Kapsel Development

SAP Mobile Platform 3.0 SP02

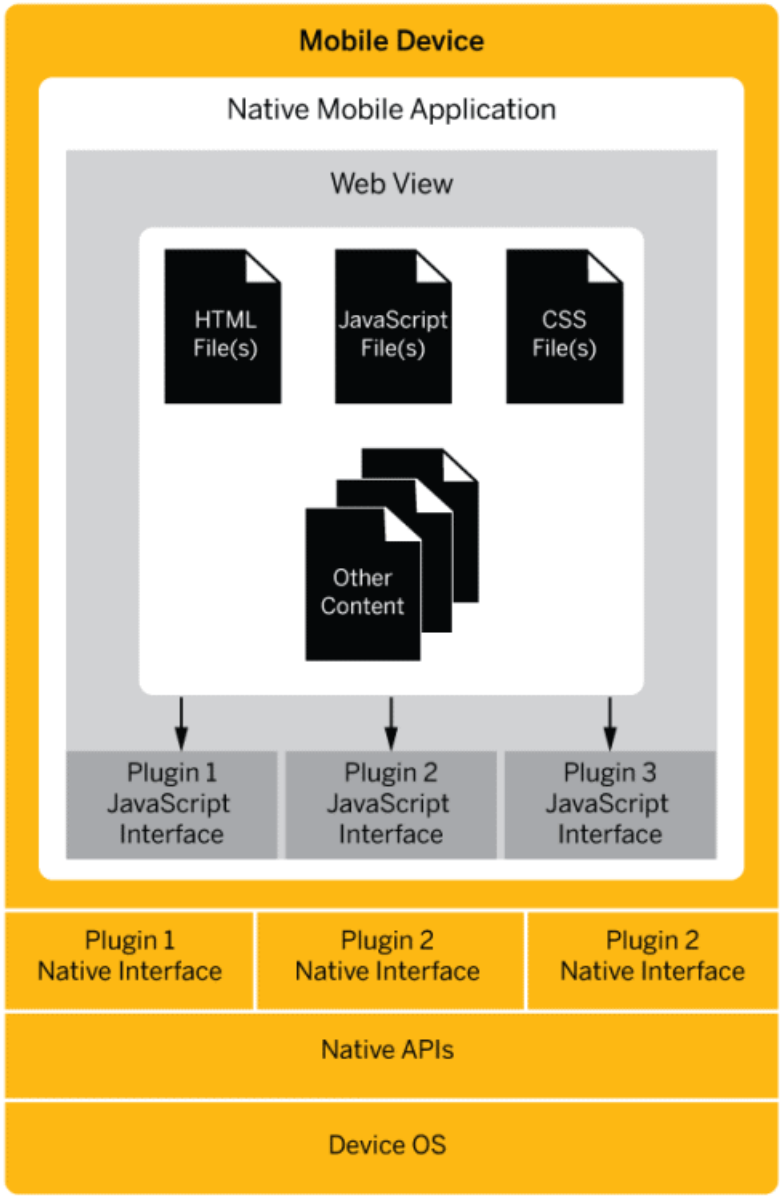
DOCUMENT ID: DC-01-0302-01
LAST REVISED: February 2014

Contents

Kapsel Development

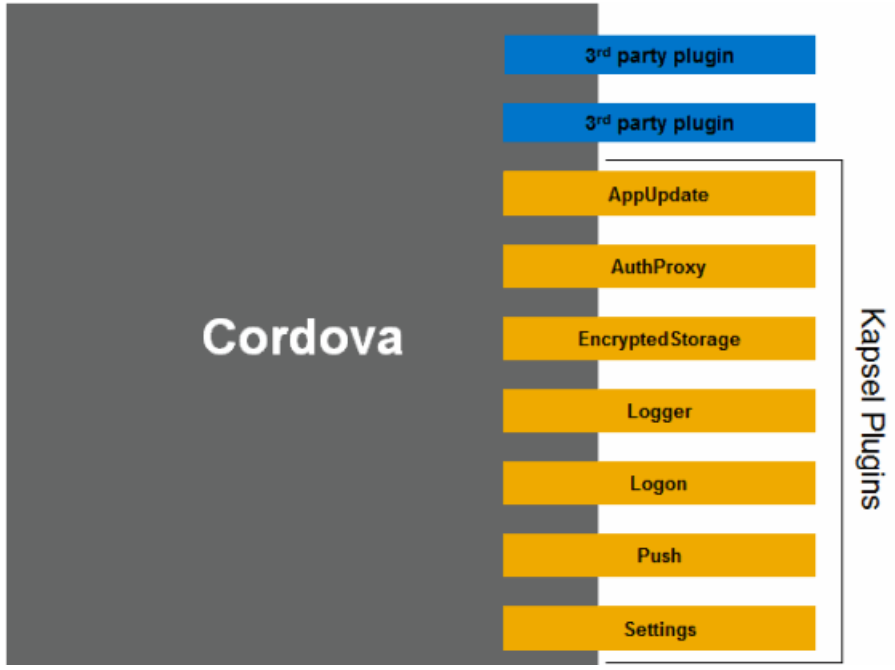
Kapsel is a set of SAP® plugins for Apache Cordova.

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.



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.

On Windows:

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

On Mac:

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

On Windows:

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

On Mac:

```
sudo npm config set proxy  http://proxy_host:port
sudo 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, which enables you to use Management Cockpit to manage the application.

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 bundled identifier that the application developer assigns or generates during application development. The administrator uses the Application ID to register the application to SAP Mobile Platform Server, and the client application code uses the Application ID while sending requests to the server. Reverse domain notation means reversing a registered domain name; for example, reverse domain notation for the object `MyApp.sap.com` is `com.sap.MyApp`. The application identifier: • Must be unique. • Must start with an alphabetic character. • Can contain only alphanumeric characters, underscores (_), and periods (.). • Cannot include spaces. • Can be up to 64 characters long. **Note:** The keywords that are not allowed to be entered as application identifiers include: Admin, AdminData, Push, smp_cloud, resource, test-resources, resources, Scheduler, odata, applications, Connections, public. These keywords are case sensitive. Formatting guidelines: • SAP recommends that application IDs contain a minimum of two periods ("."). For example, this ID is valid: `com.sap.mobile.app1`. • Application IDs cannot start with a period ("."). For example, this ID is invalid: `.com.sap.mobile.app1`. • Application IDs cannot include two consecutive periods ("."). For example, this ID is invalid: `com..sap.mobile.app1`.

Field	Value
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 applications, including Android, BlackBerry, iOS, Windows Mobile 8, and Windows 8. • Hybrid – Kapsel container-based applications. • Agentry – metadata-driven application. You can configure only one Agentry application per SAP Mobile Platform Server; after that, Agentry no longer appears as an option.
Description	(Optional) Short description of the application. The description: • Can contain alphanumeric characters. • Can contain most special characters, except percent signs (%) and ampersands (&). • Can be up to 255 characters long.

3. Click **Save**. You see application-related tabs, such as Back End, Authentication, Push, and so forth. The tabs you see differ by application type. 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 steps that follow assume you have selected the application, and are working through each of the relevant tabs for your selected application. The steps use a "navigational shorthand"— such as "select **Applications > Back End**—to indicate the tab where tasks are performed, relative to that selected application, rather than repeating the entire navigation instruction.

Defining Back-end Connections for Native and Hybrid Apps

Define back-end connections for the selected native or hybrid application. SAP Mobile Platform supports one primary endpoint per application ID. However, the administrator can create multiple secondary endpoints for other services used by the application; SAP Mobile Platform treats these additional endpoints as proxy connections.

1. From Management Cockpit, select the **Home** tab, and then **Configure Application**. Alternatively, select the **Applications** tab.
2. On the Applications tab, select one of the applications.

You see application-related tabs, such as Back End, Authentication, Push, and so forth. The tabs you see differ by application type. 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 steps that follow assume you have selected the application, and are working through each of the relevant tabs for your selected application. The steps use a navigational shorthand— such as "select **Applications > Back End**"—to indicate the tab where tasks are performed, relative to that selected application, rather than repeating the entire navigation instruction.

3. Enter values for the selected application:

Field	Value
Connection Name	(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 (.). • Cannot include spaces.

Endpoint	The URL (back-end connection, or service document) 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. Include a trailing slash to avoid triggering a redirection of the URL, and losing important HTTP header details. This is especially important when configuring the application with security, such as SSOToken and Certificates, and when Rewrite URL is enabled. 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/`

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. Enable this setting only 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.

Field	Value
Rewrite URL	(Optional) Whether to mask the back-end URL with the equivalent SAP Mobile Platform Server URL. Enable this setting 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, this property is enabled.

Allow Anonymous Access	(Optional) Whether to enable anonymous access, which means the user can access the application without entering a user name and password. However, the back-end system still requires login credentials for data access, whether it is a read-only user, or a back-end user with specific roles.
	• If enabled and the back end requires it, enter the login credential values used to access the back-end system:
	  • **User name** – supply the user name for the back-end system.
	  • **Password** – (required if you set a user name) supply the password for the back-end system.
	• If disabled (the default value) or the back end does not require it, you need not provide these credentials.
	Note: If you use Allow Anonymous Access for a native OData application, do not also assign the No Authentication Challenge security profile to the application; anonymous OData requests are not sent, and 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 and minimum is 500 connections. Factors to consider when resetting this property: • The expected number of concurrent users of the application. • The load that is acceptable to the back-end system. • The load that the underlying hardware and network can handle. Increase the maximum number of connections only if SAP Mobile Platform Server hardware can support the additional parallel connections, and if the underlying hardware and network infrastructure can handle it.
Certificate Alias	If the back-end system has a mutual SSL authentication requirement, supply the certificate alias name given to the private key and technical user certificate that is used to access the back-end system. The alias is located in `smp_keystore`. Otherwise, leave the entry blank.

4. (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 back-end connection is added to the list.

You can maintain the list of server-level back-end connections (including all the connections in SAP Mobile Platform Server), and of application-specific back-end connections. Application-specific back-end connections are the secondary connections that are enabled for an application; by default, no secondary connections are enabled. You must explicitly enable additional back-end connections for an application. Users who are registered to an application can access only these back-end connections. If a user attempts to access a back-end connection (request-response) that is not enabled for an application, it is not allowed and a 403, `Forbidden` error is thrown.

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

   Select **Server-level Connections** from the drop-down to show all available connections for the server. Use the checkbox to enable additional connections for the application.

Note:

- You can authenticate multiple back ends using various authentication provider options in the back-end security profile.
- If the back-end system issues a "302 Redirect" response, which means it is redirecting the request to a different URL, then you must also add the target URL to the list of application-specific connections.

Defining Application Authentication

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

Prerequisites

Configure security profiles for application authentication.

Task

Security profiles are made up of one or more authentication providers. These authentication providers can be shared across multiple security profiles, and can be modified in Management Cockpit. For more information on authentication providers, see *Authentication in SAP Mobile Platform*.

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. Select a security profile name from the Name list.

The name appears under Security Profile Properties, and the providers that are associated with the security profile 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. By default the property is enabled, which prevents the user authentication from succeeding in this scenario.

5. Under Authentication Providers, you can select a security profile URL to view its settings. To change its settings, you must modify it using **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
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)

Security Configuration	Implemented Using	Security Provider
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*.

Using UI Development Frameworks

Kapsel is UI5 framework agnostic. You can use any third party framework with Kapsel that is compatible with Cordova. This section discusses framework integration with Kapsel, and provides an overview of common UI libraries for standards-based Web development.

SAPUI5

If you have not already selected a framework for your UI development, selecting SAPUI5 allows you to efficiently leverage Kapsel's integration with SAPUI5. Kapsel's Login plugin uses the UI5 framework. Another strength of UI5 is that you can write an application, for desktop and mobile, using a single code base.

SAPUI5, also known as SAP UI Development Toolkit for HTML5, is the SAP client-side HTML5 rendering library with a large set of RIA-like standard and extension controls based on JavaScript (JS), and a lightweight programming model. The rendering control library is Open AJAX-compliant and based on open source jQuery and can be used together with other client-side libraries.

For more information:

- *SAPUI5*
- *SAPUI5 for Mobile*
- *SAPUI5 Mobile Demo Apps*

jQuery Mobile

Some of the other popular open-source frameworks include:

jQuery Mobile is an HTML5 based user interface system for mobile devices.

For more information:

- *jQuery Mobile*

Sencha Touch

Sencha Touch is an HTML5 mobile application framework that encourages a model, view, and controller pattern.

For more information:

- *Building SAP mobile apps with Sencha Touch*
- *SAP and Sencha Touch*

Kendo UI

Kendo UI is an HTML5/JavaScript framework for modern Web and mobile application development.

For more information:

- *Kendo UI*

Configuring the Client for Authentication

You can configure a client for basic authentication or mutual certificate authentication with Afaria Server.

Configuring the iOS Client for Basic Authentication over HTTP(S)

Configure the iOS client for basic authentication over HTTP or HTTPS.

1. For HTTPS, import the root certificate to the iOS device.
2. Set the selected application ID in the `sap.Logon.init` method when initializing the application.
3. Set the project settings' keychain group to "clienthubEntitlements" and $(CFBundleIdentifier).
4. Start the application on device and register with the following settings:
   - **Username –** Username as assigned by your administrator.
   - **Password –** Password for your assigned username.
   - **Host –** Enter the hostname for your smp server. This value must match the server name used for generating the server certificate.
   - **Port –** For HTTPS, enter 8081.

     For HTTP, enter 8080.
   - **Secure –** For HTTPS, enter `ON`.

     For HTTP, enter `OFF`.
   - **Security configure –** Enter the selected security configuration name for sharing the credentials with other applications.
5. After registering, a new registration appears in Management Cockpit for the application.

Configuring the iOS Client for Mutual Authentication over HTTP(S)

Configure the iOS client for mutual authentication using a certificate provisioned through Afaria Server.

1. Verify that the application ID uploaded in Afaria Server's application deployment package matches the Application ID set on the device application.
2. If the SAP Mobile Platform Server uses a self-signed certificate for the HTTPS connection, then the device needs to import the server certificate to trust the server certificate.

3. Open the Xcode project and in `MAFLogonMangerNG.bundle` set `keyMAFUseAfaria to YES` (launches Afaria if installed on the device).
4. Verify that the `handleOpenURL()` method is defined in `index.js`. If it is not defined, then define an empty method for it, so that the client application functions properly when launched by the Afaria client.

   ```
   function handleOpenURL(url){}
   ```
5. Create the `Url schemes` and `Url identifier` items in the .plist file for Logon.
6. Register a custom URL scheme by creating the `Url schemes` and `Url identifier` items in the application's info .plist file, to allow communications with the Afaria client.
7. The client connection settings can come from either the Afaria package configuration or the `clienthub.plist` file. The settings in the `clienthub.plist` file are applied only if Client Hub is enabled for sharing credentials. If both settings are available, the settings in the `clienthub.plist` file take priority over the Afaria package configuration.

Configuring the Android Client for Basic Authentication over HTTP(S)

Configure the iOS client for basic authentication over HTTP or HTTPS.

1. For HTTPS, import the root certificate to the device.
2. Start the application on device and register with the following settings:
   - **Username –** Username as assigned by your administrator.
   - **Password –** Password for your assigned username.
   - **Host –** Enter the hostname for your smp server. This value must match the server name used for generating the server certificate.
   - **Port –** For HTTPS, enter 8081.

     For HTTP, enter 8080.
   - **Secure –** For HTTPS, enter `ON`.

     For HTTP, enter `OFF`.
   - **Security configure –** Enter `BASIC`.
3. After registering, a new registration appears in Management Cockpit for the application.

Configuring the Android Client for Mutual Authentication over HTTP(S)

Configure the Android client for mutual authentication using a certificate provisioned through Afaria Server.

1. Verify that the application ID uploaded in Afaria Server's application deployment package matches the Application ID set on the device application.
2. Import the root certificate to the device. Note that the server's DNS name, not an IP address, must be described in the certificate's common name (CN) field.
3. Verify that the `handleOpenURL()` method is defined in `index.js`. If it is not defined, then define an empty method for it, so that the client application functions properly when launched by the Afaria client.

   ```
   function handleOpenURL(url){}
   ```
4. When you use Client Hub, you can use an optional `clienthub.property` file to specify the registration settings. Update the file server connection information based on your own server, and add `clienthub.property` file into the resource folder of the application project.

Managing Application Registration Using Client Hub

Kapsel application can use the Client Hub, integrated with Logon Manager, to simplify user onboarding and configuration to enable easier and faster enterprise-wide deployments.

The Client Hub saves the end user from managing multiple passwords for mobile applications, thereby improving the user experience. The Client Hub provides these functions:

- Manages single sign-on (SSO) on the device.
- Enables cosigned business applications with the same security configuration to securely share credentials on the device.
- Supports multiple security configurations per device.

For more information on Client Hub, see these topics in *SAP Mobile Platform SDK > Developer 3.0 SP02*:

- *Client Hub*
- *Managing iOS Application Registration Using Client Hub*
- *Managing Android Application Registration Using Client Hub*

Provisioning Applications Using Afaria

SAP Mobile Platform supports Afaria device management and security functionality. You can use Afaria to provision native and hybrid applications that use the MAF Logon UI component. You can generate certificate requests which in turn are passed through Afaria to the corporate PKI system for CA signature.

When provisioning an application, you must use the provisioning file method to configure the application configuration data, rather than entering application data in the provisioning text box in Afaria. In this method, you create a provisioning file containing a set of parameters.

For information on setting up the Afaria environment, and the format for creating the provisioning file, see:

- *SAP Mobile Platform Server > Administrator 3.0 SP02 > Application Administration > Provisioning Applications > Provisioning with Afaria*.

Preparing the Kapsel Application for Afaria Provisioning

To prepare the Kapsel application to be provisioned through Afaria, provide an initial context to the Login plugin, and specify an application ID configured on SAP Mobile Platform Server with an X.509 security profile.

Specify information needed for provisioning in these files:

- The `clienthub.properties` file (for Android) or `clienthub.plist file` (for iOS) provide an initial context to the Login plugin and instruct the Login plugin that it is to use a certificate from Afaria for authentication.
- The `index.html` file specifies an application ID that must match an application ID on the SAP Mobile Platform Server that is configured with a security profile for X.509 authentication.

Create an index.html which specifies an application ID that is configured on SAP Mobile Platform Server with a security profile for X.509 authentication. The following is a sample index.html:

