argument	Description
message	String		log message to be logged
tag	String	(optional)	Tag can indicate what this message is for.(ex. SMP_LOGGER, SMP_AUTHPROXY) It can be appended to messages at front.
successCallback	function	(optional)	success callback method upon success state.When a debug message is successfully logged, it is fired. No object will be passed to success callback.

errorCallback	function	(optional)	error callback method upon error state. For this method, since the Kapsel Logger native code always calls the success callback after executing the API of the oData logger library, it is very unusual to fire an error callback.(If Cordova caused a system exception, the error callback could be fired by Cordova.)

Example

```
sap.Logger.info("info message", "INFO_TAG");
```

Source

logger.js, line 100 on page 202.

setLogLevel(level, [successCallback], [errorCallback]) method

Set log level. This function sets the log level for logging.

Coverage of logging data in each log level: DEBUG < INFO < WARN < ERROR.

Following is the expected behavior to cover log messages at specific log levels.

Error : only error

Warn : error, warn

Info : error, warn, info

Debug : error, warn, info, debug

For example, if you want to get all log messages, you need to set it to 'Debug' level. If WARN level is set, logging data contains WARN and ERROR messages.

Default log level is ERROR.

Syntax

<static> `setLogLevel(` *level*, [*successCallback*], [*errorCallback*])

Parameters

Name	Type	Argument	Description
level	String		log level to set [DEBUG, INFO, WARN, ERROR]
successCallback	function	(optional)	success callback method upon success state.When log level is successfully set, it is fired. No object will be passed to success callback.
errorCallback	function	(optional)	error callback method upon error state. For this method, since kapsel logger native code always calls success callback after executing the API of oData logger library, it is very unusual to fire error callback.(If cordova caused system exception, error callback could be fired by cordova)

Example

```
sap.Logger.setLogLevel(sap.Logger.DEBUG, function(logLevel) {
  alert("Log level set");
}, function() {
  alert("Failed to set log level");
});
```

Source
logger.js, line 175 on page 204.

upload(successCallback, errorCallback) method
Upload a log file that contains log entries to SAP Mobile Platform server.

This function uploads a log file, which is helpful for developers who want to collect logging data from the app to trace bugs and issues. It uploads a log file which contains log entries based on log level. Developers can access the log data in the Management Cockpit or a specific folder in the installed server directly.

On iOS logger, when uploading a log file, the uploaded log messages are filtered by the log level at upload. For example, when you upload a log file at the ERROR log level, uploaded log messages contain only error log level messages. When you upload a log file at the INFO level, uploaded log messages contain error, warn, and info log level messages.

Android logger just filters the generated log messages "at the log level." In other words, the already generated and filtered log messages at another log level are not affected by current log level. Log messages are not filtered at uploading. For example, set the log level to DEBUG and log four levels (DEBUG, INFO, WARN and ERROR) of messages. At this time, the Android logger has four log level messages. If you set the log level to WARN and upload a log file, the log file has four log level messages which were already generated at the DEBUG level.

Syntax
<static> `upload`(*successCallback*, *errorCallback*)

Parameters

Name	Type	Description
successCallback	function	success callback method upon success state.This method is fired when the log file is successfully uploaded(with http statusCode and statusMessage for success).

errorCallback	function	error callback method upon error state.If there is a connectivity error, such as an HTTP error or unknown server error, this method is fired with http statusCode and statusMessage for error.

Example

```
sap.Logger.upload(function() {
  alert("Upload Successful");
}, function(e) {
  alert("Upload Failed. Status: " + e.statusCode + ", Message: " +
e.statusMessage);
});
```

Source

logger.js, line 250 on page 207.

warn(message, [tag], [successCallback], [errorCallback]) method

Add a warning message to the log. This function logs messages with 'WARN' log level.

Syntax

<static> `warn`(*message*, [*tag*], [*successCallback*], [*errorCallback*])

Parameters

Name	Type	Argument	Description
message	String		log message to be logged
tag	String	(optional)	Tag can indicate what this message is for.(ex. SMP_LOGGER, SMP_AUTHPROXY) It can be appended to messages at front.

successCallback	function	(optional)	success callback method upon success state.When a debug message is successfully logged, it is fired. No object will be passed to success callback.
errorCallback	function	(optional)	error callback method upon error state. For this method, since the Kapsel Logger native code always calls success callback after executing the API of the oData logger library, it is very unusual to fire an error callback.(If Cordova caused a system exception, the error callback could be fired by Cordova.)

Example

```
sap.Logger.warn("warn message", "WARN_TAG");
```

Source

logger.js, line 125 on page 203.

Source code

logger.js

```
1       // ${project.version}

2       var exec = require('cordova/exec');

3

4       /**

5       * The Kapsel Logger plugin provides a Cordova plugin wrapper
around the SAP Mobile Platform client logging API.

6        * <br><br>

7        *
```

```
 * The Logger plugin has ERROR, WARN, INFO, and DEBUG log levels and log messages are captured based on the configured and selected log level.
 * A Kapsel application can be set to these log levels by programmatic control, and by the administrator changing a setting on the server.
 * For Android and iOS, the default log level is ERROR, so by default only ERROR level logs are captured.
 * sap.Logger.setLogLevel() method is used to set other levels. If you want to get log messages at all log levels,
 * you must set the log level to DEBUG. (DEBUG < INFO < WARN < ERROR) <br>
 * If the log level is set to DEBUG, the application captures all log messages. <br>
 * If you set the log level to INFO, the application captures INFO, WARN, and ERROR log messages. <br>
 * If you set the log level to WARN, the application captures WARN and ERROR log messages. <br>
 * If you set the log level to ERROR, the application captures only Error log messages.
 * <br><br>
 *
 * Using the provided sap.Logger.upload() method allows developers to upload a log file to SAP Mobile Platform Server,
 * where an administrator can view them and remotely set the appropriate log level to control the amount of information
 * that is written to the log. When the sap.Logger.upload() method is triggered, a log file will be uploaded.
 * If the Log Upload checkbox is selected in the Management Cockpit, the client can upload a log file by calling sap.Logger.upload().
 * If the Log Upload checkbox is disabled in the Management Cockpit, the client does not upload the log file to the server. The attempt to upload causes an "HTTP/1.1 403 Forbidden" error.
 * To support manual uploading of the log, you should implement a button or some other mechanism that calls sap.Logger.upload() when needed.
 * <br>
 * For the Logger plugin to upload a log file these conditions must be met: 1) Log Upload checkbox enabled In the Management Cockpit 2) sap.Logger.upload() is called by developer.
```

```
     * <br>
     * The expected work flow, with the current architecture consists of the following: <br>
    * 1) If a user has an issue that needs to be analyzed by an administrator or developer, the user reports the issue as appropriate.<br>
     * 2) The administrator, or developer, enables the log collection for the user on the SAP Mobile Platform server.<br>
     * 3) The administrator lets the user know that he, or she, can upload log file. <br>
     * 4) The user uploads thelog file to the server, and the administrator gets the uploaded log file in the Management Cockpit.<br>
     * 5) The administrator sends the file to the developer to debug.
     * <br><br>
     *
     * Currently, on iOS, if the current log level is ERROR (default level), only ERROR level messages are displayed on the console
    * even if other log level messages are generated. But if the current log level is DEBUG, INFO, or WARN,
     * all generated log messages, regardless of log level, are displayed on the console. <br>
    * On Android, all generated log messages, regardless of log level, are displayed in the Android logcat view (console)
     * <br><br>
     *
     * When the Kapsel Settings plugin is added to the project, Settings will: 1) Get log level from the server 2) Set it into Logger on the client
     * 3) Call sap.Logger.upload() after a logon success event, for example, when the app is launched or resumed and logon is successful.
    * The Settings plugin retrieves the selected log level(type) from the Management Cockpit on the server,
   * sets the log level to Logger plugin, and then automatically uploads a log file to the server.
    * If the Settings plugin is not added to the project, a log file can be uploaded only by the developer calling the sap.Logger.upload() method manually.
```

```
     * In the Management Cockpit, in the Client Logging dialog box, the Log Upload checkbox is able to enable or disable log file upload, and you can choose the log type(level).
    * You can also view a list of the uploaded log files. On the server side, there are seven log types: NONE, FATAL, ERROR, WARNING, INFO, DEBUG and PATH.
    * Since the Kapsel Logger plugin supports only DEBUG, INFO, WARN, and ERROR, the Logger plugin implicitly matches FATAL to ERROR, and PATH to DEBUG.
     * If NONEÂ isÂ setÂ inÂ the Management Cockpit, LoggerÂ setsÂ itÂ toÂ defaultÂ log level.
     * <br><br>
     * <b>Adding and Removing the Logger Plugin</b><br>
     * Add or remove the Logger plugin using the
     * <a href="http://cordova.apache.org/docs/en/edge/guide_cli_index.md.html#The%20Command-line%20Interface">Cordova CLI</a>.<br>
     * <br>
   * To add the Logger plugin to your project, use the following command:<br>
     * Cordova plugin add <path to directory containing Kapsel plugins>\logger<br>
     * <br>
     * To remove the Logger plugin from your project, use the following command:<br>
     * cordova plugin rm com.sap.mp.cordova.plugins.logger
     * <br>
     *
     * @namespace
     * @alias Logger
     * @memberof sap
     */

    Logger = function () {
        /**
         * Formating for message
         * @private
         */
        var format = function (message) {
            if ((message === null) || (message === undefined)) {
                return "";
            }

            return message.toString();
        }


        /**
         * Add a debug message to the log.
         * This function logs messages with a 'DEBUG' log level.
         *
         * @memberof sap.Logger
         * @method debug
         * @param {String} message Log message to log.
       * @param {String} [tag]   Tag value added to the log entry used to indicate the source of the message (ex. SMP_LOGGER, SMP_AUTHPROXY).
       * @param {function} [successCallback]  Callback function called when the message has been successfully added to the log.
       *                                * No object will be passed to success callback.
       * @param {function} [errorCallback]    Callback function called when an error occurs while adding the message to the log.
         * Since Kapsel Logger native code will always call the success callback function, the
       * errorCallback function will be executed by Cordova if an error or exception occurs
         * while making the call to the plugin.
         * @public
         * @memberof sap.Logger
         * @example
```

```
         * sap.Logger.debug("debug message", "DEBUG_TAG");
         */
        this.debug = function (message, tag, successCallback, errorCallback) {
            exec(successCallback, errorCallback, "Logging", "logDebug", [format(message), tag]);
        }

        /**
         * Add an info message to the log.
         * This function logs messages with the 'INFO' log level.
         *
         * @memberof sap.Logger
         * @method info
         * @param {String} message Log message to be logged
         * @param {String} [tag]   Tag value added to the log entry used to indicate the source of the message (for example, SMP_LOGGER, SMP_AUTHPROXY).
         * @param {function} [successCallback]  Callback function called when the message has been successfully added to the log.
         * No object will be passed to success callback.
         * @param {function} [errorCallback]    Callback function called when an error occurs while adding the message to the log.
         * Since Kapsel Logger native code will always call the success callback function, the
         * errorCallback function will be executed by Cordova if an error or exception occurs
         * while making the call to the plugin.
         * @public
         * @memberof sap.Logger
         * @example
         * sap.Logger.info("info message", "INFO_TAG");
         */
        this.info = function (message, tag, successCallback, errorCallback) {
            exec(successCallback, errorCallback, "Logging", "logInfo", [format(message), tag]);
        }

        /**
         * Add a warning message to the log.
         * This function logs messages with the 'WARN' log level.
         *
         * @memberof sap.Logger
         * @method warn
         * @param {String} message Log message to be logged.
        * @param {String} [tag]   Tag value added to the log entry used to indicate the source of the message (for example, SMP_LOGGER, SMP_AUTHPROXY).
        * @param {function} [successCallback]  Callback function called when the message has been successfully added to the log.
         * No object will be passed to success callback.
        * @param {function} [errorCallback]    Callback function called when an error occurs while adding the message to the log.
         * Since Kapsel Logger native code will always call the success callback function, the
        * errorCallback function will be executed by Cordova if an error/exception occurs
         * while making the call to the plugin.
         * @public
         * @memberof sap.Logger
         * @example
         * sap.Logger.warn("warn message", "WARN_TAG");
         */
        this.warn = function (message, tag, successCallback, errorCallback) {
            exec(successCallback, errorCallback, "Logging", "logWarning", [format(message), tag]);
        }

        /**
         * Add an error message to the log.
         * This function logs messages with the 'ERROR' log level.
         *
         * @memberof sap.Logger
         * @method error
         * @param {String} message log Message to be logged.
        * @param {String} [tag]   Tag value added to the log entry used to indicate the source of the message (for example, SMP_LOGGER, SMP_AUTHPROXY).
        * @param {function} [successCallback]  Callback function called when the message has been successfully added to the log.
         * No object will be passed to success callback.
        * @param {function} [errorCallback]    Callback function called when an error occurs while adding the message to the log.
         * Since Kapsel Logger native code will always call the success callback function, the
         * errorCallback function will be executed by Cordova if an error or exception occurs
         * while making the call to the plugin.
         * @public
         * @memberof sap.Logger
         * @example
         * sap.Logger.error("error message", "ERROR_TAG");
         */
        this.error =  function (message, tag, successCallback, errorCallback) {
            exec(successCallback, errorCallback, "Logging", "logError", [format(message), tag]);
        }

        /**
         * Set log level.
         * This function sets the log level for logging. <br>
         * Coverage of logging data in each log level:  DEBUG <
INFO < WARN < ERROR. <br>
         * Following is the expected behavior to cover log
messages at specific log levels: <br>
         *    ERROR : only ERROR messages <br>
         *    WARN  : ERROR and WARN messages <br>
         *    INFO  : ERROR, WARN and INFO <br>
         *    DEBUG : ERROR, WARN, INFO and DEBUG <br>
        * For example, if you want to get all log messages, you
need to set the log to the 'Debug' level.
       * If the WARN level is set, logging data contains WARN and
ERROR messages. <br>
         * Default log level is ERROR.
         *
         * @memberof sap.Logger
         * @method setLogLevel
         * @param {String} level Log level to set [DEBUG, INFO,
WARN, ERROR]
        * @param {function} [successCallback] Callback function
called when the log level has been successfully set.
         * No object will be passed to success callback.
        * @param {function} [errorCallback]   Callback function
called when an error occurs while setting the log level.
         * Since Kapsel Logger native code will always call the
success callback function, the
        * errorCallback function will be executed by Cordova if
an error or exception occurs
         * while making the call to the plugin.
         * @memberof sap.Logger
         * @example
         * sap.Logger.setLogLevel(sap.Logger.DEBUG,
successCallback, errorCallback);
         *
         * function successCallback() {
         *     alert("Log level set");
         * }
```

```
     *
     * function errorCallback() {
     *     alert("Failed to set log level");
     * }
     */
    this.setLogLevel = function (level, successCallback, errorCallback) {
        if (level.toLowerCase() === "fatal")
            level = "ERROR";
        else if (level.toLowerCase() === "path")
            level = "DEBUG";
        else if (level.toLowerCase() === "warning")
            level = "WARN";

        else if (level.toLowerCase() === "debug")
            level = "DEBUG";
        else if (level.toLowerCase() === "info")
            level = "INFO";
        else if (level.toLowerCase() === "error")
            level = "ERROR";

        exec(successCallback, errorCallback, "Logging", "setLogLevel", [level]);
    }

    /**
     * Get log level.
     * This function gets the current log level.
     * Use this function to know what kind of log level messages can be generated and affected at the current log level.
     *
     * @memberof sap.Logger
     * @method getLogLevel
     * @param {function} successCallback Callback function called when the log level has been successfully retrieved.
     * When the current log level is successfully retrieved, it is fired with the current log level. [DEBUG, INFO, WARN, ERROR]
     * Log level of String type will be passed to success callback.
     * Default log level is ERROR.
     * @param {function} [errorCallback] Callback function called when an error occurs while getting the current log level.  For this method, error callback is optional.
     * Since Kapsel Logger native code will always call the success callback function, the
     * errorCallback function will be executed by Cordova if an error or exception occurs
     * while making the call to the plugin.
     * @memberof sap.Logger
     * @example
     * sap.Logger.getLogLevel(successCallback, errorCallback);
     *
     * function successCallback(logLevel) {
     *     alert("Log level is " + logLevel);
     * }
     *
     * function errorCallback() {
     *     alert("Failed to get log level");
     * }
     */
    this.getLogLevel = function(successCallback, errorCallback) {
        exec(successCallback, errorCallback, "Logging", "getLogLevel",[]);
    }

    /**
     * Upload a log file, with log entries, to SAP Mobile Platform server.<br>
     * This function uploads a log file, which is helpful for collecting logging data from the app to trace bugs and issues.
     * It uploads a log file, which contains log entries based on log level.
     * Developers can access the log data in the Management Cockpit and/or a specific folder in installed server directly.<br><br>
     *
     * On iOS, the uploaded log messages are filtered by the log level at upon upload.
     * For example, when you upload a log file with an ERROR log level, the uploaded log messages contain only ERROR log level messages.
     * When you upload log files with an INFO level, uploaded log messages contain ERROR, WARN, and INFO log level messages.
     *
     * <br><br>
     * On Android, generated log messages are filtered "at the log level."
     * In other words, the already generated and filtered log messages at another log level are not affected by the current log level.
     * Log messages are not filtered upon upload. For example, if you set the log level to DEBUG log messages are filtered at four levels (DEBUG, INFO, WARN, and ERROR.
     * Logger on Android has four log levels messages. So, if you set the log level to WARN and upload a log file, the log file has four log level messages that were already generated at the DEBUG level.
     *
     * @memberof sap.Logger
     * @method upload
     * @param {function} successCallback Callback function called when a log file is successfully uploaded to the server.
     * When a log file is successfully uploaded, it is fired. (with http statusCode and statusMessage for success)
     * @param {function} errorCallback   Callback function called when an error occurs while uploading a log file to the server.
     * If there is a connectivity error, such as an HTTP error, or unknown server error,
     * it is fired with http statusCode and statusMessage for error.
     * @public
     * @memberof sap.Logger
     * @example
     * sap.Logger.upload(successCallback, errorCallback);
     *
     * function successCallback() {
     *     alert("Upload Successful");
     * }
     *
     * function errorCallback(e) {
     *     alert("Upload Failed. Status: " + e.statusCode + ", Message: " + e.statusMessage);
     * }
     */
     this.upload =  function (successCallback, errorCallback) {
         sap.Logon.unlock(function (connectionInfo) {
             //Add application ID required for REST call
             connectionInfo.applicationId = sap.Logon.applicationId;

             exec(successCallback, errorCallback, "Logging", "uploadLog", [connectionInfo]);
         }, function () {
             errorCallback({statusCode : 0, statusMessage : "Logon failed"});
         });
     }
 }

 /**
 * Constant variable for Error log level. It contains "ERROR" string.
```

```
     * @memberof sap.Logger
     * @constant
     * @type String
     * @example
     * sap.Logger.setLogLevel(sap.Logger.ERROR);
     */
    Logger.prototype.ERROR = "ERROR";

    /**
    * Constant variable for Warning log level. It contains "WARN" string.
     * @memberof sap.Logger
     * @constant
     * @type String
     * @example
     * sap.Logger.setLogLevel(sap.Logger.WARN);
     */
    Logger.prototype.WARN = "WARN";

    /**
     * Constant variable for Information log level. It contains "INFO" string.
     * @memberof sap.Logger
     * @constant
     * @type String
     * @example
     * sap.Logger.setLogLevel(sap.Logger.INFO);
     */
    Logger.prototype.INFO = "INFO";

    /**
    * Constant variable for Debug log level. It contains "DEBUG" string.
```

```
     * @memberof sap.Logger
     * @constant
     * @type String
     * @example
     * sap.Logger.setLogLevel(sap.Logger.DEBUG);
     */
    Logger.prototype.DEBUG = "DEBUG";

    module.exports = new Logger();