```
<html>
    <head>
        <script src="datajs-1.1.1beta2.js"></script>
        <script type="text/javascript" charset="utf-8"
src="cordova.js"></script>
        <script>
            applicationContext = null;
           var appId = "certAuth"; // Change this to app id on server

            function init() {

                // Optional initial connection context
                var context = {
                  "serverHost": "example.corp", //Place your SMP 3.0
server name here
                    "https": "false",
                    "serverPort": "8080",
                 "user": "username", //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(logonSuccessCallback, errorCallback,
appId, context, sap.logon.IabUi);
                console.log("init completed");
            }
```

```
            function read() {
                var url = applicationContext.applicationEndpointURL;
                var logonCert = new
sap.AuthProxy.CertificateFromLogonManager(appId);
                var headers = {};
                headers["X-SMP-
APPCID"]=applicationContext.applicationConnectionId;
                var errorCB = function(errorInfo){
                    alert("error: " + JSON.stringify(errorInfo));
                }
                var successCB = function(result){
                    alert("success: " + JSON.stringify(result));
                }


sap.AuthProxy.get(url,headers,successCB,errorCB,null,null,null,logo
nCert);
            }

            function readSuccessCallback(data, response) {
                alert("success: " + JSON.stringify(data));
                /*var carrierTable =
document.getElementById("carrierTable");

                for (var i = data.results.length -1; i >= 0; i--) {
                    var row = carrierTable.insertRow(1);
                    var cell1 = row.insertCell(0);
                    var cell2 = row.insertCell(1);
                    cell1.innerHTML = data.results[i].carrid;
                    cell2.innerHTML = data.results[i].CARRNAME;
                }*/
            }

            function clearTable() {
                var carrierTable =
document.getElementById("carrierTable");
                while(carrierTable.rows.length > 1) {
                    carrierTable.deleteRow(1);
                }
            }

            function logonSuccessCallback(result) {
                console.log("logonSuccessCallback " +
JSON.stringify(result));
              if (result) {  //calling registerOrUnlock returns null
the second time it is called.
                    applicationContext = result;
                }
            }

            function logonLockSuccessCallback(result) {
                console.log("logonLockSuccessCallback " +
JSON.stringify(result));
                applicationContext = null;
            }
```

```
            function logonUnregisterSuccessCallback(result) {
                console.log("logonUnregisterSuccessCallback " +
JSON.stringify(result));
                applicationContext = null;
            }

            function register() {
                sap.Logon.registerOrUnlock(logonSuccessCallback,
errorCallback);
            }

            function unRegister() {

sap.Logon.core.deleteRegistration(logonUnregisterSuccessCallback,
errorCallback);
                clearTable();
            }

            function lock() {

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

sap.Logon.lock(logonLockSuccessCallback,errorCallback);
                clearTable();
            }

            function unlock() {
               sap.Logon.unlock(logonSuccessCallback,errorCallback);
            }

            document.addEventListener("deviceready", init, false);