```

Using the Kapsel Push Plugin

The Push notification plugin enables push notification capability for Kapsel applications.

Push Plugin Overview

The push plugin APIs enable you to send push data to Kapsel applications.

The push notification system consists of:

1. The Kapsel application, which runs on the device and receives the notifications.
2. The notification service provider, for example, APNS for Apple devices, and GCM for Android devices.
3. The SAP Mobile Platform Server, which collects device IDs from the clients and push notifications through the notification service provider.

The Kapsel Push plugin allows you to enroll applications for notification with notification registration, as well as to receive and process incoming notifications for Kapsel applications. This plugin also supports background notification processing.

In a typical deployment, SAP Mobile Platform Server sends push messages to a push server through a RESTful API, which in turn delivers the push message to the user agent, which then provides execution instructions for the app. The user agent then delivers the push message to the designated app.

The push API tasks include:

- Registering and unregistering a push notification
- Push notificaton handling
- Push notification configuration
- Error message handling

Note: As a best practice, you should rarely use the unregister function. It is explained in detail at *http://developer.android.com/google/gcm/adv.html#unreg*.

Provisioning Devices for Push

You must register your device with a notification service, such as Apple Push Notification Service (APNS) for Apple devices, or Google Cloud Messaging (GCM) for Android devices.

In a production environment, when you register your application with the notification service provider, the device ID (iOS) or the registration ID (Android), is sent to the SAP Mobile Platform server. For iOS, the push certificate is stored there, and is used to authenticate push requests to the APNS server. When a push request is processed, that information is then used to target specific apps running on individual devices.

Provision the iOS Device for APNS

SAP Mobile Platform provides support for Apple Push Notification Service (APNS) by pushing notifications to Kapsel while it is offline.

With APNS, each device establishes encrypted IP connections to the service and receives notifications about availability of new items that are awaiting retrieval from the server. On 3G networks, this feature overcomes network issues with always-on connectivity and battery life consumption.

Note: APNS cannot be used on a simulator.

Examples of cases when notifications are sent include when the server identifies that a new message needs to be sent to the device, for example, when a new app is assigned to the device, or a push notification message is sent to the server and targeting a particular user when the app is not running.

See the *Apple Local and Push Notification Programming Guide* at *Provisioning and Development*, where APNS is documented in detail.

Register with Apple to download and use the iOS SDK to develop with the simulator. To deploy applicationss to devices, you must create a certificate in your developer account and provision your device.

Generating a Certificate Request File

Create a certificate signing request file to use for authenticating the creation of the SSL certificate.

1. Launch the Keychain Access application on your Mac (usually found in the **Applications** > **Utilities** folder.
2. Select **Keychain Access** > **Certificate Assistant** > **Keychain Access**.
3. Enter your e-mail address and name, then select **Save to disk**, and click **Continue**.

   This downloads the `.certSigningRequest` file to your desktop.

Creating an App ID

Create a new App ID for the application.

As a convention, the App ID is in the form of a reversed addresse, for example, com.example.MyPushApp. The App ID must not contain a wildcard character ("*").

1. Go to the *Apple Developer Member Center* Web site, log in, if required, and select *Certificates, Identifiers & Profiles*.
2. Select **Identifiers** > **App IDs**, and click the **+**.
3. Enter a name for your App ID, and, under App Service, select **Push Notifications**.
4. Accept the default App ID prefix, or choose another one.
5. Under App ID Suffix, select **Explicit App ID**, and enter your iOS app's Bundle ID.

   This string should match the Bundle Identifier in your iOS app's `Info.plist`.
6. Select **Continue**.

   Verify that all the values are correct. Push Notifications should be enabled, and the Identifier field should match your app's Bundle Identifier (plus App ID Prefix).
7. Click **Submit**.

Configuring the App ID for Push Notifications

Once you create an App ID, you must configure it for push notifications.

1. From the list of iOS App IDs, select the App ID to configure, then select **Settings**.
2. Scroll down to the Push Notifications section and, under Development SSL Certificate, select **Create Certificate**.

   Here you can create both a Development SSL Certificate and a Production SSL Certificate.
3. Follow the instructions for creating a Certificate Signing Request (CSR), select **Continue**, then select **Choose File** to locate the `.certSigningRequest` you created.
4. Click **Generate**.
5. Click **Done** once the certificate is ready, and download the generated SSL certificate from the iOS App ID Settings screen.
6. Install the SSL in your Keychain.
   a) In Keychain Access, under My Certificates, find the certificate you just added, right-click on it, select **Export Apple Development IOS Push Services**, and save it as a `.p12` file.

   Note: Do not enter an export password when prompted. You may, however, need to enter your OSX password to allow Keychain Access to export the certificate from your keychain.

Creating the Provisioning File

Create a provisioning profile to authenticate your device to run the app you are developing.

If you create a new App ID or modify an existing one, you must regenerate and install your provisioning file.

1. Navigate to the *Apple Developer Member Center* Web site, and select *Certificates, Identifiers & Profiles*.
2. From the iOS Apps section, select **Povisioning File**, and select the + button to create a new provisioning file.
3. Choose **iOS App Development** as your provisioning profile type, then click **Continue**.
4. From the drop-down, choose the App ID you created and click **Continue**.
5. Select your iOS Development certificate in the next screen, and click **Continue**.
6. Select which devices to include in the provisioning profile, and click **Continue**.
7. Choose a name for your provisioning profile, then click **Generate**.
8. Click **Download** to download the generated provisioning file.
9. Double-click the downloaded provisioning file to install it.

   Xcode's Organizer opens in the Devices pane. Your new provisioning profile appears in the Provisioning Profiles section of your Library. Verify that the status for the profile is "Valid profile." If the profile is invalid, verify that your developer certificate is installed in your Keychain.

Provision Android Devices for Push

Use this procedure to provision your Android device for Google Cloud Messaging Service (GCM).

Configuring Google Cloud Messaging Service

Google Cloud Messaging (GCM) is a service that allows you to send data from the server to Android devices, and also to receive messages from devices on the same connection.

For information about GCM, see *http://developer.android.com/google/gcm/gs.html*.

1. Open the Google APIs Console page and select **Services**.
2. Turn the Google Cloud Messaging toggle to **ON**.
3. Accept the terms of the Google APIs Terms of Service.
4. Create a new server key.
   a) Select **API Access**.
   b) Click **Create new Server key**.

      Either a server key or browser key work. If you use a server key, it allows you to whitelist IP addresses.

      Note: If you get a “Sender Mismatch ID” error when using a Server key, use the Browser key instead. You can find your ProjectID from the URL in your Google API console. Usually, it looks similar to this: `https://code.google.com/apis/console/.........#project:348986612458`, where 348986612458 is the ProjectID.

c) Click **Create**.

Take note of the API key value, as you need this to register.

Adding the Push Notification Plugin

Install the Push plugin using the Cordova command line interface.

Prerequisites

- Set up the development environment.
- Create your Cordova Project.
- Add your OS platforms.

Task

When you add the Push plugin to your project, the Settings and Logger plugins are also added automatically.

1. Add the Push plugin by entering the following at the command prompt, or terminal:

   On Windows:

   ```
   cordova -d plugin add <SDK_HOME>\MobileSDK3\KapselSDK
   \plugins\push
   ```

   On Mac:

   ```
   cordova -d plugin add ~<SDK_HOME>/MobileSDK3/KapselSDK/
   plugins/push
   ```

   Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).

2. (Optional) To see a list of installed plugins in your Cordova project, open a command prompt or terminal window, navigate to your Cordova project folder, and enter:

   ```
   cordova plugins
   ```

   The Cordova command line interface returns a JSON array showing installed plugins, for example:

   ```
   [ 'org.apache.cordova.core.camera',
   'org.apache.cordova.core.device-motion',
   'org.apache.cordova.core.file' ]
   ```

   In this example, the Cordova project has the Cordova core Camera, Accelerator (device-motion), and File plugins installed.

3. Modify the files in the `www` folder for the project as necessary, then copy them to the platform directories by running:

```
cordova -d prepare android
cordova -d prepare ios
```

4. Use the Android IDE or Xcode to deploy and run the project.

Note: If you are using an iOS device, remember to add the "clienthubEntitlements" to the Keychain Groups in the Entitlement section in Xcode.

Configuring Push on SAP Mobile Platform Server

You must explicitly register the application connection using the Management Cockpit.

1. Start the Management Cockpit.
2. Select **Applications**, and click **New**.
3. In the **New Application** window, enter values.

Field	Value
ID	Unique identifier for the application in reverse domain notation.
Name	Application name.
Vendor	(Optional) Vendor who developed the application.
Version	Application version. Currently, only version 1.0 is supported.
Type	Application type. • Native – native iOS and Android applications. • Hybrid – container-based applications, such as Kapsel. • Agentry – metadata-driven applications, such as Agentry. Application configuration options differ depending on your selection.
Description	(Optional) Short description of the application.

4. Click the **Backend** tab and configure the endpoint information.
5. Click the **Push** tab to configure the push settings.

   For Android GCM, see *Android Push Notifications*.

   For Apple APNS, see *Apple Push Notifications*
6. In the settings for your device, enable the app to receive push notifications.

Android Push Notifications

Configure Android push notifications for the selected application, to enable client applications to receive Google Cloud Messaging (GCM) notifications.

1. From Management Cockpit, select **Applications** > **Push**.
2. Under Android, enter the access key for API key. This is the access key you obtained for your Google API project (*http://developer.android.com/google/gcm/gs.html*).
3. Enter a value for Sender ID. This is the project identifier.

4. (Optional) Configure push notifications for each device type supported.

Apple Push Notifications

Configure Apple Push Notifications for the selected application, to enable client applications to receive APNS notifications.

1. From Management Cockpit, select **Applications > Push**.

2. Under Apple, select **APNS endpoint**. "None" is the default endpoint value for all the applications.

3. Select **Sandbox** to configure APNS in a development and testing environment, or **Production** to configure APNS in a production environment.

   a) Click **Browse** to navigate to the certificate file.
   b) Select the file, and click **Open**.
   c) Enter a valid password.

   Note: The default URL is for a production environment; for a development and testing environment, change the URL to gateway.sandbox.push.apple.com.

4. (Optional) Configure push notifications for each device type supported.

Testing Push Notifications

Test the push and settings plugins.

1. Open the project in your development IDE.

2. Build and run the project.

3. Send a REST request to send a notification to the Kapsel app.

Sample Application for Android

You can use this code to test the Push and Settings APIs on Android.