            </script>

    </head>
    <body>
        <h1>Logon Sample</h1>
        <button id="register" onclick="register()">Register</button>
        <button id="read" onclick="read()">Read</button>
        <button id="unregister" onclick="unRegister()">Unregister</
button>
        <button id="lock" onclick="lock()">Lock</button>
        <button id="unlock" onclick="unlock()">Unlock</button>
        <table id="carrierTable"><tr><th>Carrier ID</th><th>Carrier
Name</th></tr></table>
    </body>
</html>
```

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 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 either through SAP Mobile Platform Server, or through SAP Gateway 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. To keep your keys safe from unauthorized use, you should store all keys in the data vault.

The data vault is deleted if the user forgets their password while unlocking the application, violates a password policy set on the server, or explictly deletes the registration. Data stored by the EncryptedStorage plugin is also deleted, because once the data vault is deleted this data would no longer be accessible. For security reasons, when the data vault is deleted, the Login plugin sends a notification to the other Kapsel plugins so they can clean up their data if required.

The Logon plugin also lets the user lock and unlock the application, to protect sensitive data.

Security Configurations
Kapsel supports the following security configurations:

- Basic authentication (HTTP)
- Basic authentication (HTTPS)
- SSO over HTTP
- SSO over HTTPS
- Mutual certificate authentication between the client and SAP Mobile Platform Server
- SSO with certificate (X509) MCIM
- SSO with token - MCIM
- SSO with username password - MCIM

From the client perspective, the client authenticates either through basic authentication, or through mutual certificate authentication. In the basic authentication scenario, the client must provide credentials (username and password), and in mutual certificate authentication the client must provide a root server certificate.

Domain Whitelisting
Kapsel plugins support Apache Cordova's domain whitelisting model. Whitelisting allows you to control access to external network resources. Apache Cordova whitelisting allows you to whitelist individual network resources (URLs), for example, http://www.google.com.

For information about the whitelist rules, see *http://docs.phonegap.com/en/3.3.0/guide_appdev_whitelist_index.md.html*.

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.

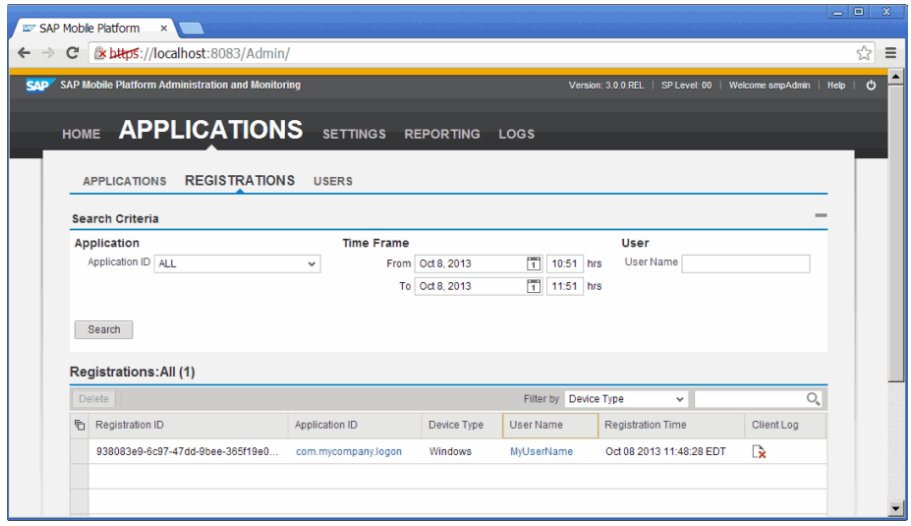
6. Make a call to the Logon plugin's `init` method as shown:

   ```
   //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

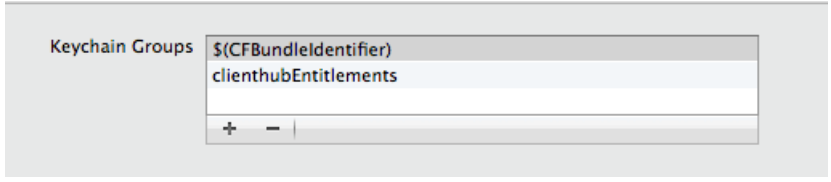
the user. As part of this process, the appropriate screens are shown to gather user input and manage the entire process.

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.



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

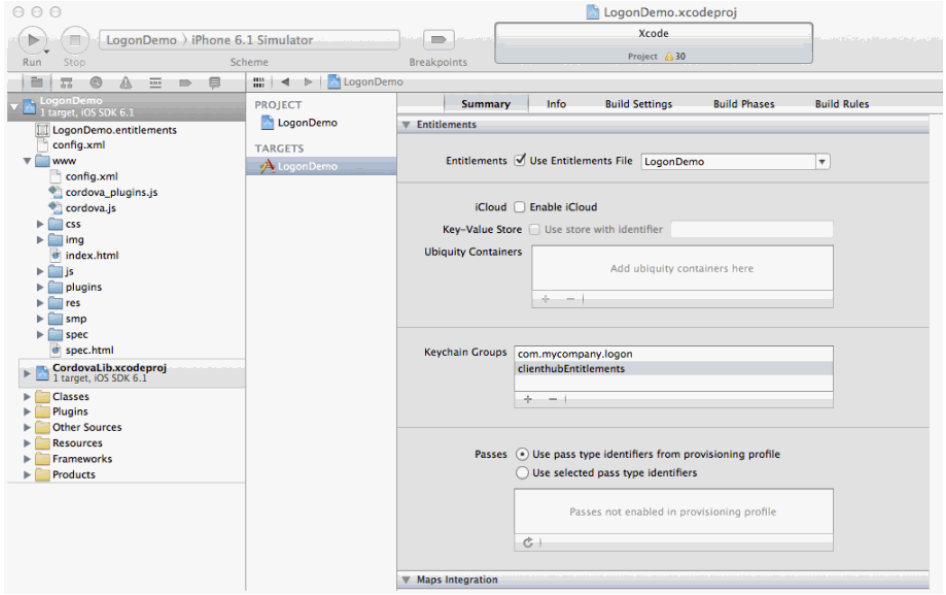
Deploy and run the Logon project on iOS.

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. Double-click the `<ProjectName>.xcodeproj` file to open the project in Xcode.
5. Add the **clienthubEntitlements** and **$(CFBundleIdentifier)** keychain groups to the Capabilities (Xcode 5) or Entitlements section of the project.

   This shows an example:



6. The Afaria client is opened after calling sap.Logon.init(...), but can be disabled by modifying the file `MAFLogonManagerOptions.plist`. In Xcode, locate this file under **Resources** > **MAFLogonManagerNG.bundle** > **MAFLogonManagerOptions.plist**, and set **keyMAFUseAfaria** to `false`.
7. 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 34	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 35	Get an (JSON serializable) object from the DataVault for a given key.
init(successCallback, errorCallback, applicationId, [context], [logonView]) on page 36	Initialization method to set up the Logon plugin.
lock(onsuccess, onerror) on page 39	Locks the Logon plugin's secure data vault.
managePasscode(onsuccess, onerror) on page 39	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 40	Set an (JSON serializable) object in the DataVault.
showRegistrationData(onsuccess, onerror) on page 41	Calling this method will show a screen which displays the current registration settings for the application.
unlock(onsuccess, onerror) on page 42	

Type Definitions

Name	Description
errorCallback(errorObject) on page 42	Callback function that is invoked in case of an error.
getSuccessCallback(value) on page 47	Callback function that is invoked upon successfully retrieving an object from the DataVault.
successCallback(context) on page 47	Callback function that is invoked upon successfully registering or unlocking or retrieving the context.
successCallbackNoParameters on page 52	Callback function that will be invoked with no parameters.

Source
LogonController.js, line 1476 on page 103.

applicationId member

The application ID with which *sap.Logon.init* on page 36 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 1534 on page 106.

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 1554 on page 107.

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 52	The callback to call if the screen flow succeeds.onsuccess will be called without parameters for this method.

onerror	*sap.Logon~errorCallback* on page 42	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 1708 on page 112.

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 47	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 42	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 1576 on page 107.

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 47			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 42			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 47 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 1526 on page 105.

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 52	The function to invoke upon success.
onerror	*sap.Logon~errorCallback* on page 42	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 1615 on page 109.

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 52	The function to invoke upon success.
onerror	*sap.Logon~errorCallback* on page 42	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 1689 on page 112.

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 52	The function to invoke upon success.onsuccess will be called without parameters for this method.
onerror	*sap.Logon~errorCallback* on page 42	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 1599 on page 108.

showRegistrationData(onsuccess, onerror) method

Calling this method will show a screen which displays the current registration settings for the application.

Syntax

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

Parameters

Name	Type	Description
onsuccess	*sap.Logon~successCallback-NoParameters* on page 52	The callback to call if the screen flow succeeds.onsuccess will be called without parameters for this method.
onerror	*sap.Logon~errorCallback* on page 42	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 1725 on page 113.

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 39 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 47	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 42	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 1638 on page 110.

errorCallback(errorObject) type

Callback function that is invoked in case of an error.

Syntax

`errorCallback`(*errorObject*)

Parameters

Name	Type	Description

errorObject	Object	Depending on the origin of the error the object can take several forms.(Unfortunately the error object structure and content is not uniform among the platforms, this will probably change in the future.) Errors originating from the logon plugin have only an 'errorKey' property. The possible values for 'errorKey': ERR_CHANGE_TIMEOUT_FAILED ERR_FORGOT_SSO_PIN ERR_INIT_FAILED ERR_INVALID_ACTION ERR_INVALID_STATE ERR_PASSCODE_REQUIRES_DIGIT ERR_PASSCODE_REQUIRES_LOWER ERR_PASSCODE_REQUIRES_SPECIAL ERR_PASSCODE_REQUIRES_UPPER ERR_PASSCODE_TOO_SHORT ERR_PASSCODE_UNDER_MIN_UNIQUE_CHARS ERR_REGISTRATION_CANCEL ERR_REG_FAILED ERR_REG_FAILED_UNATHORIZED ERR_REG_FAILED_WRONG_SERVER ERR_SETPASSCODE_FAILED ERR_SET_AFARIA_CREDENTIAL_FAILED ERR_SSO_PASSCODE_SET_ERROR ERR_UNKNOWN_SCREEN_ID ERR_UNLOCK_FAILED ERR_USER_CANCELLED Errors originating in the logon core (either iOS or Android) have the following 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. On iOS the 'errorDomain' property of the core errors can take the following values: MAFLogonCoreErrorDomain, MAFSecureStoreManagerErrorDomain, and MAFLogonCoreCDVPluginErrorDomain. In the MAFLogonCoreErrorDomain the following errors are thrown (throwing methods in paren): 3 errMAFLogonErrorCommunicationManagerError (register, update settings, delete, change backend password) 9 errMAFLogonErrorCouldNotDecideCommunicator (register) 11 errMAFLogonErrorOperationNotAllowed (all) 12 errMAFLogonErrorInvalidServerHost (register) 13 errMAFLogonErrorInvalidBackendPassword (changeBackendPassword) 15 errMAFLogonErrorUploadTraceFailed (uploadTrace) 16 errMAFLogonErrorInvalidMCIMSSOPin (setMCIMSSOPin) 18 errMAFLogonErrorCertificateKeyError (register) 19 errMAFLogonErrorCertificateError (register) 20 errMAFLogonErrorAfariaInvalidCredentials (setAfariaCredentialWithUser) In the MAFSecureStoreManagerErrorDomain the following errors are

			thrown (throwing methods in paren): 0 errMAFSecureStoreManagerErrorUnknown (persist, unlock, changePasscode, delete, getContext) 1 errMAFSecureStoreManagerErrorAlreadyExists (persist) 2 errMAFSecureStoreManagerErrorDataTypeError (unlock, getContext) 3 errMAFSecureStoreManagerErrorDoesNotExist (unlock, persist, getContext) 4 errMAFSecureStoreManagerErrorInvalidArg unlock, (persist, getContext) 5 errMAFSecureStoreManagerErrorInvalidPassword (unlock) 6 errMAFSecureStoreManagerErrorLocked (getContext) 7 errMAFSecureStoreManagerErrorOutOfMemory (persist, unlock, changePasscode, delete, getContext) 8 errMAFSecureStoreManagerErrorPasswordExpired (unlock, getContext) 9 errMAFSecureStoreManagerErrorPasswordRequired (persist, changePasscode) 10 errMAFSecureStoreManagerErrorPasswordRequiresDigit (persist, changePasscode) 11 errMAFSecureStoreManagerErrorPasswordRequiresLower (persist, changePasscode) 12 errMAFSecureStoreManagerErrorPasswordRequiresSpecial (persist, changePasscode) 13 errMAFSecureStoreManagerErrorPasswordRequiresUpper (persist, changePasscode) 14 errMAFSecureStoreManagerErrorPasswordUnderMinLength (persist, changePasscode) 15 errMAFSecureStoreManagerErrorPasswordUn-

		derMinUniqueChars (persist, changePasscode) 16 errMAF-SecureStoreManagerErrorDeleted (unlock) In the MAFLogonCoreCDVPluginErrorDomain the following errors are thrown: 1 (init failed) 2 (plugin not initialized) 3 (no input provided) On Android the 'errorDomain' property of the core errors can take the following values: MAFLogonCoreErrorDomain and MAFLogonCoreCDVPluginErrorDomain. There are no logon specific error codes, the 'errorCode' property only wraps the error values from the underlying libraries.

Source
LogonController.js, line 1728 on page 113.

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 1734 on page 113.

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 1730 on page 113.

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

Syntax
`successCallbackNoParameters()`

Source
LogonController.js, line 1732 on page 113.