Make sure you examine the code carefully and make the necessary changes as explained in the comments.

```
<html>
    <head>
        <script src="cordova.js"></script>
        <script>
            applicationContext = null;
            appId = "bobapp2"; //Place your application id here

            smpURL = null;

            function init() {
                // Optional initial connection context
                var context = {
                 "serverHost": "machine_name.com", //Place your SMP
3.0 server name here
                   "https": "false",
                    "serverPort": "80",
                  "user": "smpAdmin", //Place your user name for the
OData Endpoint here
                  "password": "s3pAdmin",  //Place your password for
the OData Endpoint here
                    "communicatorId": "REST",
                    "passcode": "password",
                    "unlockPasscode": "password"
                };
                sap.Logon.init(logonSuccessCallback, function()
{ alert("Logon Failed"); }, appId, context, sap.Logon.IabUi);
                sap.Logger.setLogLevel(sap.Logger.DEBUG);
            }

            function register() {
                try {
                   sap.Logon.registerOrUnlock(logonSuccessCallback,
errorCallback);
                }
                catch (e) {
                    alert("Problem with register");
                }
            }

            function unRegister() {
                try {

sap.Logon.core.deleteRegistration(logonUnregisterSuccessCallback,
errorCallback);
                }
                catch (e) {
                    alert("problem with unregister");
                }
            }

            function logonSuccessCallback(result) {
                console.log("logonSuccessCallback " +
JSON.stringify(result));
              if (result) {  //calling registerOrUnlock returns null
the second time it is called.  Possible bug.
                    applicationContext = result;
                 smpURL = applicationContext.applicationEndpointURL;
                    console.log(smpURL);
                    if (smpURL.charAt(smpURL.length - 1) == "/") {
                        smpURL = smpURL.substring(0,
applicationContext.applicationEndpointURL.length - 1);
                    }
                    console.log(smpURL);
                    smpURL = smpURL.substring(0,
smpURL.lastIndexOf("/"));
                    console.log(smpURL);
                }
            }

            function logonUnregisterSuccessCallback(result) {
                console.log("logonUnregisterSuccessCallback " +
JSON.stringify(result));
                applicationContext = null;
            }

            function errorCallback(e) {
                alert("An error occurred");
                alert(JSON.stringify(e));
            }

            function registerForPush() {
                var nTypes = sap.Push.notificationType.SOUNDS |
sap.Push.notificationType.ALERT | sap.Push.notificationType.BADGE;
                sap.Push.registerForNotificationTypes(nTypes,
regSuccess, regFailure, proccessNotification, "186452565698");  //
GCM Sender ID, null for APNS

            }

            function unregisterForPush() {
                var nTypes = sap.Push.notificationType.SOUNDS |
sap.Push.notificationType.ALERT;

sap.Push.unregisterForNotificationTypes(unregCallback);
            }

            function regSuccess(mesg) {
                alert("Successfully registered"+mesg);
            }

            function regFailure(errorInfo) {
                alert("Failed to register");
                alert(JSON.stringify(errorInfo));
                console.log("Error while registering.  " +
JSON.stringify(errorInfo));
            }

            function unregCallback(msg) {
                alert("In unregCallback with params");
                console.log("Unregistered" + JSON.stringify(msg));
            }

            function unregCallback() {
                alert("In unregCallback with no params");
            }

            function proccessNotification(notification) {
                console.log("Received a notifcation: " +
JSON.stringify(notification));
            }

            function proccessMissedNotification(notification) {
                alert("In processMissedNotification");
                console.log("In processMissedNotification");
                console.log("Received a missed notifcation: " +
JSON.stringify(notification));
            }
            function checkForNotification(notification) {

sap.Push.checkForNotification(proccessMissedNotification);
            }

            function showRegistrationInfo() {
                xmlhttp = new XMLHttpRequest();
                var url = smpURL + "/odata/applications/latest/" +
appId + "/Connections('" +
applicationContext.applicationConnectionId + "')";
                alert(url);
                xmlhttp.open("GET", url, false);
                xmlhttp.setRequestHeader("X-SMP-APPCID",
applicationContext.applicationConnectionId);
                xmlhttp.send();
                var responseText = xmlhttp.responseText;
                alert(responseText);
                console.log(responseText);
            }

            function getBadgeCallback(data) {
                alert("The badge number is : "+ data["badgecount"]);
            }

            function getBadgeNum(){
                if(device.platform == "Android"){
                    alert("badge number is iOS only!");
                    return;
                }
                sap.Push.getBadgeNumber(getBadgeCallback);


            }

            function badgeCallBack(msg){
                alert("Set badget number : " + msg);
            }

            function setBadgeNum(){
                if(device.platform == "Android"){
                    alert("badge number is iOS only!");
                    return;
                }

                sap.Push.setBadgeNumber(10, badgeCallBack);

            }

            function resetBadgeCallback(msg){
                alert("Reset badge number : " + msg);
            }

            function resetBadgeNum(){

                if(device.platform == "Android"){
                    alert("badge number is iOS only!");
                    return;
                }
                sap.Push.resetBadge(resetBadgeCallback);

            }

            document.addEventListener("deviceready", init, false);
            </script>
    </head>
    <body>
        <h1>Push</h1>
        <button onclick="registerForPush()">Register For Push</
button><br><br>
        <button onclick="unregisterForPush()">Unregister For Push</
button><br><br>
        <button onclick="checkForNotification()">Check for
Notification</button><br><br>
        <button onclick="showRegistrationInfo()">Registration Info</
button><br><br>
        <button onclick="getBadgeNum()">Get Badge Number</
button><br><br>
        <button onclick="setBadgeNum()">Set Badge Number</
button><br><br>
        <button onclick="resetBadgeNum()">Reset Badge Number to 0</
button><br>
    </body>
</html>
```

Notification Data Sent Through HTTP Headers

Notification data can be sent by the back end as a generic HTTP headers or as device platform-specific HTTP headers.

The notification URL is:

```
http[s]://<host:port/Notification>/<registration id>
```

Note: Applications built in SAP Mobile Platform 3.0 and later should adopt the header format `X-SMP-XXX`. To maintain backward compatibility, applications built in earlier versions can continue to use the header format `X-SUP-XXX`. However, `X-SUP-XXX`headers will be removed future releases.

- **Generic header**

  The generic HTTP header is used in the HTTP request to send any notification type such as APNS, GCM, Blackberry, or WNS.

  Header format for notification data in SAP Mobile Platform 3.x and later:

  ```
  <X-SMP-DATA>
  ```
- **APNS-specific headers**

  Use these APNS-specific HTTP headers to send APNS notifications via SAP Mobile Platform:

Header Structure (SAP Mobile Platform and later)	Consists of
`<X-SMP-APNS-ALERT>`	A JSON document. You can use this header or other individual headers listed in this table.
`<X-SMP-APNS-ALERT-BODY>`	Text of the alert message.
`<X-SMP-APNS-ALERT-ACTION-LOC-KEY>`	If a string is specified, this header shows an alert with two buttons: **Close** and **View**. iOS uses the string as a key to get a localized string for the correct button title instead of **View**. If the value is null, the system shows an alert. Clicking **OK** dismisses the alert.
`<X-SMP-APNS-ALERT-LOC-KEY>`	Key to an alert-message string in a `Localizable.strings` file for the current localization.
`<X-SMP-APNS-ALERT-LOC-ARGS>`	Variable string values to appear in place of the format specifiers in `loc-key`.
`<X-SMP-APNS-ALERT-LAUNCH-IMAGE>`	File name of an image file in the application bundle. It may include the extension. Used as the launch image when you tap the action button or move the action slider. If this property is not specified, the system uses on of the following: • The previous snapshot • The image identified by the `UILaunchImageFile` key in the `Info.plist` file of the application • The `Default.png`.
`<X-SMP-APNS-BADGE>`	Number that appears as the badge on the application icon.
`<X-SMP-APNS-SOUND>`	Name of the sound file in the application bundle.
`<X-SMP-APNS-DATA>`	Custom payload data values. These values must use the JSON-structured and primitive types, such as dictionary (object), array, string, number, and Boolean.

For additional information about APNS headers, see the Apple Web site: *http://developer.apple.com/library/mac/#documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/ApplePushService/ApplePushService.html*.

- **GCM-specific headers**
  Use these GCM-specific HTTP headers to send GCM notifications:

Header Structure (SAP Mobile Platform and later)	Consists of
`<X-SMP-GCM-COLLAPSEKEY >`	An arbitrary string (such as "Updates Available") that collapses a group of like messages when the device is offline, so that only the last message is sent to the client. **Note:** If you do not include this header, the default value "Updates Available, is used
`<X-SMP-GCM-DATA>`	Payload data, expressed as parameters prefixed with data and suffixed as the key.
`<X-SMP-GCM-DELAYWHILEIDLE>`	(Optional) Represented as 1 or true for true, any other value for false, which is the default value.
`<X-SMP-GCM-TIMETOLIVE>`	Time (in seconds) that the message remains available on GCM storage if the device is offline.

For additional information about GCM headers, see the Android Web site: *http://developer.android.com/guide/google/gcm/gcm.html#send-msg*.

- **BES/BIS-specific header**

  Use the BlackBerry-specific HTTP header to send BES/BIS notifications:

  ```
  <x-sup-rim-data> or <X-SMP-RIM-DATA>
  ```
- **WNS specific header**

  Use these HTTP headers to send Windows 8 desktop and tablet application notifications:

Header Structure (SAP Mobile Platform and later)	Consists of
`<X-SMP-WNS-DATA>`	Send payload data to the device as raw notification. Payload data may also be a binary data encoded as a Base64-encoded string. Size should not exceed 5KB.
`<X-SMP-WNS-ALERT>`	Text string of the notification, as Tile and Toast notifications.
`<X-SMP-WNS-BADGE>`	Number that appears as the badge on the application icon.

- MPNS (Notification for Windows Phone)

  Use these Windows Phone-specific HTTP headers to send MPNS notifications:

Request Header Structure	Consists of
`<X-SMP-MPNS-DATA>`	Send payload data to device as raw notification. Payload data may also be a binary data encoded as a Base64-encoded string. String length should not exceed more than 2900 characters.
`<X-SMP-MPNS-ALERT>`	Text string of the notification, as Tile and Toast notifications.
`<X-SMP-MPNS-BADGE>`	Number that appears as the badge on the application icon.

Kapsel Push API Reference

The Kapsel Push API Reference provides usage information for Push API classes and methods, as well as provides sample source code.

Push namespace

The push plugin provides an abstraction layer over the

Google Cloud Messaging for Android (GCM) and *Apple Push Notification Service (APNS)*.

A notification can be sent to a device registered with an application through a rest call at

```
http://SMP_3.0_SERVER:8080/Notifications/
application_registration_id
```

Adding and Removing the Push Plugin

The Push plugin is added and removed using the *Cordova CLI*.

To add the Push plugin to your project, use the following command:

cordova plugin add <path to directory containing Kapsel plugins>\push

To remove the Push plugin from your project, use the following command:

```
cordova plugin rm com.sap.mp.cordova.plugins.push
```

Methods

Name	Description
checkForNotification(callback) on page 225	This method checks for any notifications received while the application was not running in the foreground.
getBadgeNumber(callback) on page 226	Used to fetch the badge count for the application.
registerForNotificationTypes(types, successCallback, errorCallback, notificationlistenerfunc, [senderId]) on page 227	Function called by the application to register notification types to receive.
resetBadge(callback) on page 228	Used to reset the badge count for the application.
setBadgeNumber(number, callback) on page 229	Used to set the badge count for the application.
unregisterForNotificationTypes(callback) on page 229	Function called by the application to unregister from future notifications.

Type Definitions

Name	Description
callback([devtok]) on page 230	This method updates the application with the new device token in the SAP Mobile Platform server.

Source

push.js, line 29 on page 232.

checkForNotification(callback) method

This method checks for any notifications received while the application was not running in the foreground.

Application developer can call this function directly or register with an event handler to be called automatically. It is ok to call this function evenif the device is not yet registered for push notification.

Syntax

<static> `checkForNotification`(*callback*)

Parameters

Name	Type	Description
callback	function	The callback function that receives the notification.The callback function will receive a string as it's argument. This string will contain the notification message send from the server intact.

Example

```
function processBackgroudMessage(mesg){

}
function checkBackgroundNotification() {
    sap.Push.checkForNotification(processBackgroudMessage);
}
document.addEventListener("onSapLogonSuccess",
checkBackgroundNotification, false);
document.addEventListener("onSapResumeSuccess",
checkBackgroundNotification, false);
```

Source

push.js, line 340 on page 244.

getBadgeNumber(callback) method

Used to fetch the badge count for the application.

This function is used only by iOS. Other platforms do not have the badge count concept.

Syntax

<static> `getBadgeNumber(` *callback*)

Parameters

Name	Type	Description
callback	function	Success callback to call when to send the badge count.The callback function will contain an argument in json format with the current badge count. Look into the example for the deail on how to use them.

Example

```
function getBadgeNumCallback(data) { badgecount =
data["badgecount"];}
sap.Push.getBadgeNumber(getBadgeNumCallback);
```

Source

push.js, line 247 on page 240.

registerForNotificationTypes(types, successCallback, errorCallback, notificationlistenerfunc, [senderId]) method

Function called by the application to register notification types to receive.

Syntax

<static> `registerForNotificationTypes`(*types*, *successCallback*, *errorCallback*, *notificationlistenerfunc*, [*senderId*])

Parameters

Name	Type	Argument	Description
types	string		Types of notifications the application wants to receive.The different types of notifications are expressed in <code>notification-Type</code> Notifi-caion types allowed are Disable all notifica-tions (NONE: 0), Set badge count on app icon (BADGE: 1), Play sounds on receiving notification (SOUNDS: 2) and Show alert on receiving notification (ALERT: 4).
successCallback	string		Success callback to call when registration is successful.

errorCallback	string		Error callback to call when registration attempt fails.
notificationlistenerfunc	string		The function that receives the notification for processing by the application.
senderId	string	(optional)	The sender ID that is used for GCM registration.For other platforms it is null.

Example

```
regid = "211112269206";
function registerSuccess(mesg){}
function registerFailure(mesg) {}
function ProcessNotification(mesg){}
sap.Push.registerForNotificationTypes(sap.Push.notificationType.bad
ge | sap.Push.notificationType.sound |
sap.Push.notificationType.alert, registerSuccess, registerFailure,
ProcessNotification, regid);
```

Source

push.js, line 200 on page 238.

resetBadge(callback) method

Used to reset the badge count for the application.

This function is used only by iOS. Other platforms do not have the badge count concept.

Syntax

<static> `resetBadge`(*callback*)

Parameters

Name	Type	Description
callback	function	Success callback to call when the badge count is reset.The callback function will contain an argument in string format. This argument can be used for informative purpose.

Example

```
function badgeCallback(mesg){}
sap.Push.resetBadge(badgeCallback);
```

Source
push.js, line 284 on page 242.

setBadgeNumber(number, callback) method
Used to set the badge count for the application.

This function is used only by iOS. Other platforms do not have the badge count concept.

Syntax
<static> `setBadgeNumber(` *number*, *callback*)

Parameters

Name	Type	Description
number	number	The badge count to set for the application.
callback	function	Success callback to call when to send the badge count.The call-back function will contain an argument in string format. This argument can be used for informative purpose.

Example

```
function badgeCallback(mesg){}
badgenum = 10;
sap.Push.setBadgeNumber(badgenum, badgeCallback);
```

Source
push.js, line 265 on page 241.

unregisterForNotificationTypes(callback) method
Function called by the application to unregister from future notifications.

Syntax
<static> `unregisterForNotificationTypes(` *callback*)

Parameters

Name	Type	Description
callback	function	Success callback to call when deregistration is successful.This callback function will contain a string with a message. This message is just for informative purpose.

Example

```
function unregCallback(mesg){}
sap.Push.unregisterForNotificationTypes(unregCallback);
```

Source

push.js, line 229 on page 240.

callback([devtok]) type

This method updates the application with the new device token in the SAP Mobile Platform server.

Syntax

<static> `callback`([*devtok*])

Parameters

Name	Type	Argument	Description
devtok	string	(optional)	The device token received from the APNS/GCM device registration.

Example

```
function callback(mesg) {}
devToken ="123123213213";//sample device token
sap.Push.updateWithDeviceToken(devToken, callback);
```

Source

push.js, line 305 on page 242.