Source code

LogonController.js

```
1       var utils = sap.logon.Utils;
2       var TIMEOUT = 2000;
3
4       var _oLogonCore;
5       var _oLogonView;
6       var _hasLogonSuccessEventFired = false;
7
8       var _providedContext;
9       var flowqueue;
10
11      var init = function (successCallback, errorCallback,
applicationId, context, customView) {
12
13          document.addEventListener("resume",
14                                    function(){
15                                    resume(
16                                            function()
{ fireEvent('onSapResumeSuccess', arguments);},
17                                            function()
{ fireEvent('onSapResumeError', arguments);}
18                                            );
19                                    },
20                                    false);
21
22          // The success callback used for the call to
_oLogonCore.initLogon(...)
23          var initSuccess = function(){
24              utils.log('LogonController: LogonCore successfully
initialized.');
25
26            // Now that Logon is initialized, registerOrUnlock is
automatically called.
27              registerOrUnlock( successCallback, errorCallback );
```

```
28          }
29
30          var initError = function(error){
31             // If a parameter describing the error is given, pass
it along.
32              // Otherwise, construct something to call the error
callback with.
33              if( error ) {
34                  errorCallback( error );
35              } else {
36
errorCallback( utils.Error('ERR_INIT_FAILED') );
37              }
38          }
39
40
41          utils.log('LogonController.init enter');
42          utils.log(applicationId);
43          module.exports.applicationId = applicationId;
44
45          // Make note of the context given (if any)
46          if( context ){
47              _providedContext = context;
48          }
49
50          _oLogonView = customView;
51          if (!_oLogonView) {
52              _oLogonView = sap.logon.IabUi;
53          }
54
55          flowqueue = new FlowRunnerQueue();
56
```

```
57          //
coLogonCore.cordova.require("com.sap.mp.cordova.plugins.logon.Logon
Core");
58          _oLogonCore = sap.logon.Core;
59          _oLogonCore.initLogon(initSuccess, initError,
applicationId);
60
61          //update exports definition
62          module.exports.core = _oLogonCore;
63      }
64
65      var fireEvent = function (eventId, args) {
66          if (typeof eventId === 'string') {
67              //var event = document.createEvent('Events');
68              //event.initEvent(eventId, false, false);
69
70              if (!window.CustomEvent) {
71                window.CustomEvent = function(type, eventInitDict)
{
72                      var newEvent =
document.createEvent('CustomEvent');
73                      newEvent.initCustomEvent(
74                                                  type,
75                                                 !!(eventInitDict &&
eventInitDict.bubbles),
76                                                 !!(eventInitDict &&
eventInitDict.cancelable),
77                                                  (eventInitDict ?
eventInitDict.detail : null));
78                      return newEvent;
79                  };
80              }
81
82            var event = new CustomEvent(eventId, { 'detail':{ 'id':
eventId, 'args': args }});
83
```

```
84              setTimeout(function() {
85                          document.dispatchEvent(event);
86                          }, 0);
87          } else {
88              throw 'Invalid eventId: ' + JSON.stringify(event);
89          }
90      }
91
92      var FlowRunner = function(callbacks, pLogonView, pLogonCore,
flowClass) {
93
94          var onFlowSuccess;
95          var onFlowError;
96          var onFlowCancel;
97
98          var logonView;
99          var logonCore;
100         var flow;
101
102         var onsuccess = callbacks.onsuccess;
103         var onerror = callbacks.onerror;
104
105
106
107         logonView = pLogonView;
108         logonCore = pLogonCore;
109
110         onFlowSuccess = function onFlowSuccess() {
111             utils.logJSON('onFlowSuccess');
112             logonView.close();
113             onsuccess.apply(this, arguments);
114         }
```

```
115
116         onFlowError = function onFlowError() {
117             utils.logJSON('onFlowError');
118             logonView.close();
119             onerror.apply(this, arguments);
120         }
121
122         onFlowCancel = function onFlowCancel(){
123             utils.logJSON('onFlowCancel');
124             //logonView.close();
125             onFlowError(new utils.Error('ERR_USER_CANCELLED'));
126         }
127
128         var handleCoreStateOnly = function(currentState){
129             handleCoreResult(null, currentState);
130         }
131
132         var handleCoreResult = function (currentContext,
currentState) {
133             if (typeof currentContext === undefined)
currentContext = null;
134
135             //workaround for defaultPasscodeAllowed
136             if (currentState) {
137               if (currentContext && currentContext.policyContext
&& currentContext.policyContext.defaultAllowed){
138                     currentState.defaultPasscodeAllowed = true;
139                 }
140                 else {
141                     currentState.defaultPasscodeAllowed =
false;
142                 }
143             }
144
```

```
145             utils.logJSON(currentContext, 'handleCoreResult
currentContext');
146             utils.logJSON(currentState, 'handleCoreResult
currentState');
147
148
149             utils.logJSON(flow.name);
150             var matchFound = false;
151             var rules = flow.stateTransitions;
152
153
154         ruleMatching:
155             for (key in rules){
156
157                 var rule = flow.stateTransitions[key];
158                 //utils.logJSON(rule, 'rule');
159
160                 //utils.logJSON(rule.condition,
'rule.condition');
161                 if (typeof rule.condition === 'undefined') {
162                   throw 'undefined condition in state transition
rule';
163                 }
164
165
166                 if (rule.condition.state === null) {
167                     if (currentState)
168                     {
169                        continue ruleMatching; // non-null state
(and rule) mismatch
170                     }
171                     //else {
172                     //    // match:
173                     //    // rule.condition.state === null &&
```

```
174                     //    // (typeof currentState ===
'undefined') // null or undefined
175                     //}
176                 }
177                 else if (rule.condition.state !== 'undefined' &&
currentState){
178                     utils.log('stateMatching');
179
180                  stateMatching:
181                     for (field in rule.condition.state) {
182                         utils.log(field);
183                         if (rule.condition.state[field] ===
currentState[field])
184                         {
185                             utils.log('field matching ' +
field);
186                          continue stateMatching; // state field
match
187                         }
188                         else {
189                             utils.log('field mismatching ' +
field);
190                           continue ruleMatching; // state field
(and rule) mismatch
191                         };
192                     }
193                 }
194
195                 if (rule.condition.context === null) {
196                     if (currentContext)
197                     {
198                       continue ruleMatching; // non-null context
(and rule) mismatch
199                     }
200                     //else {
```

```
201                     //    // match:
202                     //    // rule.condition.context === null &&
203                     //    // (typeof currentContext ===
'undefined') // null or undefined
204                     //}
205                 }
206               else if (rule.condition.context !== 'undefined' &&
currentContext){
207
208                     utils.log('contextMatching');
209                 contextMatching:
210                     for (field in rule.condition.context) {
211                         utils.log(field);
212                         if (rule.condition.context[field] ===
currentContext[field])
213                         {
214                             utils.log('field matching ' +
field);
215                           continue contextMatching;  // context
field match
216                         }
217                         else {
218                             utils.log('field mismatching ' +
field);
219                         continue ruleMatching;  // context field
(and rule) mismatch
220                         };
221                     }
222                 }
223                 utils.log('match found');
224                 utils.logJSON(rule, 'rule');
225
226                 if (typeof rule.action === 'function') {
227                     rule.action(currentContext);
228                 }
```

```
229                 else if (typeof rule.action === 'string') {
230                     // the action is a screenId
231                     var screenId = rule.action;
232                     utils.log('handleCoreResult: ' + screenId);
233                     utils.logKeys(flow.screenEvents[screenId]);
234                     if(!currentContext){
235                         currentContext = {};
236                     }
237
238                     if( !currentContext.registrationContext &&
_providedContext ){
239                      // The current registrationContext is null,
and we have been given a context when initialized,