Source code

push.js

```
// ${project.version}
var exec = require("cordova/exec");

/**
 * The Kapsel Push plugin provides an abstraction layer over the
 * <a href="http://developer.android.com/google/gcm/index.html">Google Cloud Messaging for Android (GCM)</a>
 * and
 * <a href="http://developer.apple.com/library/mac/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html#//apple_ref/doc/uid/TP40008194-CH100-SW9">Apple Push Notification Service (APNS)</a>.
 * <br/><br/>
 * A notification can be sent to a device registered with an application through a
 * rest call at <pre>http://SMP_3.0_SERVER:8080/Notifications/application_registration_id</pre>
 * <br/><br/>
 * <b>Adding and Removing the Push Plugin</b><br/>
 * Add or remove the Push plugin using the
 * <a href="http://cordova.apache.org/docs/en/edge/guide_cli_index.md.html#The%20Command-line%20Interface">Cordova CLI</a>.<br/>
 * <br/>
 * To add the Push plugin to your project, use the following command:<br/>
 * cordova plugin add <path to directory containing Kapsel plugins>\push<br/>
 * <br/>
 * To remove the Push plugin from your project, use the following command:<br/>
 * cordova plugin rm com.sap.mp.cordova.plugins.push
 * <br/>
 *
 * @namespace
 * @alias Push
 * @memberof sap
 */

module.exports = {



    /**
     * Helper method for handling failure callbacks. It is configured as a failure callback in <code> call_native() </code>
     *
     *
     * @param {msg} Error message with the cause of failure
     *
     * @private
     * @name failure
     * @function
     */

failure: function (msg) {
    sap.Logger.debug("Javascript Callback Error: " + msg,"PUSHJS",function(m){},function(m){});

},

    /**
     * Helper method for calling native methods.
     *
     * @param {function} callback
     * @param {string} Name of the action to invoke on the plugin
```

```
         * @param {array} List of arguments
         * @private
         * @name call_native
         * @function
         */
    call_native: function (callback, name, args) {

        if(arguments.length == 2) {
            args = []
        }
        ret = exec(
                callback,             /**  Called when signature capture is successful */
                   sap.Push.failure,           /** Called when signature capture encounters an error */
                'SMPPushPlugin',        /**  Tell Cordova that to run "PushNotificationPlugin" */
                 name,                   /**  Tell the plugin the action to perform */
                args);                 /**  List of arguments to the plugin */
        return ret;
    },

        /**
         * Helper method to check if platform is iOS.
         *
         * @return {bool} Whether the current platform is iOS or not.
         * @private
         * @name isPlatformIOS
         * @function
         */
    isPlatformIOS: function () {
       return device.platform == "iPhone" || device.platform == "iPad" || device.platform == "iPod touch" || device.platform == "iOS"
    },
        /**
         * Function called by the application to get connection information.
         *
         * @param {string} [types] Types of notifications the application wants to receive. The different types of notifications are expressed in <code>notificationType</code>
         * @param {string} [successCB] Success callback to call when registration is successful.
        * @param {string} [errorCB] Error callback to call when registration attempt fails.
         * @private
         * @memberof sap.Push
         * @function getConnectionSettings
         * @example
         * sap.Push.getConnectionSettings(function(){
         * sap.Logger.debug("getting Connection Settings","PUSHJS",function(m){},function(m){});
         *  console.log("getting Connection Settings");
         *  sap.Push.registerForNotification(types, successCallback, errorCallback, notificationListenerFunc, senderId );
         **/
        getConnectionSettings : function (successCB, errorCB) {


            if (sap.Settings.isInitialized == true)
            {
                /*It is already initialized */
                successCB();

            } else {
```

```
sap.Settings.isInitialized = true;
var pd ="";
sap.Logon.unlock(function (connectionInfo) {
                 var userName = connectionInfo["registrationContext"]["user"];
                 var password  = connectionInfo["registrationContext"]["password"];
                 var applicationConnectionId = connectionInfo["applicationConnectionId"];
                 var securityConfig = connectionInfo["registrationContext"]["securityConfig"];
                 var endpoint = connectionInfo["applicationEndpointURL"];
                 var keySSLEnabled = "false";
                 var splitendpoint = endpoint.split("/");
                 if (splitendpoint[0] == "https:")
                 {
                     keySSLEnabled="true";
                 }
                 if (securityConfig == null) {
                     securityConfig = "";
                 }
                 var burl = splitendpoint[2];
                 var appId = splitendpoint[3];
                 pd = appId+userName+password;
                 sap.Settings.store = new sap.EncryptedStorage("SettingsStore", pd);
                 connectionData = {
                     "keyMAFLogonOperationContextConnectionData": {
                         "keyMAFLogonConnectionDataApplicationSettings":
                         {
                             "DeviceType":device.platform,
"DeviceModel":device.model,
"ApplicationConnectionId":applicationConnectionId
                                          },
"keyMAFLogonConnectionDataBaseURL":burl
                                      },
"keyMAFLogonOperationContextApplicationId":appId,
"keyMAFLogonOperationContextBackendUserName":userName,
"keyMAFLogonOperationContextBackendPassword":password,
"keyMAFLogonOperationContextSecurityConfig":securityConfig,
"keySSLEnabled":keySSLEnabled
                                  };
sap.Settings.start(connectionData,
                                                  function(mesg)
{
sap.Settings.isInitialized = true;
sap.Logger.debug("Setting Exchange is succesful
","SETTINGSJS",function(m){},function(m){});
successCB();
                                                     },
                                                   function(mesg)
{
sap.Logger.debug("Setting Exchange failed" +
mesg,"SETTINGSJS",function(m){},function(m){});
sap.Settings.isInitialized = false;
errorCB();
```

```
                                                     });
                                  }
                                  , function () {
                                      console.log("unlock failed");
                                 sap.Logger.debug("unlock failed ","SETTINGSJS",function(m){},function(m){});
                                  }
                             );

               }


     },

         /**
          * Function called by the application to register the notification types to receive.
          *
          * @param {string} [types] Types of notifications the application wants to receive. The different types of notifications are expressed in <code>notificationType</code>
         * @param {string} [successCallback] Success callback to call when registration is successful.
        * @param {string} [errorCallback] Error callback to call when registration attempt fails.
       * @param {string} [notificationlistenerfunc] The function that receives the notification for processing by the application.
         * @param {string} [senderId] The sender ID that is used for GCM registration. For other platforms it is null.
          * @private
          * @memberof sap.Push
          * @function registerForNotificationTypes
          * @example
          * regid = "211112269206";
          * function registerSuccess(mesg){}
          * function registerFailure(mesg) {}
          * function ProcessNotification(mesg){}
          * sap.Push.registerForNotificationTypes(sap.Push.notificationType.badge | sap.Push.notificationType.sound | sap.Push.notificationType.alert, registerSuccess, registerFailure, ProcessNotification, regid);
          */

     registerForNotification: function (types, successCallback, errorCallback, notificationListenerFunc, senderId ) {
         if(device.platform == "iPhone" || device.platform == "iPad" || device.platform == "iPod touch" || device.platform == "iOS" || device.platform == "Android") {
             sap.Push.RegisterSuccess = successCallback;
             sap.Push.RegisterFailed  = errorCallback;
             sap.Push.ProcessNotificationForUser = notificationListenerFunc;
             sap.Push.call_native(successCallback, "registerForNotificationTypes", [types, senderId]);

         }

     },

         /* Core APIS */

         /**
          * Function called by the application to register the notification types to receive.
          *
          * @param {string} types Types of notifications the application wants to receive. The different types of notifications are expressed in <code>notificationType</code>
         *              Notificaion types allowed are Disable all notifications (NONE: 0), Set badge count on app icon (BADGE: 1), Play sounds on receiving notification (SOUNDS: 2) and Show alert on receiving notification (ALERT: 4).
```

```
         * @param {string} successCallback Success callback to call when registration is successful.
         * @param {string} errorCallback Error callback to call when registration attempt fails.
         * @param {string} notificationlistenerfunc The function that receives the notification for processing by the application.
         * @param {string} [senderId] The sender ID that is used for GCM registration. For other platforms it is null.
         * @public
         * @memberof sap.Push
         * @function registerForNotificationTypes
         * @example
         * regid = "211112269206";
         * function registerSuccess(mesg){}
         * function registerFailure(mesg) {}
         * function ProcessNotification(mesg){}
         * sap.Push.registerForNotificationTypes(sap.Push.notificationType.badge | sap.Push.notificationType.sound | sap.Push.notificationType.alert, registerSuccess, registerFailure, ProcessNotification, regid);
         */
    registerForNotificationTypes: function (types, successCallback, errorCallback, notificationListenerFunc, senderId ) {
        sap.Push.getConnectionSettings(function(){
                            sap.Logger.debug("getting Connection Settings","PUSHJS",function(m){},function(m){});
                        console.log("getting Connection Settings");
                sap.Push.registerForNotification(types,  successCallback, errorCallback, notificationListenerFunc, senderId );
                                        },
                                        function(){});
    },


        /**
         * Function called by the application to unregister from future notifications.
         *
         * @param {function} callback Success callback to call when deregistration is successful. This callback function will contain a string with a message. This message is for informative purposes only.
         * @public
         * @memberof sap.Push
         * @function unregisterForNotificationTypes
         * @example
         * function unregCallback(mesg){}
         * sap.Push.unregisterForNotificationTypes(unregCallback);
         */

    unregisterForNotificationTypes: function (callbak) {
        if(device.platform == "iPhone" || device.platform == "iPad" || device.platform == "iPod touch" || device.platform == "iOS" || device.platform == "Android") {
sap.Push.call_native(callbak,"unregisterForNotification");
        }
    },

        /**
        * Used to fetch the badge count for the application. This function is used by iOS only. Other platforms do not have the badge count concept.
         *
         * @param {function} callback Success callback to call when to send the badge count. The callback function will contain an argument in JSON format with the current badge count. Look at the example for the details on how to use them.
         * @public
         * @memberof sap.Push
         * @function getBadgeNumber
         * @example
         * function getBadgeNumCallback(data) { badgecount = data["badgecount"];}
         * sap.Push.getBadgeNumber(getBadgeNumCallback);
         */
    getBadgeNumber: function(callback)
        {
            if (sap.Push.isPlatformIOS()) {
                sap.Push.call_native(callback, "getBadgeNumber");
            }
        },

        /**
         * Used to set the badge count for the application. This function is used by iOS only. Other platforms do not have the badge count concept.
         *
         * @param {number} number The badge count to set for the application.
         * @param {function} callback Success callback to call when to send the badge count. The callback function will contain an argument in string format. This argument can be used for informative purposes.
         * @public
         * @memberof sap.Push
         * @function setBadgeNumber
         * @example
         * function badgeCallback(mesg){}
         * badgenum = 10;
         * sap.Push.setBadgeNumber(badgenum, badgeCallback);
         */
    setBadgeNumber: function (number, callback) {
        if (sap.Push.isPlatformIOS()) {
            sap.Push.call_native(callback, "setBadgeNumber", [number]);
        }
    },

        /**
        * Used to reset the badge count for the application.  This function is used by iOS only. Other platforms do not have the badge count concept.
         *
         * @param {function} callback Success callback to call when the badge count is reset. The callback function will contain an argument in string format. This argument can be used for informative purpose.
         * @public
         * @memberof sap.Push
         * @function resetBadge
         * @example
         * function badgeCallback(mesg){}
         * sap.Push.resetBadge(badgeCallback);
         */
    resetBadge: function (callback) {
        if (sap.Push.isPlatformIOS()) {
            sap.Push.call_native(callback, "resetBadge");
        }
    },





        /**
        * This method updates the application with the new device token in the SAP Mobile Platform server.
         *
        * @param {string} [devtok] The device token received from the APNS/GCM device registration.
         * @public
         * @callback {function} [callback] The callback function that is called with the registration result.
         * @memberof sap.Push
         * @example
         * function callback(mesg) {}
         * devToken ="123123213213";//sample device token
         * sap.Push.updateWithDeviceToken(devToken, callback);
         */

    updateWithDeviceToken:  function (devtok, callback) {
        if (sap.Push.isPlatformIOS() || device.platform == "Android" ) {
            sap.Push.call_native(callback, "updateWithDeviceToken", [devtok]);
        }
    },

        /**
        * This method checks for any notifications received while the application was not running in the foreground. You can call this
        * function directly or register with an event handler to be called automatically. It is okay to call this function even if the device is not yet registered for push notifications.
         * @param {function} callback The callback function that receives the notification. The callback function will receive a string as its argument. This string will contain the notification message sent from the server, intact.
         * @memberof sap.Push
         * @example
         * function processBackgroudMessage(mesg){
         *
         * }
         * function checkBackgroundNotification() {
```

```
         * sap.Push.checkForNotification(processBackgroudMessage);
         * }
         * document.addEventListener("onSapLogonSuccess", checkBackgroundNotification, false);
         * document.addEventListener("onSapResumeSuccess", checkBackgroundNotification, false);
         **/

    checkForNotification: function(callback) {
        if (sap.Push.isPlatformIOS() || device.platform == "Android" ) {
            sap.Push.call_native(callback, "checkForNotification");
        }
    },

        /**
       * This is an internal function, which is called when there is a push notification.
         * @private
         **/
    ProcessNotification: function(message) {
        if (sap.Push.ProcessNotificationForUser == null )
        {
            console.log("No Processing function provided");
           sap.Logger.debug("Notification listener function is not registered. Register it by calling registerForNotificationTypes","PUSHJS",function(m){},function(m){});
        } else {
            sap.Push.ProcessNotificationForUser(message);
        }
    },
        /**
        * This is an internal function, which is automatically called when the plugin is initialized. Used for Android only.
        * @private
        **/
    initPlugin: function(callback) {
        if ( device.platform == "Android")
        {
            args = [];
            exec(
                callback,
                function(){ sap.Logger.debug("Plugin Initialization","PUSHJS",function(m){},function(m){}); } ,
                'SMPPushPlugin',
                "initPlugin",
                args);
        }
    }

    };


    /**
     * Local private variables
     */
    module.exports.RegisterSuccess = null;
    module.exports.RegisterFailed = null;
    module.exports.ProcessNotificationForUser = null;
    /**
     * Enum for types of push notification.
     * @enum {number}
     * @private
     */
    module.exports.notificationType = {
        /** Disable all notifications */
    NONE: 0,
        /** Set badge count on app icon */
    BADGE: 1,
        /** Play sounds upon receiving notification */
    SOUNDS: 2,
        /** Show alert upon receiving notification */
    ALERT: 4
    };




    document.addEventListener('deviceready', 
module.exports.initPlugin, false);


```

Using the Kapsel EncryptedStorage Plugin

The EncryptedStorage plugin provides an encrypted local storage mechanism to allow storage of an application's private data on the user's device.

EncryptedStorage Plugin Overview

The Kapsel EncryptedStorage plugin adds an encrypted key/value pair storage option to Cordova, which uses the same API method signature as the browser's local storage option and is non-blocking.

This allows you to store data locally and securely on the device, so that you do not have to retrieve the data from the server every time the application is opened. The user can access and view the data on the device. The data in the encrypted local store is protected by the user's operating system account credentials, so that data cannot be accessessed by anyone who is not logged on as the authenticated user, however, the data stored in local storage is not secure against access by other applications run by the authenticated user, so you should not use encrypted local storage to store sensitive information such as digital rights managment keys or licensing tokens.

Secure storage is an API based on the w3 Web storage API, interface Storage (*http://www.w3.org/TR/2013/PR-webstorage-20130409/#the-storage-interface*.

Note: On Android, you cannot store more than 1MB for a single key/value pair, as the strings are encoded in UTF-8, which means the maximum length of a complex string that can be successfully stored is less than the maximum length of a string with only simple characters (since simple characters are encoded with a single byte, and complex characters are encoded with up to 4 bytes).

Adding the EncryptedStorage Plugin

Install the the EncryptedStorage plugin using the Cordova command line interface.

Prerequisites

- Set up the development environment.
- Create your Cordova Project.
- Add your OS platforms.

Task

1. Add the EncryptedStorage plugin by entering the following at the command prompt, or terminal:

   On Windows:

   ```
   cordova -d plugin add <SDK_HOME>\MobileSDK3\KapselSDK
   \plugins\encryptedstorage
   ```

   On Mac:

   ```
   cordova -d plugin add ~<SDK_HOME>/MobileSDK3/KapselSDK/
   plugins/encryptedstorage
   ```

   Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).
2. (Optional) To see a list of installed plugins in your Cordova project, open a command prompt or terminal window, navigate to your Cordova project folder, and enter:

   ```
   cordova plugins
   ```

   The Cordova command line interface returns a JSON array showing installed plugins, for example:

   ```
   [ 'org.apache.cordova.core.camera',
   'org.apache.cordova.core.device-motion',
   'org.apache.cordova.core.file' ]
   ```

   In this example, the Cordova project has the Cordova core Camera, Accelerator (device-motion), and File plugins installed.
3. Modify the files in the `www` folder for the project as necessary, then copy them to the platform directories by running:

```
cordova -d prepare android
cordova -d prepare ios
```

4. Use the Android IDE or Xcode to deploy and run the project.

Note: If you are using an iOS device, remember to add the "clienthubEntitlements" to the Keychain Groups in the Entitlement section in Xcode.

Kapsel EncryptedStorage API Reference

The Kapsel EncryptedStorage API Reference provides usage information for EncryptedStorage API classes and methods, as well as provides sample source code.

EncryptedStorage namespace

The EncryptedStorage class is used as a secure local store.

The EncryptedStorage API is based on the W3C web storage API, but has two major differences: it is asynchronous, and it has a constructor with a password.

Note: There is a security flaw on some versions of Android with the Pseudo Random Number Generation. The first time the native code of this plugin runs it applies the fix for this issue. However, the fix needs to be applied before any use of Java Cryptography Architecture primitives. Therefore, it is a good idea to run this plugin (call a function that has a native component: length, key, getItem, setItem, removeItem, clear) before using any other security-related plugin, to protect yourself against the possibility that the other plugin does not apply this fix. No other Kapsel plugins are affected, so you need not do this on their behalf. For more details about the security flaw, see *http://android-developers.blogspot.com/2013/08/some-securerandom-thoughts.html*

Adding and Removing the EncryptedStorage Plugin

The EncryptedStorage plugin is added and removed using the *Cordova CLI.*

To add the EncryptedStorage plugin to your project, use the following command:

cordova plugin add <path to directory containing Kapsel plugins>\encryptedstorage

To remove the EncryptedStorage plugin from your project, use the following command:

```
cordova plugin rm com.sap.mp.cordova.plugins.encryptedstorage
```

Members

Name	Description
COMPLEX_STRING_MAXIMUM_LENGTH on page 250	This constant is the length of the largest string that is guaranteed to be successfully stored on Android.
ERROR_BAD_PASSWORD on page 250	This error code indicates that the operation failed due to an incorrect password.
ERROR_GREATER_THAN_MAXIMUM_SIZE on page 251	This error indicates that the string was too large to store.
ERROR_INVALID_PARAMETER on page 251	This error code indicates an invalid parameter was provided. (eg: a string given where a number was required).
ERROR_UNKNOWN on page 251	This error code indicates an unknown error occurred.
SIMPLE_STRING_MAXIMUM_LENGTH on page 251	This constant is the length of the largest string that can successfully be stored on Android.