240                         // so use the one we were given.
241                         currentContext.registrationContext =
_providedContext;
242                   } else if (currentContext.registrationContext
&& _providedContext && !currentContext.registrationReadOnly && !
(currentState.stateAfaria=='initializationSuccessful')){
243                         for (key in _providedContext) {
244                             //if (!
currentContext.registrationContext[key]){
245                         currentContext.registrationContext[key]
= _providedContext[key];
246                             //}
247                         }
248                     }
249
250                     logonView.showScreen(screenId,
flow.screenEvents[screenId], currentContext);
251
252                 }
253                 else {
254                     onFlowError(new
utils.Error('ERR_INVALID_ACTION'));
255                 }
```

```
256
257                 matchFound = true;
258                 break ruleMatching;
259             }
260
261             if (!matchFound) {
262                 onFlowError(new
utils.Error('ERR_INVALID_STATE'));
263             }
264         }
265
266         flow = new flowClass(logonCore, logonView,
handleCoreResult, onFlowSuccess, onFlowError, onFlowCancel);
267
268         this.run = function() {
269             utils.log('FlowRunner.run '  + flowClass.name);
270             utils.logKeys(flow , 'new flow ');
271             logonCore.getState(handleCoreStateOnly,
onFlowError);
272         }
273     }
274
275     var FlowRunnerQueue = function() {
276         var isRunning = false;
277         var innerQueue = [];
278
279         this.add = function(flowRunner) {
280             innerQueue.push(flowRunner);
281             if (isRunning == false) {
282                 isRunning = true;
283                 process();
284             }
285         }
```

```
286
287         this.runNextFlow = function() {
288             innerQueue.shift();
289             if (innerQueue.length == 0) {
290                 isRunning = false;
291             }
292             else {
293                 process();
294             }
295          }
296
297          var process = function() {
298             if (innerQueue.length > 0) {
299                 var flowRunner = innerQueue[0];
300                 flowRunner.run();
301             }
302             else {
303                 isRunning = false;
304             }
305          }
306      }
307
308
309      var MockFlow = function MockFlow(logonCore, logonView,
onCoreResult, onFlowSuccess, onFlowError, onFlowCancel) {
310          //wrapped into a function to defer evaluation of the
references to flow callbacks
311          //var flow = {};
312
313          this.name = 'mockFlowBuilder';
314
315          this.stateTransitions = [
316                                      {
```

```
317                                     condition: {
318                                     state: {
319                                     secureStoreOpen: false,
320                                     }
321                                     },
322                                     action: 'SCR_MOCKSCREEN'
323                                     },
324                                     {
325                                     condition: {
326                                     state: {
327                                     secureStoreOpen: true,
328                                     }
329                                     },
330                                     action: 'SCR_MOCKSCREEN'
331                                     },
332
333                                     ];
334
335         this.screenEvents = {
336             'SCR_TURN_PASSCODE_ON': {
337             onsubmit: onFlowSuccess,
338             oncancel: onFlowCancel,
339             onerror: onFlowError,
340             }
341         };
342
343         utils.log('flow constructor return');
344         //return flow;
345     }
346
347     var RegistrationFlow = function RegistrationFlow(logonCore,
logonView, onCoreResult, onFlowSuccess, onFlowError, onFlowCancel)
{
```

```
348         //wrapped into a function to defer evaluation of the
references to flow callbacks
349
350         this.name = 'registrationFlowBuilder';
351
352         var registrationInProgress = false;
353
354         var onCancelSSOPin = function() {
355
onFlowError(errorWithDomainCodeDescription("MAFLogon","0","SSO
Passcode set screen was cancelled"));
356         }
357
358         var onCancelRegistration = function() {
359
onFlowError(errorWithDomainCodeDescription("MAFLogon","1","Registra
tion screen was cancelled"));
360         }
361
362         // internal methods
363         var showScreen = function(screenId) {
364             return function(coreContext) {
365                 logonView.showScreen(screenId,
this.screenEvents[screenId], coreContext);
366             }.bind(this);
367         }.bind(this);
368
369         var onUnlockSubmit = function(context){
370             utils.logJSON(context,
'logonCore.unlockSecureStore');
371             logonCore.unlockSecureStore(onCoreResult,
onUnlockError, context)
372         }
373
374         var onUnlockError = function(error) {
```

```
375             utils.logJSON("onUnlockError: " +
JSON.stringify(error));

376

377             // TODO switch case according to the error codes

378             if (error

379                   && error.errorDomain && error.errorDomain ===
"MAFSecureStoreManagerErrorDomain"

380                  && error.errorCode && error.errorCode === "16")
{

381                     // Too many attempts --> DV deleted

382
logonView.showNotification("ERR_TOO_MANY_ATTEMPTS_APP_PASSCODE")

383             }

384             else {

385
logonView.showNotification("ERR_UNLOCK_FAILED");

386             }

387

388         }

389

390         var onSetAfariaCredentialError = function(error) {

391             utils.logJSON("onSetAfariaCredentialError: " +
JSON.stringify(error));

392

393
logonView.showNotification("ERR_SET_AFARIA_CREDENTIAL_FAILED");

394         }

395

396         var noOp = function() { }

397

398         var onErrorAck = function(ack) {

399           if (ack.key === 'ERR_TOO_MANY_ATTEMPTS_APP_PASSCODE')
{

400                     onFlowError(new
utils.Error('ERR_TOO_MANY_ATTEMPTS_APP_PASSCODE'));

401             }
```

```
402         }
403
404
405         var onRegistrationBackButton = function() {
406             if (registrationInProgress == true) {
407                 utils.log('back button pushed, no operation is
required as registration is running');
408             }
409             else {
410                 onCancelRegistration();
411             }
412         }
413
414         var onUnlockVaultWithDefaultPasscode = function(){
415             utils.log('logonCore.unlockSecureStore - default
passcode');
416             var unlockContext = {"unlockPasscode":null};
417           logonCore.unlockSecureStore(onCoreResult, onFlowError,
unlockContext)
418         }
419
420         var onRegSucceeded = function(context, state) {
421             onCoreResult(context, state);
422             registrationInProgress = false;
423         }
424
425         var onRegError = function(error){
426             utils.logJSON(error, 'registration failed');
427
logonView.showNotification(getRegistrationErrorText(error));
428             registrationInProgress = false;
429         }
430
431         var onRegSubmit = function(context){
```

```
432             utils.logJSON(context,
'logonCore.startRegistration');

433             registrationInProgress = true;

434             logonCore.startRegistration(onRegSucceeded,
onRegError, context)

435         }

436

437         var onCreatePasscodeSubmit = function(context){

438             utils.logJSON(context,
'logonCore.persistRegistration');

439             logonCore.persistRegistration(onCoreResult,
onCreatePasscodeError, context);

440         }

441

442         var onCancelRegistrationError = function(error){

443             utils.logJSON("onCancelRegistrationError: " +
JSON.stringify(error));

444
logonView.showNotification(getRegistrationCancelError(error));

445         }

446

447         var onCreatePasscodeError = function(error) {

448             utils.logJSON("onCreatePasscodeError: " +
JSON.stringify(error));

449
logonView.showNotification(getSecureStoreErrorText(error));

450         }

451

452         var onSSOPasscodeSetError = function(error) {

453             utils.logJSON("onSSOPasscodeSetError: " +
JSON.stringify(error));

454
logonView.showNotification(getSSOPasscodeSetErrorText(error));

455         }

456

457         var callGetContext = function(){
```

```
458             utils.log('logonCore.getContext');
459             logonCore.getContext(onCoreResult, onFlowError);
460         }
461
462         var onFullRegistered = function()
463         {
464             var getContextSuccessCallback = function(result){
465
466                 if(!_hasLogonSuccessEventFired) {
467                     fireEvent("onSapLogonSuccess", arguments);
468                     _hasLogonSuccessEventFired = true;
469                 }
470
471                 onFlowSuccess(result);
472             }
473             utils.log('logonCore.getContext');
474             logonCore.getContext(getContextSuccessCallback,
onFlowError);
475         }
476
477         var onForgotAppPasscode = function(){
478             utils.log('logonCore.deleteRegistration');
479             logonCore.deleteRegistration(onFlowError,
onFlowError);
480         }
481
482         var onForgotSsoPin = function(){
483             utils.log('forgotSSOPin');
484             logonView.showNotification("ERR_FORGOT_SSO_PIN");
485         }
486
487         var onSkipSsoPin = function(){
488             utils.logJSON('logonCore.skipClientHub');