Methods

Name	Description
clear(successCallback, errorCallback) on page 252	This function removes all items from the store.
getItem(key, successCallback, errorCallback) on page 252	This function gets the value corresponding to the given key.
key(index, successCallback, errorCallback) on page 253	This function gets the key corresponding to the given index.
length(successCallback, errorCallback) on page 254	This function gets the length of the store.
removeItem(key, successCallback, errorCallback) on page 255	This function removes the item corresponding to the given key.
setItem(key, value, successCallback, errorCallback) on page 256	This function sets an item with the given key and value.

Type Definitions

Name	Description
errorCallback(errorCode) on page 257	Callback function that is invoked in case of an error.
getItemSuccessCallback(value) on page 258	
keySuccessCallback(key) on page 258	
lengthSuccessCallback(length) on page 259	
successCallback on page 259	Callback function that is invoked on a successful call to a function that does not need to return anything.

Source

encryptedstorage.js, line 38 on page 261.

COMPLEX_STRING_MAXIMUM_LENGTH member

This constant is the length of the largest string that is guaranteed to be successfully stored on Android.

The limit depends on how many bytes the string takes up when encoded with UTF-8 (under which encoding characters can take up to 4 bytes). This is the maximum length of a string for which every character takes all 4 bytes. Note that this size restriction is present only on Android and not iOS.

Syntax

<constant> `COMPLEX_STRING_MAXIMUM_LENGTH`

Source

encryptedstorage.js, line 306 on page 271.

ERROR_BAD_PASSWORD member

This error code indicates that the operation failed due to an incorrect password.

The password is set by the constructor of EncryptedStorage.

Syntax

<constant> `ERROR_BAD_PASSWORD`

Source

encryptedstorage.js, line 278 on page 270.

ERROR_GREATER_THAN_MAXIMUM_SIZE member

This error indicates that the string was too large to store.

Only applies to Android. For iOS, no hard limit is imposed, but be aware of device memory constraints.

Syntax

<constant> `ERROR_GREATER_THAN_MAXIMUM_SIZE`

Source

encryptedstorage.js, line 286 on page 270.

ERROR_INVALID_PARAMETER member

This error code indicates an invalid parameter was provided. (eg: a string given where a number was required).

Syntax

<constant> `ERROR_INVALID_PARAMETER`

Source

encryptedstorage.js, line 270 on page 270.

ERROR_UNKNOWN member

This error code indicates an unknown error occurred.

Syntax

<constant> `ERROR_UNKNOWN`

Source

encryptedstorage.js, line 263 on page 270.

SIMPLE_STRING_MAXIMUM_LENGTH member

This constant is the length of the largest string that can successfully be stored on Android.

Only if all the characters in the string are encoded in 1 byte in UTF-8 can a string actually be this big. Since characters in UTF-8 can take up to 4 bytes, if you do not know the contents of a string, the maximum length that is guaranteed to be successful is EncryptedStorage.COMPLEX_STRING_MAXIMUM_LENGTH, which is EncryptedStorage.SIMPLE_STRING_MAXIMUM_LENGTH/4. Note that this size restriction is present only on Android and not iOS.

Syntax

<constant> `SIMPLE_STRING_MAXIMUM_LENGTH`

Source

encryptedstorage.js, line 294 on page 271.

clear(successCallback, errorCallback) method

This function removes all items from the store.

If there are no items in the store in the first place, that is still counted as a success.

Syntax

`clear(` *successCallback*, *errorCallback*)

Parameters

Name	Type	Description
successCallback	*sap.EncryptedStorage~successCallback* on page 259	If successful, the successCallback is invoked with no parameters.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 257	If there is an error, the errorCallback is invoked with an ErrorInfo object as the parameter.

Example

```
var store = new sap.EncryptedStorage("storeName", "storePassword");
var successCallback = function() {
   alert("Store cleared!");
}
var errorCallback = function(error) {
   alert("An error occurred: " + JSON.stringify();
}
store.clear(successCallback, errorCallback);
```

Source

encryptedstorage.js, line 229 on page 268.

getItem(key, successCallback, errorCallback) method

This function gets the value corresponding to the given key.

If there is no item with the given key, then the success callback is invoked with null as the parameter.

Syntax

`getItem(` *key*, *successCallback*, *errorCallback*)

Parameters

Name	Type	Description
key	String	The key of the item for which to get the value.If null or undefined is passed, "null" is used.
successCallback	*sap.EncryptedStorage~getItemSuccessCallback* on page 258	If successful, the successCallback is invoked with the value as the parameter (or null if the key did not exist).
errorCallback	*sap.EncryptedStorage~errorCallback* on page 257	If there is an error, the errorCallback is invoked with an ErrorInfo object as the parameter.

Example

```
var store = new sap.EncryptedStorage("storeName", "storePassword");
var successCallback = function(value) {
   alert("Value is " + value);
}
var errorCallback = function(error) {
   alert("An error occurred: " + JSON.stringify(error));
}
store.getItem("theKey", successCallback, errorCallback);
```

Source
encryptedstorage.js, line 118 on page 264.

key(index, successCallback, errorCallback) method
This function gets the key corresponding to the given index.

Syntax
`key(` *index*, *successCallback*, *errorCallback*)

Parameters

Name	Type	Description

index	number	The index of the store for which to get the key. Valid indices are integers from zero (the first index), up to, but not including, the length of the store.If the index is out of bounds, then the success callback is invoked with null as the parameter.
successCallback	*sap.EncryptedStorage~keySuccessCallback* on page 258	If successful, the successCallback is invoked with the key as the parameter.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 257	If there is an error, the errorCallback is invoked with an ErrorInfo object as the parameter.

Example

```
// This example shows how to get the key for the last item.
var store = new sap.EncryptedStorage("storeName", "storePassword");
var errorCallback = function( error ){
   alert("An error occurred: " + JSON.stringify(error));
}
var keySuccessCallback = function(key) {
   alert("Last key is " + key);
}
var lengthSuccessCallback = function(length) {
   store.key(length - 1, keySuccessCallback, errorCallback);
}
store.length(lengthSuccessCallback, errorCallback);
```

Source

encryptedstorage.js, line 79 on page 263.

length(successCallback, errorCallback) method

This function gets the length of the store.

The length of a store is the number of key/value pairs that are in the store.

Syntax

`length`(*successCallback*, *errorCallback*)

Parameters

Name	Type	Description

successCallback	*sap.EncryptedStorage~lengthSuccessCallback* on page 259	If successful, the successCallback is invoked with the length of the store as the parameter.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 257	If there is an error, the errorCallback is invoked with an ErrorInfo object as the parameter.

Example

```
var store = new sap.EncryptedStorage("storeName", "storePassword");
var successCallback = function(length) {
   alert("Length is " + length);
}
var errorCallback = function(error) {
   alert("An error occurred: " + JSON.stringify(error));
}
store.length(successCallback, errorCallback);
```

Source

encryptedstorage.js, line 46 on page 261.

removeItem(key, successCallback, errorCallback) method

This function removes the item corresponding to the given key.

If there is no item with the given key in the first place, that is still counted as a success.

Syntax

`removeItem(` *key*, *successCallback*, *errorCallback*)

Parameters

Name	Type	Description
key	String	The key of the item to remove.If null or undefined is passed, "null" is used.
successCallback	*sap.EncryptedStorage~successCallback* on page 259	If successful, the successCallback is invoked with no parameters.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 257	If there is an error, the errorCallback is invoked with an ErrorInfo object as the parameter.

Example

```
var store = new sap.EncryptedStorage("storeName", "storePassword");
var successCallback = function() {
   alert("Value removed");
}
var errorCallback = function(error) {
   alert("An error occurred: " + JSON.stringify(error));
}
store.removeItem("somekey", successCallback, errorCallback);
```

Source

encryptedstorage.js, line 195 on page 267.

setItem(key, value, successCallback, errorCallback) method

This function sets an item with the given key and value.

If no item exists with the given key, then a new item is created. If an item does exist with the the given key, then its value is overwritten with the given value.

Note: On Android there is a size limit on the string to be stored. See *sap.EncryptedStorage#SIMPLE_STRING_MAXIMUM_LENGTH* on page 251 and *sap.EncryptedStorage#COMPLEX_STRING_MAXIMUM_LENGTH* on page 250 for more details.

Syntax

`setItem(` *key*, *value*, *successCallback*, *errorCallback*)

Parameters

Name	Type	Description
key	String	The key of the item to set.If null or undefined is passed, "null" is used.
value	String	The value of the item to set.If null or undefined is passed, "null" is used.
successCallback	*sap.EncryptedStorage~successCallback* on page 259	If successful, the successCallback is invoked with no parameters.

errorCallback	*sap.EncryptedStorage~errorCallback* on page 257	If there is an error, the errorCallback is invoked with an ErrorInfo object as the parameter.

Example

```
var store = new sap.EncryptedStorage("storeName", "storePassword");
var successCallback = function() {
   alert("Item has been set.");
}
var errorCallback = function(error) {
   alert("An error occurred: " + JSON.stringify(error));
}
store.setItem("somekey", "somevalue", successCallback,
errorCallback);
```

Source

encryptedstorage.js, line 154 on page 265.

errorCallback(errorCode) type

Callback function that is invoked in case of an error.

Syntax

`errorCallback(` *errorCode*)

Parameters

Name	Type	Description
errorCode	number	An error code indicating what went wrong.Will be one of *sap.EncryptedStorage#ERROR_UNKNOWN* on page 251, *sap.EncryptedStorage#ERROR_INVALID_PARAMETER* on page 251, *sap.EncryptedStorage#ERROR_BAD_PASSWORD* on page 250, or *sap.EncryptedStorage#ERROR_GREATER_THAN_MAXIMUM_SIZE* on page 251.

Example

```
function errorCallback(errCode) {
   //Set the default error message. Used if an invalid code is passed
to the
```

```
   //function (just in case) but also to cover the
   //sap.EncryptedStorage.ERROR_UNKNOWN case as well.
   var msg = "Unkown Error";
   switch (errCode) {
      case sap.EncryptedStorage.ERROR_INVALID_PARAMETER:
         msg = "Invalid parameter passed to method";
         break;
      case sap.EncryptedStorage.ERROR_BAD_PASSWORD :
         msg = "Incorrect password";
         break;
      case sap.EncryptedStorage.ERROR_GREATER_THAN_MAXIMUM_SIZE:
         msg = "Item (string) value too large to write to store";
         break;
   };
   //Write the error to the log
   console.error(msg);
   //Let the user know what happened
  navigator.notification.alert(msg, null, "EncryptedStorage Error",
"OK");
};
```

Source
encryptedstorage.js, line 328 on page 272.

getItemSuccessCallback(value) type

Callback function that is invoked on a successful call to EncryptedStorage.getItem. If the returned value is null, that means the key passed to EncryptedStorage.getItem did not exist.

Syntax
`getItemSuccessCallback(` *value*)

Parameters

Name	Type	Description
value	String	The value of the item with the given key.Will be null if the key passed to EncryptedStorage.getItem did not exist.

Source
encryptedstorage.js, line 326 on page 272.

keySuccessCallback(key) type

Callback function that is invoked on a successful call to EncryptedStorage.key. If the key returned is null that means the index passed to EncryptedStorage.key was out of bounds.

Syntax
`keySuccessCallback(` *key* `)`

Parameters

Name	Type	Description
key	String	The key corresponding to the given index.Will be null if the index passed to EncryptedStorage.key was out of bounds.

Source
encryptedstorage.js, line 324 on page 272.

lengthSuccessCallback(length) type

Callback function that is invoked on a successful call to EncryptedStorage.length.

Syntax
`lengthSuccessCallback(` *length* `)`

Parameters

Name	Type	Description
length	number	The number of key/value pairs in the store.

Source
encryptedstorage.js, line 322 on page 272.

successCallback type
Callback function that is invoked on a successful call to a function that does not need to return anything.

Syntax
`successCallback()`

Source
encryptedstorage.js, line 320 on page 272.

Source code

encryptedstorage.js

```
// ${project.version}
var argscheck = require('cordova/argscheck'),
    exec = require("cordova/exec");

/**
 * The EncryptedStorage class is used as a secure local store.  The EncryptedStorage API is based on the
 * W3C Web storage API, but has two major differences--it is asynchronous, and it has a constructor with
 * a password.<br/>
 * <br/>
 * <b>Note:</b> There is a security flaw on some versions of Android with the Pseudo Random Number Generation.
 * The first time the native code of this plugin runs it applies the fix for this issue.  However, the
 * fix needs to be applied before any use of Java Cryptography Architecture primitives.  Therefore, it
 * is a good idea to run this plugin, for example, call a function that has a native component such as length, key, getItem,
 * setItem, removeItem, or clear, before using any other security-related plugin, to protect yourself
 * against the possibility that the other plugin does not apply this fix. This issue affects only the EncryptedStorage plugin,
 *so you need not do this for other Kapsel plugins.  For more details about the security flaw, see
 * <a href="http://android-developers.blogspot.com/2013/08/some-securerandom-thoughts.html">
 * http://android-developers.blogspot.com/2013/08/some-securerandom-thoughts.html</a><br/>
 * <br/>
 * <b>Adding and Removing the EncryptedStorage Plugin</b><br/>
 * Add or remove the EncryptedStorage plugin by using the
 * <a href="http://cordova.apache.org/docs/en/edge/guide_cli_index.md.html#The%20Command-line%20Interface">Cordova CLI</a>.<br/>
```

```
     * <br/>
    * To add the EncryptedStorage plugin to your project, use the following command:<br/>
     * cordova plugin add <path to directory containing Kapsel plugins>\encryptedstorage<br/>
     * <br/>
     * To remove the EncryptedStorage plugin from your project, use the following command:<br/>
     * cordova plugin rm com.sap.mp.cordova.plugins.encryptedstorage
     * @namespace
     * @alias EncryptedStorage
     * @memberof sap
    * @param {String} storeName The name of the store to create.  All stores with different names
     * act independently, while stores with the same name (and same password) act as the same store.
     * If null or undefined is passed, an empty string is used.
     * @param {String} password The password of the store to create.  If null or undefined is passed,
     * an empty string is used.
     */
    EncryptedStorage = function (storeName, password) {
        // private variables
        var that = this;
        var storagePassword = password ? password : "";
        var storageName = storeName ? storeName : "";

        // privileged functions

        /**
       * This function gets the length of the store.  The length of a store
        * is the number of key/value pairs that are in the store.
     * @param {sap.EncryptedStorage~lengthSuccessCallback} successCallback If successful,
     * the successCallback is invoked with the length of the store as
     * the parameter.
     * @param {sap.EncryptedStorage~errorCallback} errorCallback If there is an error,
     * the errorCallback is invoked with an ErrorInfo object as the parameter.
     * @memberof sap.EncryptedStorage
     * @function length
     * @instance
     * @example
     * var store = new sap.EncryptedStorage("storeName", "storePassword");
     * var successCallback = function(length) {
     *    alert("Length is " + length);
     * }
     * var errorCallback = function(error) {
     *    alert("An error occurred: " + JSON.stringify(error));
     * }
     * store.length(successCallback, errorCallback);
     */
    this.length = function (successCallback, errorCallback) {
        try{
            argscheck.checkArgs('FF', 'EncryptedStorage.length', arguments);
        }catch(ex){
            errorCallback(this.ERROR_INVALID_PARAMETER);
            return;
        }

        cordova.exec(successCallback, errorCallback, "EncryptedStorage",
            "length", [storageName, storagePassword]);
     }

     /**
     * This function gets the key corresponding to the given index.
    * @param {number} index The index of the store for which to get the key.
     * Valid indices are integers from zero (the first index), up to, but not including,
     * the length of the store.  If the index is out of bounds, then the success
     * callback is invoked with null as the parameter.
     * @param {sap.EncryptedStorage~keySuccessCallback} successCallback If successful,
     * the successCallback is invoked with the key as the parameter.
     * @param {sap.EncryptedStorage~errorCallback} errorCallback If there is an error,
     * the errorCallback is invoked with an ErrorInfo object as the parameter.
     * @memberof sap.EncryptedStorage
     * @function key
     * @instance
     * @example
     * // This example shows how to get the key for the last item.
     * var store = new sap.EncryptedStorage("storeName", "storePassword");
     * var errorCallback = function( error ){
     *    alert("An error occurred: " + JSON.stringify(error));
     * }
     * var keySuccessCallback = function(key) {
     *    alert("Last key is " + key);
     * }
     * var lengthSuccessCallback = function(length) {
```

```
     *    store.key(length - 1, keySuccessCallback, errorCallback);
     * }
     * store.length(lengthSuccessCallback, errorCallback);
     */
    this.key = function (index, successCallback, errorCallback) {
        try{
         argscheck.checkArgs('NFF', 'EncryptedStorage.key', arguments);
        }catch(ex){
            errorCallback(this.ERROR_INVALID_PARAMETER);
            return;
        }

        cordova.exec(successCallback, errorCallback, "EncryptedStorage",
            "key", [storageName, storagePassword, index]);
    }

    /**
    * This function gets the value corresponding to the given key.  If there is no
     * item with the given key, then the success callback is invoked with null as
     * the parameter.
    * @param {String} key The key of the item for which to get the value. If null or undefined is
     * passed, "null" is used.
     * @param {sap.EncryptedStorage~getItemSuccessCallback} successCallback If successful,
     * the successCallback is invoked with the value as the parameter (or null if the key
     * did not exist).
     * @param {sap.EncryptedStorage~errorCallback} errorCallback If there is an error,
    * the errorCallback is invoked with an ErrorInfo object as the parameter.
     * @memberof sap.EncryptedStorage
     * @function getItem
     * @instance
     * @example
     * var store = new sap.EncryptedStorage("storeName", "storePassword");
     * var successCallback = function(value) {
     *    alert("Value is " + value);
     * }
     * var errorCallback = function(error) {
     *    alert("An error occurred: " + JSON.stringify(error));
     * }
     * store.getItem("theKey", successCallback, errorCallback);
     */
    this.getItem = function (key, successCallback, errorCallback) {
        try{
            argscheck.checkArgs('SFF', 'EncryptedStorage.getItem', arguments);
        }catch(ex){
            errorCallback(this.ERROR_INVALID_PARAMETER);
            return;
        }

        cordova.exec(successCallback, errorCallback, "EncryptedStorage",
            "getItem", [storageName, storagePassword, key]);
    }

    /**
```

```
         * This function sets an item with the given key and value.  If no item exists with
         * the given key, a new item is created. If an item does exist with the
         * the given key, its value is overwritten with the given value.<br/>
         * <br/>
         * Note: On Android there is a size limit on the string to be stored.  See
         * {@link sap.EncryptedStorage#SIMPLE_STRING_MAXIMUM_LENGTH} and {@link sap.EncryptedStorage#COMPLEX_STRING_MAXIMUM_LENGTH}
         * for more details.
         * @param {String} key The key of the item to set.  If null or undefined is passed,
         * "null" is used.
         * @param {String} value The value of the item to set.  If null or undefined is passed,
         * "null" is used.
         * @param {sap.EncryptedStorage~successCallback} successCallback If successful,
         * the successCallback is invoked with no parameters.
         * @param {sap.EncryptedStorage~errorCallback} errorCallback If there is an error,
         * the errorCallback is invoked with an ErrorInfo object as the parameter.
         * @memberof sap.EncryptedStorage
         * @function setItem
         * @instance
         * @example
         * var store = new sap.EncryptedStorage("storeName", "storePassword");
         * var successCallback = function() {
         *    alert("Item has been set.");
         * }
         * var errorCallback = function(error) {
```

```
         *    alert("An error occurred: " + JSON.stringify(error));
         * }
        * store.setItem("somekey", "somevalue", successCallback, errorCallback);
         */
        this.setItem = function (key, value, successCallback, errorCallback) {
            try{
                argscheck.checkArgs('SSFF', 'EncryptedStorage.setItem', arguments);
            }catch(ex){
                errorCallback(this.ERROR_INVALID_PARAMETER);
                return;
            }

            cordova.exec(successCallback, errorCallback, "EncryptedStorage",
                "setItem", [storageName, storagePassword, key, value]);
        }

        /**
         * This function removes the item corresponding to the given key. If there is no
         * item with the given key in the first place, that is still counted as a success.
        * @param {String} key The key of the item to remove.  If null or undefined is
         * passed, "null" is used.
         * @param {sap.EncryptedStorage~successCallback} successCallback If successful,
         * the successCallback is invoked with no parameters.
         * @param {sap.EncryptedStorage~errorCallback} errorCallback If there is an error,
        * the errorCallback is invoked with an ErrorInfo object as the parameter.
```

```
         * @memberof sap.EncryptedStorage
         * @function removeItem
         * @instance
         * @example
         * var store = new sap.EncryptedStorage("storeName", "storePassword");
         * var successCallback = function() {
         *    alert("Value removed");
         * }
         * var errorCallback = function(error) {
         *    alert("An error occurred: " + JSON.stringify(error));
         * }
         * store.removeItem("somekey", successCallback, errorCallback);
         */
        this.removeItem = function (key, successCallback, errorCallback) {
            try{
                argscheck.checkArgs('SFF', 'EncryptedStorage.removeItem', arguments);
            }catch(ex){
                errorCallback(this.ERROR_INVALID_PARAMETER);
                return;
            }

            cordova.exec(successCallback, errorCallback, "EncryptedStorage",
                "removeItem", [storageName, storagePassword, key]);
        }

        /**
       * This function removes all items from the store. If there are no
```

```
     * items in the store, it is still counted as a success.
     * @param {sap.EncryptedStorage~successCallback} successCallback If successful,
     * the successCallback is invoked with no parameters.
     * @param {sap.EncryptedStorage~errorCallback} errorCallback If there is an error,
     * the errorCallback is invoked with an ErrorInfo object as the parameter.
     * @memberof sap.EncryptedStorage
     * @function clear
     * @instance
     * @example
     * var store = new sap.EncryptedStorage("storeName", "storePassword");
     * var successCallback = function() {
     *    alert("Store cleared!");
     * }
     * var errorCallback = function(error) {
     *    alert("An error occurred: " + JSON.stringify();
     * }
     * store.clear(successCallback, errorCallback);
     */
    this.clear = function (successCallback, errorCallback) {
        try{
            argscheck.checkArgs('FF', 'EncryptedStorage.clear', arguments);
        }catch(ex){
            errorCallback(this.ERROR_INVALID_PARAMETER);
            return;
        }

        cordova.exec(successCallback, errorCallback, "EncryptedStorage",
                 "clear", [storageName, storagePassword]);
         }
     };

     // Error codes
     /**
      * This error code indicates an unknown error occurred.
      * @memberof sap.EncryptedStorage
      * @name sap.EncryptedStorage#ERROR_UNKNOWN
      * @constant
      */
     EncryptedStorage.prototype.ERROR_UNKNOWN = 0;
     /**
      * This error code indicates an invalid parameter was 
provided.
      * (eg: a string given where a number was required).
      * @memberof sap.EncryptedStorage
      * @name sap.EncryptedStorage#ERROR_INVALID_PARAMETER
      * @constant
      */
     EncryptedStorage.prototype.ERROR_INVALID_PARAMETER = 1;
     /**
     * This error code indicates that the operation failed due to 
an incorrect password.  The password is
      * set by the constructor of {@link EncryptedStorage}.
      * @memberof sap.EncryptedStorage
      * @name sap.EncryptedStorage#ERROR_BAD_PASSWORD
      * @constant
      */
     EncryptedStorage.prototype.ERROR_BAD_PASSWORD = 2;
     /**
      * This error indicates that the string was too large to 
store. This error applies only to Android.
```

```
     * For iOS, no hard limit is imposed, but be aware of device memory constraints.
     * @memberof sap.EncryptedStorage
     * @name sap.EncryptedStorage#ERROR_GREATER_THAN_MAXIMUM_SIZE
     * @constant
     */
   EncryptedStorage.prototype.ERROR_GREATER_THAN_MAXIMUM_SIZE = 3;
    /**
     * This constant is the length of the largest string that can successfully be stored on Android.  Only if all the
    * characters in the string are encoded in 1 byte in UTF-8 can a string actually be this big.  Since
     * characters in UTF-8 can take up to 4 bytes, if you do not know the contents of a string, the maximum
     * length that is guaranteed to be successful is {@link EncryptedStorage.COMPLEX_STRING_MAXIMUM_LENGTH}, which is
     * {@link EncryptedStorage.SIMPLE_STRING_MAXIMUM_LENGTH}/4.  This size restriction is present only on
     * Android.
     * @memberof sap.EncryptedStorage
     * @name sap.EncryptedStorage#SIMPLE_STRING_MAXIMUM_LENGTH
     * @constant
     */
    EncryptedStorage.prototype.SIMPLE_STRING_MAXIMUM_LENGTH = 1048542;
    /**
     * This constant is the length of the largest string that is guaranteed to be successfully stored on Android.  The
     * limit depends on how many bytes the string takes up when encoded with UTF-8 (under which encoding
     * characters can take up to 4 bytes).  This is the maximum length of a string for which every character
     * takes all 4 bytes.  This size restriction is present only on Android.
     * @memberof sap.EncryptedStorage
```

```
 * @name sap.EncryptedStorage#COMPLEX_STRING_MAXIMUM_LENGTH
 * @constant
 */
EncryptedStorage.prototype.COMPLEX_STRING_MAXIMUM_LENGTH = 262135;

module.exports = EncryptedStorage;


/**
* Callback function that is invoked on a successful call to a function that does
 * not need to return anything.
 *
 * @callback sap.EncryptedStorage~successCallback
 */

/**
 * Callback function that is invoked on a successful call to {@link EncryptedStorage.length}.
 *
 * @callback sap.EncryptedStorage~lengthSuccessCallback
 *
* @param {number} length The number of key/value pairs in the store.
 */

/**
 * Callback function that is invoked on a successful call to {@link EncryptedStorage.key}.
* If the key returned is null, that means the index passed to {@link EncryptedStorage.key} was out of bounds.
 *
 * @callback sap.EncryptedStorage~keySuccessCallback
```

```
     *
     * @param {String} key The key corresponding to the given index.  Will be null if the index passed to
     * {@link EncryptedStorage.key} was out of bounds.
     */

    /**
     * Callback function that is invoked on a successful call to {@link EncryptedStorage.getItem}.
     * If the returned value is null, that means the key passed to {@link EncryptedStorage.getItem} did not exist.
     *
     * @callback sap.EncryptedStorage~getItemSuccessCallback
     *
     * @param {String} value The value of the item with the given key.  Will be null if the key passed to
     * {@link EncryptedStorage.getItem} did not exist.
     */

    /**
     * Callback function that is invoked in case of an error.
     *
     * @callback sap.EncryptedStorage~errorCallback
     *
     * @param {number} errorCode An error code indicating what went wrong.  Will be one of {@link sap.EncryptedStorage#ERROR_UNKNOWN},
     * {@link sap.EncryptedStorage#ERROR_INVALID_PARAMETER}, {@link sap.EncryptedStorage#ERROR_BAD_PASSWORD}, or
     * {@link sap.EncryptedStorage#ERROR_GREATER_THAN_MAXIMUM_SIZE}.
     *
     * @example
     * function errorCallback(errCode) {
     *    //Set the default error message. Used if an invalid code is passed to the
```

```
 *    //function (just in case) but also to cover the
 *    //sap.EncryptedStorage.ERROR_UNKNOWN case as well.
 *    var msg = "Unkown Error";
 *    switch (errCode) {
 *       case sap.EncryptedStorage.ERROR_INVALID_PARAMETER:
 *          msg = "Invalid parameter passed to method";
 *          break;
 *       case sap.EncryptedStorage.ERROR_BAD_PASSWORD :
 *          msg = "Incorrect password";
 *          break;
 *       case sap.EncryptedStorage.ERROR_GREATER_THAN_MAXIMUM_SIZE:
 *          msg = "Item (string) value too large to write to store";
 *          break;
 *    };
 *    //Write the error to the log
 *    console.error(msg);
 *    //Let the user know what happened
 *    navigator.notification.alert(msg, null, "EncryptedStorage Error", "OK");
 * };
 */
```

Using the Kapsel Settings Plugin

Use the Settings plugin to trigger an operation with the SAP Mobile Platform Server that allows an application to store device and user settings for later use.

Settings Plugin Overview

The Settings plugin exchanges application connection settings with the server settings.

If application settings such as log level and log upload mode are changed on the server, the Settings plugin synchronizes the information with the Kapsel application. Since some of that information is used in the Push plugin, the Push plugin requires the Settings plugin.

The APIs for the Settings plugin allow device and user settings to be stored on the device to make a connection with the SAP Mobile Platform Server. The client sends the server the

DeviceType, DeviceModel, PushEnabled, and other push-related statuses. The settings also use the device token that is received during device registration. The server uses this information to determine whether to send a GCM or APNS push notification.

Adding the Settings Plugin

To install the Settings plugin, use the Cordova command line interface.

Prerequisites

- Set up the development environment.
- Create your Cordova Project.
- Add your OS platforms.

Task

The Settings plugin has dependencies on the Logger plugin, so when you install the Settings plugin, the Logger plugin is added automatically.

1. Add the Settings plugin by entering the following at the command prompt, or terminal:

   On Windows:

   ```
   cordova -d plugin add <SDK_HOME>\MobileSDK3\KapselSDK
   \plugins\settings
   ```

   On Mac:

   ```
   cordova -d plugin add ~<SDK_HOME>/MobileSDK3/KapselSDK/
   plugins/settings
   ```

   Note: The path you enter to the Kapsel plugin must be the absolute path (not relative path).

2. (Optional) To see a list of installed plugins in your Cordova project, open a command prompt or terminal window, navigate to your Cordova project folder, and enter:

   ```
   cordova plugins
   ```

   The Cordova command line interface returns a JSON array showing installed plugins, for example:

   ```
   [ 'org.apache.cordova.core.camera',
   'org.apache.cordova.core.device-motion',
   'org.apache.cordova.core.file' ]
   ```

   In this example, the Cordova project has the Cordova core Camera, Accelerator (device-motion), and File plugins installed.

3. Modify the files in the `www` folder for the project as necessary, then copy them to the platform directories by running:

   ```
   cordova -d prepare android
   cordova -d prepare ios
   ```

4. Use the Android IDE or Xcode to deploy and run the project.

Note: If you are using an iOS device, remember to add the "clienthubEntitlements" to the Keychain Groups in the Entitlement section in Xcode.

Kapsel Settings API Reference

The Kapsel Settings API Reference provides usage information for Settings API classes and methods, as well as provides sample source code.

Settings namespace

Provides settings exchange functionality

Methods

Name	Description
start(connectionData, successCallback, errorCallback) on page 276	Starts the settings exchange.

Source

settings.js, line 12 on page 277.

start(connectionData, successCallback, errorCallback) method

Starts the settings exchange.

Syntax

<static> `start(` *connectionData*, *successCallback*, *errorCallback*)

Parameters

Name	Type	Description
connectionData	String	This example below shows the structure of the connection data.
successCallback	function	Function to invoke if the exchange is successful.
errorCallback	function	Function to invoke if the exchange failed.

Example

```
connectionData = {
     "keyMAFLogonOperationContextConnectionData": {
     "keyMAFLogonConnectionDataApplicationSettings":
     {
     "DeviceType":device.platform,
     "DeviceModel":device.model,
     "ApplicationConnectionId":"yourappconnectionid"
     },
     "keyMAFLogonConnectionDataBaseURL":"servername:port"
 },
 "keyMAFLogonOperationContextApplicationId":"yourapplicationid",
 "keyMAFLogonOperationContextBackendUserName":"yourusername",
 "keyMAFLogonOperationContextBackendPassword":"password",
 "keyMAFLogonOperationContextSecurityConfig":"securityConfigName",
 "keySSLEnabled":keySSLEnabled
 };
sap.Settings.start(connectionData, function(mesg) {

                                 sap.Logger.debug("Setting Exchange
is successful "+mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
                                    },
                                   function(mesg){
                                 sap.Logger.debug("Setting Exchange
failed" + mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
                                   });
```

Source
settings.js, line 96 on page 281.

Source code

settings.js

```
// ${project.version}
var exec = require("cordova/exec");


/**
 * The Kapsel Settings plugin provides settings exchange functionality.
 *
 * @namespace
 * @alias Settings
 * @memberof sap
 */
var SettingsExchange = function () {};

SettingsExchange.prototype.connectionData   = null;
SettingsExchange.prototype.store            = null;
SettingsExchange.prototype.settingsSuccess  = null;
SettingsExchange.prototype.SettingsError    = null;
SettingsExchange.prototype.isInitialized    = false;


/**
* Starts the settings exchange process upon onSapLogonSuccess event.
 * @private
 */
var doSettingExchange = function () {


    sap.Settings.isInitialized = true;
    var pd ="";
    sap.Logon.unlock(function (connectionInfo) {
                     var userName = connectionInfo["registrationContext"]["user"];
                     var password  = connectionInfo["registrationContext"]["password"];
                     var applicationConnectionId = connectionInfo["applicationConnectionId"];
                     var securityConfig = connectionInfo["registrationContext"]["securityConfig"];
                     var endpoint = connectionInfo["applicationEndpointURL"];
                     var keySSLEnabled = "false";
                     var splitendpoint = endpoint.split("/");
                     if (splitendpoint[0] == "https:")
                     {
                         keySSLEnabled="true";
                     }
                     if (securityConfig == null) {
                         securityConfig = "";
```

```
                        }
                        var burl = splitendpoint[2];
                        var appId = splitendpoint[3];
                        pd = appId+userName+password;
                        sap.Settings.store = new sap.EncryptedStorage("SettingsStore", pd);
                        connectionData = {
                            "keyMAFLogonOperationContextConnectionData": {
                                "keyMAFLogonConnectionDataApplicationSettings":
                                {
                                    "DeviceType":device.platform,
                                    "DeviceModel":device.model,
                                    "ApplicationConnectionId":applicationConnectionId
                                },
                                "keyMAFLogonConnectionDataBaseURL":burl
                            },
                            "keyMAFLogonOperationContextApplicationId":appId,
                            "keyMAFLogonOperationContextBackendUserName":userName,
                            "keyMAFLogonOperationContextBackendPassword":password,
                            "keyMAFLogonOperationContextSecurityConfig":securityConfig,
                            "keySSLEnabled":keySSLEnabled
                        };
                        sap.Settings.start(connectionData,
                                           function(mesg) {
                                               sap.Settings.isInitialized = true;
```

```
                                                sap.Logger.debug("Setting Exchange is successful "+mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
                                            },
                                            function(mesg){
                                                sap.Logger.debug("Setting Exchange failed" + mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
                                                sap.Settings.isInitialized = false;
                                            });
                         }
                         , function () {
                         sap.Logger.debug("unlock failed ","SMP_SETTINGS_JS",function(m){},function(m){});
                         }
                         );


    };

    document.addEventListener("onSapLogonSuccess", doSettingExchange, false);
    document.addEventListener("onSapResumeSuccess", doSettingExchange, false);

    SettingsExchange.prototype.reset = function(key, sucessCB, errorCB)
    {
        if ((typeof(sap.Settings.store) != undefined) && (sap.Settings.store != null)) {
            sap.Settings.store.removeItem(key, sucessCB, errorCB);
        } else {
            errorCB("Cannot access setting store");
        }
    }



    /**
     * Starts the settings exchange.
     * @public
     * @memberof sap.Settings
     * @method start
    * @param {String} connectionData The example below shows the structure of the connection data.
    * @param {function} successCallback Function to invoke if the exchange is successful.
     * @param {function} errorCallback Function to invoke if the exchange failed.
     * @example
     * connectionData = {
     *      "keyMAFLogonOperationContextConnectionData": {
     *      "keyMAFLogonConnectionDataApplicationSettings":
     *      {
     *      "DeviceType":device.platform,
     *      "DeviceModel":device.model,
     *      "ApplicationConnectionId":"yourappconnectionid"
     *      },
     * "keyMAFLogonConnectionDataBaseURL":"servername:port"
     *  },
     * "keyMAFLogonOperationContextApplicationId":"yourapplicationid",
     * "keyMAFLogonOperationContextBackendUserName":"yourusername",
     * "keyMAFLogonOperationContextBackendPassword":"password",
     * "keyMAFLogonOperationContextSecurityConfig":"securityConfigName",
     *  "keySSLEnabled":keySSLEnabled
     *  };
     * sap.Settings.start(connectionData, function(mesg) {
     *
     * sap.Logger.debug("Setting Exchange is successful "+mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
     *                                     },
     *                                    function(mesg){
    *                                 sap.Logger.debug("Setting Exchange failed" + mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
     *                                    });
     */
   SettingsExchange.prototype.start = function (connectionData, successCallback, errorCallback) {
        sap.Settings.settingsSuccess = successCallback;
        sap.Settings.SettingsError = errorCallback;
        sap.Settings.connectionData = connectionData;
        sap.Logger.debug("Accessing the data from vault","SMP_SETTINGS_JS",function(m){},function(m){});
        sap.Settings.store.getItem("settingsdata", sap.Settings.getStoreDataSuccess, sap.Settings.getStoreDataError);


    };







    /**
     *  This is a private function. End user will not use this plugin directly.
    *  This function gets called after the start function is able to read the current settings from the secured storage.
     *  @private
     *  @param {String} value This is the value of the current setting exchange stored in the secured store.
     **/

    SettingsExchange.prototype.getStoreDataSuccess  = function(value){
        storedSettings = value;
        sap.Logger.debug("Exchanging the data","SMP_SETTINGS_JS",function(m){},function(m){});
        exec(sap.Settings.SettingsExchangeDone,
             sap.Settings.SettingsExchangeError,
             "SMPSettingsExchangePlugin",
             "start", [JSON.stringify(connectionData),storedSettings]);
    }

    /**
     *  This is a private function. End user will not use this plugin directly.
    *  This function is called after the start function is unable to read the current settings from the secured storage.
     *  @private
    *  @param {String} message This is the error message produced by the encrypted storage.
     **/
    SettingsExchange.prototype.getStoreDataError  = function(mesage){
        sap.Logger.debug("Setting exchange failed to read data store: Proceeding without data",function(m){},function(m){});
    }


    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called after the settings exchange completes succefully.
     *  @private
    *  @param {String} message This is the  message produced when the settings plugin completes successfully.
     **/

    SettingsExchange.prototype.SettingsExchangeDone = function(message) {
        sap.Logger.debug("Setting Exchange Success","SMP_SETTINGS_JS",function(m){},function(m){});
        var jsondata =  JSON.parse(message);
        settingsString = JSON.stringify(jsondata["data"]);
        sap.Settings.store.setItem("settingsdata",settingsString,sap.Settings.SettingsWriteDone,sap.Settings.SettingsWriteError);
        if (sap.Settings.settingsSuccess != null) {
            sap.Logger.debug("Setting exchange successful","SMP_SETTINGS_JS",function(m){},function(m){});
            sap.Settings.settingsSuccess(jsondata["msg"]);
        }
    }

    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called after the settings exchange completes succefully
     *  @private
    *  @param {String} message This is the error message produced when the settings plugin has an error.
     **/
    SettingsExchange.prototype.SettingsExchangeError = function(message) {
        sap.Logger.error("Setting Exchange failed calling the error callback funciton","SMP_SETTINGS_JS",function(m){},function(m){});
        if (sap.Settings.SettingsError != null) {
            sap.Settings.SettingsError(message);
        }
    }

    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called after the setting data is stored successfully.
     *  @private
     *  @param {String} message This is the message produced upon successful storing of settings to the encrypted store.
     **/
    SettingsExchange.prototype.SettingsWriteDone  = function(message) {
        sap.Logger.debug("Setting stored","SMP_SETTINGS_JS",function(m){},function(m){});

    }

    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called after the storing of the setting data fails.
     *  @private
     *  @param {String} message This is the message produced upon failure to store the settings to the encrypted store.
     **/
    SettingsExchange.prototype.SettingsWriteError  = function(message) {
        sap.Logger.error("Setting store failed","SMP_SETTINGS_JS",function(m){},function(m){});
    }

    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called after the deviceready. This uploads the logs to the server.
     *  @private
     *  @param {boolean} uploadLog This indicates whether the upload log is currently enabled or disbled.
     **/
    SettingsExchange.prototype.logLevelUpdated  = function(logLevel)
    {
        sap.Logger.setLogLevel(logLevel, sap.Settings.LogLevelSetSuccess, sap.Settings.LogLevelSetFailed);
        sap.Logger.upload(sap.Settings.logUploadedSuccess, sap.Settings.logUploadFailed);
    }

    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called when the log upload succeeds.
     *  @private
     *  @param {mesg} logupload message
     **/
    SettingsExchange.prototype.LogLevelSetSuccess = function(mesg){
        sap.Logger.debug("Log level set successful","SMP_SETTINGS_JS",function(m){},function(m){});
    }
    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called when the log upload succeeds.
     *  @private
     *  @param {mesg} logupload message
     **/
    SettingsExchange.prototype.LogLevelSetFailed = function(mesg){
        sap.Logger.error("Log level set failed","SMP_SETTINGS_JS",function(m){},function(m){});
    }

    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called when the log upload succeeds.
     *  @private
     *  @param {mesg} logupload message
     **/
    SettingsExchange.prototype.logUploadedSuccess = function(mesg){
        sap.Logger.debug("Log upload successful","SMP_SETTINGS_JS",function(m){},function(m){});
    }
    /**
     *  This is a private function. End user will not use this plugin directly.
     *  This function is called when the log upload fails.
     *  @private
     *  @param {mesg} logupload failure message
     **/
    SettingsExchange.prototype.logUploadFailed = function(mesg){
        sap.Logger.error("upload log failed","SMP_SETTINGS_JS",function(m){},function(m){});

    }



    module.exports = new SettingsExchange();


```

Developing a Kapsel Application With OData Online

Creating an OData Application

Create an OData application in the Management Cockpit.

1. In the Management Cockpit home page, click Settings to define your application's security settings.
2. In the Edit Security Profile dialog, click **New**.
3. Enter a name for your security profile and optional description.
4. In the Authentication Providers section, click **Add**.
5. Choose an authentication provider and click **Create**.
6. In the Management Cockpit Home page, click Applications.
7. Click **New**.
8. In the New Application window, enter the values for your application:

Field	Value
ID	Unique identifier for the application in reverse domain notation.
Name	Application name.
Vendor	(Optional) Vendor who developed the application.
Version	Application version. Currently, only version 1.0 is supported.
Type	Application type. • Native – native iOS and Android applications. • Hybrid – container-based applications, such as Kapsel. • Agentry – meta data-driven applications, such as Agentry. Application configuration options differ depending on your selection.
Description	(Optional) Short description of the application.

9. Click **Save**.

   Application-related tabs appear, and you are ready to configure the application, based on the application type.
10. In the Backend page, enter the end point for the back end system to which the application will connect. For example, `http://localhost:8090/odata`.

11. In the bottom right-hand corner, click **Save**.
12. Run the application on the device or emulator, and click **Register**.
13. In the registration page, enter the values, and click **Register**.

    The user name and password combination should have permission to access the OData backend.
14.

Creating an Application Connection

You must explicitly register the application connection using SAP Mobile Platform.

You can specify customized application properties for the client with the request. Provide the application connection ID, `X-SMP-APPCID`, using an explicit request header or a cookie. If the value is missing, SAP Mobile Platform generates a universally unique ID (UUID), which is communicated to the device through the response cookie `X-SMP-APPCID`.

Create an anonymous or authenticated application connection by issuing a POST request to this URL, including the application connection properties:

```
http[s]://<host:port>/[public/]/odata/applications/{latest|v1}/
{appid}/Connections
```

The URL contains these components:

- **host –** the host is defined by host name and should match with the domain registered with SAP Mobile Platform. If the requested domain name does not match, default domain is used..
- **port –** the port for listening to OData-based requests. By default the port number is 8080.
- **public –** if included, an anonymous connection is allowed.
- **odata/applications/ –** refers to the OData services associated with the application resources.
- **{latest|v1} –** version of the service document.
- **appid –** name of the application.
- **Connections –** name of the OData collection.

Application connection properties are optional. You can create an application connection without including any application properties.

`DeviceType` is an application connection property that you may set. Valid values for `DeviceType` are:

- **Android –** Android devices.
- **iPhone –** Apple iPhone.
- **iPad –** Apple iPad.
- **iPod –** Apple iPod.
- **iOS –** iOS devices.

- **Blackberry –** Blackberry devices.
- **Windows –** includes desktop or servers with Windows OS, such as Windows XP, Windows Vista, Windows 7, and Windows Server series OS.
- **WinPhone8 –** includes Windows mobile.
- **Windows8 –** includes Windows desktop version.

Specifying any other value for `DeviceType` returns a value "Unknown" in the `DeviceType` column.

Example of creating an application connection

Request:

```
<?xml version="1.0" encoding="UTF-8"?>
<entry xmlns="http://www.w3.org/2005/Atom" xmlns:m="http://
schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices">
    <category term="applications.Connection" scheme="http://
schemas.microsoft.com/ado/2007/08/dataservices/scheme"/>
    <content type="application/xml">
        <m:properties>
           <d:AndroidGcmRegistrationId>398123745023</d
AndroidGcmRegistrationId>
        </m:properties>
    </content>
</entry>
```

Response

```
<?xml version="1.0" encoding="utf-8"?>
<entry xmlns="http://www.w3.org/2005/Atom" xmlns:m="http://
schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xml:base="https://<smp base URL>/odata/applications/latest/
e2eTest/">

<id>https://<application URL>/odata/applications/latest/e2eTest/
Connections('4891dd0f-0735-47cc-a599-76bf8a16d457')</id>
<title type="text" />
<updated>2012-10-19T09:05:25Z</updated>
<author>
<name />
</author>
<link rel="edit" title="Connection"
href="Connections('4891dd0f-0735-47cc-a599-76bf8a16d457')" />
<category term="applications.Connection" scheme="http://
schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
<content type="application/xml">
 <m:properties>
    <d:ETag>2012-10-19 14:35:24.0</d:ETag>
    <d:ApplicationConnectionId>4891dd0f-0735-47cc-
a599-76bf8a16d457</d:ApplicationConnectionId>
    <d:AndroidGcmPushEnabled m:type="Edm.Boolean">false</
d:AndroidGcmPushEnabled>
    <d:AndroidGcmRegistrationId>398123745023</d
AndroidGcmRegistrationId>
```

```
    <d:AndroidGcmSenderId />
    <d:ApnsPushEnable m:type="Edm.Boolean">false</d:ApnsPushEnable>
    <d:ApnsDeviceToken />
    <d:ApplicationVersion>1.0</d:ApplicationVersion>
    <d:BlackberryPushEnabled m:type="Edm.Boolean">false</
d:BlackberryPushEnabled>
    <d:BlackberryDevicePin m:null="true" />
    <d:BlackberryBESListenerPort m:type="Edm.Int32">0</
d:BlackberryBESListenerPort>
    <d:BlackberryPushAppID m:null="true" />
    <d:BlackberryPushBaseURL m:null="true" />
    <d:BlackberryPushListenerPort m:type="Edm.Int32">0</
d:BlackberryPushListenerPort>
    <d:BlackberryListenerType m:type="Edm.Int32">0</
d:BlackberryListenerType>
    <d:CustomizationBundleId />
    <d:CustomCustom1 />
    <d:CustomCustom2 />
    <d:CustomCustom3 />
    <d:CustomCustom4 />
    <d:DeviceModel m:null="true" />
    <d:DeviceType>Unknown</d:DeviceType>
    <d:DeviceSubType m:null="true" />
    <d:DevicePhoneNumber m:null="true" />
    <d:DeviceIMSI m:null="true" />
    <d:PasswordPolicyEnabled m:type="Edm.Boolean">false</
d:PasswordPolicyEnabled>
    <d:PasswordPolicyDefaultPasswordAllowed
m:type="Edm.Boolean">false</d:PasswordPolicyDefaultPasswordAllowed>
    <d:PasswordPolicyMinLength m:type="Edm.Int32">0</
d:PasswordPolicyMinLength>
    <d:PasswordPolicyDigitRequired m:type="Edm.Boolean">false</
d:PasswordPolicyDigitRequired>
    <d:PasswordPolicyUpperRequired m:type="Edm.Boolean">false</
d:PasswordPolicyUpperRequired>
    <d:PasswordPolicyLowerRequired m:type="Edm.Boolean">false</
d:PasswordPolicyLowerRequired>
    <d:PasswordPolicySpecialRequired m:type="Edm.Boolean">false</
d:PasswordPolicySpecialRequired>
    <d:PasswordPolicyExpiresInNDays m:type="Edm.Int32">0</
d:PasswordPolicyExpiresInNDays>
    <d:PasswordPolicyMinUniqueChars m:type="Edm.Int32">0</
d:PasswordPolicyMinUniqueChars>
    <d:PasswordPolicyLockTimeout m:type="Edm.Int32">0</
d:PasswordPolicyLockTimeout>
    <d:PasswordPolicyRetryLimit m:type="Edm.Int32">0</
d:PasswordPolicyRetryLimit>
    <d:ProxyApplicationEndpoint>http://<backend URL></
d:ProxyApplicationEndpoint>
    <d:ProxyPushEndpoint>http[s]://<host:port>/Push</
d:ProxyPushEndpoint>
    <d:MpnsChannelURI m:null="true" />
    <d:WnsChannelURI m:null="true" />
</m:properties>
</content>
</entry>
```

CORS Support
Cross-domain HTTP requests are requests for resources from a different domain than the domain of the resource making the request. Cross-Origin Resource Sharing (CORS) mechanism provides a way for web servers to support cross-site access controls, which enable secure cross-site data transfers.

Getting Application Settings

You can retrieve application connection settings for the device application instance by issuing the GET method.

You can retrieve application settings by either explicitly specifying the application connection ID, or by having the application connection ID determined from the call context (that is, from either the X-SMP-APPCID cookie or X-SMP-APPCID HTTP header, if specified). On the first call, you can simplify your client application code by having the application connection ID determined from the call context, since you have not yet received an application connection ID.

If you supply an application connection ID, perform an HTTP GET request at:

```
http[s]://<host:port>/[public/]odata/applications/{latest|v1}/
{appid}/Connections(‘{appcid}’)
```

Response

```
<?xml version='1.0' encoding='utf-8'?>
<entry xmlns="http://www.w3.org/2005/Atom"
       xmlns:m="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata"
       xmlns:d="http://schemas.microsoft.com/ado/2007/08/
dataservices"
       xml:base="https://<smp base URL>/odata/applications/v1/
e2eTest/">
  <id>http://https://mobilesmpdev.netweaver.ondemand.com/smp/odata/
applications/v1/e2eTest/
Connections('c9d8a9da-9f36-4ae5-9da5-37d6d90483b5')</id>
  <title type="text" />
  <updated>2012-06-28T09:55:48Z</updated>
  <author><name /></author>
  <link rel="edit" title="Connections"
href="Connections('c9d8a9da-9f36-4ae5-9da5-37d6d90483b5')" />
  <category term="applications.Connection" scheme="http://
schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:ETag m:type="Edm.DateTime">2012-06-28T17:55:47.685</d:ETag>

<d:ApplicationConnectionId>c9d8a9da-9f36-4ae5-9da5-37d6d90483b5</
d:ApplicationConnectionId>
      <d:AndroidGcmPushEnabled m:type="Edm.Boolean">false</
d:AndroidGcmPushEnabled>
      <d:AndroidGcmRegistrationId m:null="true" />
      <d:AndroidGcmSenderId m:null="true" />
      <d:ApnsPushEnable m:type="Edm.Boolean">true</d:ApnsPushEnable>
      <d:ApnsDeviceToken m:null="true" />
      <d:ApplicationVersion m:null="true" />
      <d:BlackberryPushEnabled m:type="Edm.Boolean">true</
d:BlackberryPushEnabled>
      <d:BlackberryDevicePin>00000000</d:BlackberryDevicePin>
      <d:BlackberryBESListenerPort m:type="Edm.Int32">5011</
d:BlackberryBESListenerPort>
      <d:BlackberryPushAppID m:null="true" />
      <d:BlackberryPushBaseURL m:null="true" />
      <d:BlackberryPushListenerPort m:type="Edm.Int32">0</
d:BlackberryPushListenerPort>
      <d:BlackberryListenerType m:type="Edm.Int32">0</
d:BlackberryListenerType>
      <d:CustomCustom1>custom1</d:CustomCustom1>
      <d:CustomCustom2 m:null="true" />
      <d:CustomCustom3 m:null="true" />
      <d:CustomCustom4 m:null="true" />
      <d:DeviceModel m:null="true" />
      <d:DeviceType>Unknown</d:DeviceType>
      <d:DeviceSubType m:null="true" />
      <d:DevicePhoneNumber>12345678901</d:DevicePhoneNumber>
      <d:DeviceImsi m:null="true" />
      <d:PasswordPolicyEnabled m:type="Edm.Boolean">true</
d:PasswordPolicyEnabled>
      <d:PasswordPolicyDefaultPasswordAllowed
m:type="Edm.Boolean">false</d:PasswordPolicyDefaultPasswordAllowed>
      <d:PasswordPolicyMinLength m:type="Edm.Int32">8</
d:PasswordPolicyMinLength>
      <d:PasswordPolicyDigitRequired m:type="Edm.Boolean">false</
d:PasswordPolicyDigitRequired>
      <d:PasswordPolicyUpperRequired m:type="Edm.Boolean">false</
d:PasswordPolicyUpperRequired>
      <d:PasswordPolicyLowerRequired m:type="Edm.Boolean">false</
d:PasswordPolicyLowerRequired>
      <d:PasswordPolicySpecialRequired m:type="Edm.Boolean">false</
d:PasswordPolicySpecialRequired>
      <d:PasswordPolicyExpiresInNDays m:type="Edm.Int32">0</
d:PasswordPolicyExpiresInNDays>
      <d:PasswordPolicyMinUniqueChars m:type="Edm.Int32">0</
d:PasswordPolicyMinUniqueChars>
      <d:PasswordPolicyLockTimeout m:type="Edm.Int32">0</
d:PasswordPolicyLockTimeout>
      <d:PasswordPolicyRetryLimit m:type="Edm.Int32">20</
d:PasswordPolicyRetryLimit>
      <d:ProxyApplicationEndpointm:null="true" />
      <d:ProxyPushEndpoint>http://xxue-desktop:8080/GWC/
SMPNotification</d:ProxyPushEndpoint>
      <d:WnsChannelURI m:null="true" />
      <d:MpnsChannelURI m:null="true" />
      <d:WnsPushEnable m:type="Edm.Boolean">false</d:WnsPushEnable>
      <d:MpnsPushEnable m:type="Edm.Boolean">true</d:MpnsPushEnable>
      </m:properties>
  </content>
</entry>
```

You can also retrieve a property value by appending the property name in the URL. For example, to retrieve the `ClientLogLevel` property value, enter:

```
http[s]://<host:port>/[public/]odata/applications/{v1|latest}/
{appid}/Connections('{appcid}')/ClientLogLevel
```

Running and Testing Kapsel Applications

Test your Cordova project by opening it in its respective development environment (Eclipse with Android plugins or Xcode), then run it in the corresponding emulator (Android) or simulator (iOS),

You can launch the emulator or simulator from the Cordova command line interface, or from the development environment.

Client-side Debugging

Debug the Kapsel application on the device or by using a desktop browser.

Debugging in a Desktop Browser

Debug the JavaScript code running in a deskstop browser.

This procedure shows how to debug using Chrome. See *https://developers.google.com/chrome-developer-tools/*. In some cases, debugging on the device is necessary, for example, when you debug touch, or code that includes JavaScript files from Apache Cordova or Kapsel, since these expect to run on a mobile device or simulator.

1. In the Chrome menu, choose **Tools** > **Developer Tools**.
2. Click **Sources** to open a source file.
3. Set break points to step through the code.
4. Use the **Network** tab to examine the OData URL sent and the values received.

Debugging on iOS

This procedure demonstrates how to debug an app that includes Apache Cordova and Kapsel plugins.

This procedure requires a device or simulator running iOS 6 and a Mac that has Safari 6.

1. Connect the device to the Mac with a USB cable, or start the simulator.
2. On the device or simulator, go to **Settings** > **Safari** > **Advanced** > **Web Inspector**, and turn it to **On**.
3. On the device or simulator, open your Kapsel app or a Web page in the Safari browser.
4. On the Mac, in Safari, choose **Develop** > **iPhone Simulator** > **index.html**.

Running the Kapsel Application on Android

Open your Cordova based Kapsel project in Eclipse and run it on the emulator.

1. In a Command Prompt window, make sure you are in the project folder and execute the command:
   ```
   cordova prepare android
   ```
2. Start Eclipse.
3. From the menu, choose **File > Import**.
4. In the Import window, select **Android > Existing Android Code Into Workspace**.
5. Browse to your project, `<ProjectName>/platforms/android`, select the android folder, and click **Open**.
6. Click **Finish**.

   The project is imported into Eclipse.
7. Right-click the project node and select **Run As > Android Application**.

Running the Kapsel Application on iOS

Open your project in Xcode and run the application on the simulator.

1. In a terminal window, make sure you are in the project folder and execute the command:
   ```
   cordova prepare ios
   ```
2. Open Xcode.
3. In a Finder window, browse to your Cordova project folder, `<Project Name>/platforms/ios`.
4. Double-click the `<ProjectName>.xcodeproj` file to open the project in Xcode.
5. Select your Simulator type and click the **Run** button.

Package and Deploy Kapsel Applications

Use the Android IDE or Xcode to package the Kapsel app, then use the Management Cockpit to upload the app to the server.

Generating and Uploading Kapsel App Files Using the Command Line Interface

The Kapsel command line interface provides a way to generate a ZIP file that contains the HTML files that make up the app.

1. Open a command prompt window, or terminal, and navigate to the folder that contains the Kapsel command line interface, for example:

   On Windows:

   `SDK_HOME\MobileSDK3\KapselSDK\cli`

   On Mac:

   `~SDK_HOME/MobileSDK3/KapselSDK/cli`

2. Run the command:

   `npm -g install`

   On Mac, you may need to run the command as sudo:

   `sudo npm -g install`

3. Change directories to the directory containing the project and run the command:

   ```
   kapsel package
   kapsel deploy <com.mycompany.app_ID> localhost:<port>
   <Admin_user_name> <Admin_password>
   ```

   You can, optionally, enter a platform in the package command, such as android or ios. The parameters to the deploy command are the app ID, the SAP Mobile Platform Server host name, and the user ID and password for Management Cockpit.

   The ZIP file containing the HTML files that make up the app is generated and then uploaded to SAP Mobile Platform Server, and the Management Cockpit shows that revision *x* was uploaded.

Changing the Default Port

By default, the Kapsel command line interface is configured to use port 8083, as this is the default used for Management Cockpit when installing SAP Mobile Platform Server.

If port 8083 is in use during installation, the installer automatically assigns a different port and notifies you. If the port number changes from 8083, you can change it using the command line.

1. Open a command prompt window, or terminal.
2. Specify the server paramater in the format `server:port`, for example:

   ```
   kapsel deploy <com.mycompany.app_ID> localhost:<port>
   <Admin_user_name> <Admin_password>
   ```

   This example shows how to deploy the Kapsel application called "com.sample.app" using port 8084.

   ```
   kapsel deploy com.sample.app localhost:8084 smpAdmin
   s3pAdmin
   ```

Preparing the Application for Upload to the Server

Upload the Kapsel app to SAP Mobile Platform Server.

1. In the Android IDE or Xcode, right-click the project's **www** folder, and compress the items to package the files in a ZIP file.
2. Log in to the Management Cockpit to upload the app.

Uploading and Deploying Hybrid Apps

Upload a new or updated version of a hybrid app package, and deploy it to make the current version available to users. The older version is retained until you delete it.

Prerequisites

The application developer creates the hybrid app package that:

- Contains the contents of the application's `www` folder and `config.xml` of the project, with a separate folder in the archive for each mobile platform (`android/www` and/or `ios/www` in all lower case). Format structure for hybrid apps:

```
|- android
|  |- config.xml
|  |- www
|- iOS
...
```

- Is compressed into a standard `.zip` file for upload.

Task

1. From Management Cockpit, select **Applications** > **App Specific Settings**.
2. Under Upload, click **Browse**.
   a) In the dialog, navigate to the directory.
   b) Select the hybrid app package, and confirm.

   New version information appears for the uploaded Kapsel app for each mobile platform. You cannot change this information.

Property	Description
Required Kapsel Version	Identifies the Kapsel SDK version used to develop the Kapsel app, for example 3.0.0. **Note:** This version attribute is informational only, and is not used by SAP Mobile Platform Server to determine whether device clients should receive the Web application update.
Development Version	Identifies the internal development version used to develop the Kapsel app.
Description	Describes the Kapsel app.

Property	Description
Revision	Identifies the production version revision. For a newly uploaded Kapsel app, this is blank. **Note:** When the Kapsel app is deployed, the revision number is incremented.

3. When ready, deploy the application:

a. Click **Deploy** and confirm to deploy an application to one mobile platform.

b. Click **Deploy All** and confirm to deploy the application to all available mobile platforms.

Deployed Kapsel app information appears as the current version and the revision number is incremented.

For device application users:

- When a device user with a default version (revision = 0) of the Kapsel app connects to the server, the server downloads the full Kapsel app.
- When a device user with a version (revision = 1 or higher) of the Kapsel app connects to the server, the server calculates the difference between the user's version and the new version, and downloads a patch containing only the required changes.
- If the application implements the AppUpdate plugin, the server checks for updates when the application starts-up or is resumed.
- If the developer enables a feature in the application, an application or a user can check for updates manually.

4. When ready, delete the older version of an application:

a. Click **Remove** and confirm to remove an older version of an application from one mobile platform.

b. Click **Remove All** and confirm to remove all older versions of an application from all available mobile platforms.

Deploying Hybrid Apps Using the REST API

Deploy a new or updated hybrid app to SAP Mobile Platform Server using the deploy application REST API.

Once the application is deployed, it is considered to be a new version. You can make it the current version using the promotion REST API. After the application is promoted, users can download a patch to upgrade the application on their devices.

Note: It is not possible to deploy a hybrid app for a specific platform: everything in the file is deployed. Once the application is deployed, you can promote or delete hybrid apps for specific platforms as needed.

Syntax

Perform a POST request to the following URI:

```
https://<host>:<admin_port>/Admin/kapsel/jaxrs/KapselApp/{APP_ID}
```

Parameters

- **file –** The file that contains the application archive, sent as multipart/form-data.

Returns

A response providing information about the new and current version of the application. For example:

```
{"newVersion":
  {"requiredKapselVersion":"1.5",
   "developmentVersion":"1.2.5",
   "description":"An update for the sample app.",
   "revision":-1},
 "currentVersion":
  {"requiredKapselVersion":"1.5",
   "developmentVersion":"1.2.4",
   "description":"A sample app.",
   "revision":2}
}
```

On successful deployment, the client receives a 201 status code; otherwise, an HTTP failure code and message.

Examples

Note: This example users the `curl` command line client and the `--cacert` flag. Your client may require you to pass other arguments or set specific configuration options.

- **Deploy application to all platforms**

  ```
  curl --user <user>:<password> --cacert <your-server.pem> --form
  "file=@C:\work\app1.zip" https://localhost:8083/Admin/kapsel/
  jaxrs/KapselApp/MyTestAppId
  ```

Removing Kapsel Plugins

Remove Kapsel plugins from the application.

To remove a plugin, refer to it by the same identifier that appears in the Cordova plugin listing. These steps show an example of how you would remove the logger plugin.

Note: Due to a known Apache issue, **plugin remove** does not currently work properly with Kapsel plugins. See *https://issues.apache.org/jira/browse/CB-41.* Shared dependencies with other plugins may also be removed, leaving the application in a bad state. Instead of removing plugins, SAP recommends that you start from a clean state and add only the plugins you require.

1. Open a command prompt window and navigate to the Cordova project's directory.
2. (Optional) To see a list of installed plugins, enter:

   ```
   cordova plugins
   ```
3. Enter:

   ```
   cordova plugin remove <plugin_name>
   ```

   For example, to remove the Logger plugin, enter:

   ```
   cordova plugin remove com.sap.mp.cordova.plugins.logger
   ```

Index