```

```
489             logonCore.skipClientHub(onCoreResult, onFlowError);
490         }
491
492         var callPersistWithDefaultPasscode = function(context){
493             utils.logJSON(context,
'logonCore.persistRegistration');
494             context.passcode = null;
495             logonCore.persistRegistration(
496                                                 onCoreResult,
497                                                 onFlowError,
498                                                 context)
499         }
500
501         // exported properties
502         this.stateTransitions = [
503                                     {
504                                     condition: {
505                                     state: {
506                                     secureStoreOpen: false,
507                                     status: 'fullRegistered',
508                                     defaultPasscodeUsed: true
509                                     }
510                                     },
511                                     action:
onUnlockVaultWithDefaultPasscode
512                                     },
513
514                                     {
515                                     condition: {
516                                     state: {
517                                     secureStoreOpen: false,
518                                     status: 'fullRegistered'
519                                     }
```

```
520                                     },
521                                     action: 'SCR_UNLOCK'
522                                     },
523
524
525                                     {
526                                     condition: {
527                                     state: {
528                                    //secureStoreOpen: false, //TODO
clarify
529                                     status: 'fullRegistered',
530                                     stateClientHub:
'availableNoSSOPin'
531                                     }
532                                     },
533                                     action: 'SCR_SSOPIN_SET'
534                                     },
535                                     {
536                                     condition: {
537                                     state: {
538                                     status: 'new'
539                                     },
540                                     context: null
541                                     },
542                                     action: callGetContext
543                                     },
544
545                                     {
546                                     condition: {
547                                     state: {
548                                     status: 'new',
549                                     stateClientHub:
'availableNoSSOPin'
```

```
550                                     }
551                                     },
552                                     action: 'SCR_SSOPIN_SET'
553                                     },
554
555                                     {
556                                     condition: {
557                                     state: {
558                                     status: 'new',
559                                     stateClientHub:
'availableInvalidSSOPin'
560                                     }
561                                     },
562                                     action: 'SCR_SSOPIN_SET'
563                                     },
564                                     {
565                                     condition: {
566                                     state: {
567                                     status: 'new',
568                                     stateClientHub:
'availableValidSSOPin',
569                                     stateAfaria:
'credentialNeeded'
570                                     },
571                                     context : {
572                                     afariaRegistration:
'certificate'
573                                     }
574                                     },
575                                     action:
'SCR_ENTER_AFARIA_CREDENTIAL'
576                                     },
577                                     {
578                                     condition: {
```

```
579                                     state: {
580                                     status: 'new',
581                                     stateClientHub:
'notAvailable',
582                                     stateAfaria:
'credentialNeeded'
583                                     }
584                                     },
585                                     action:
'SCR_ENTER_AFARIA_CREDENTIAL'
586                                     },
587                                     {
588                                     condition: {
589                                     state: {
590                                     status: 'new',
591                                     isAfariaCredentialsProvided:
false
592                                     },
593                                     context : {
594                                     afariaRegistration:
'certificate'
595                                     }
596                                     },
597                                     action:
'SCR_ENTER_AFARIA_CREDENTIAL'
598                                     },
599                                     {
600                                     condition: {
601                                     state: {
602                                     status: 'new',
603                                     stateClientHub:
'availableValidSSOPin'
604                                     },
605                                     context : {
606                                     credentialsByClientHub : true,
```

```
607                                     registrationReadOnly : true
608                                     }
609                                     },
610                                     action: function(context){
611                                     utils.logJSON(context,
'logonCore.startRegistration');
612
logonCore.startRegistration(onCoreResult, onRegError,
context.registrationContext);
613                                     }
614                                     },
615                                     {
616                                     condition: {
617                                     state: {
618                                     status: 'new',
619                                     stateClientHub:
'availableValidSSOPin',
620                                     isAfariaCredentialsProvided:
true
621                                     },
622                                     context : {
623                                     afariaRegistration:
'certificate',
624                                     registrationReadOnly : true
625                                     }
626                                     },
627                                     action: function(context){
628                                     utils.logJSON(context,
'logonCore.startRegistration');
629
logonCore.startRegistration(onCoreResult, onRegError,
context.registrationContext);
630                                     }
631                                     },
632                                     {
633                                     condition: {
```

```
634                                     state: {
635                                     status: 'new',
636                                     stateClientHub:
'availableValidSSOPin',
637                                     stateAfaria:
'initializationSuccessful'
638                                     },
639                                     context : {
640                                     registrationReadOnly : true,
641                                     afariaRegistration:
'certificate'
642                                     }
643                                     },
644                                     action: function(context){
645                                     utils.logJSON(context,
'logonCore.startRegistration');
646
logonCore.startRegistration(onCoreResult, onRegError,
context.registrationContext);
647                                     }
648                                     },
649                                     {
650                                     condition: {
651                                     state: {
652                                     status: 'new',
653                                     stateClientHub:
'notAvailable',
654                                     stateAfaria:
'initializationSuccessful'
655                                     },
656                                     context : {
657                                     afariaRegistration:
'certificate'
658                                     }
659                                     },
660                                     action: function(context){
```

```
661                                     utils.logJSON(context,
'logonCore.startRegistration');
662
logonCore.startRegistration(onCoreResult, onRegError,
context.registrationContext);
663                                     }
664                                     },
665                                     {
666                                     condition: {
667                                     state: {
668                                     status: 'new',
669                                     stateAfaria:
'initializationSuccessful'
670                                     }
671                                     },
672                                     action:
'SCR_ENTER_CREDENTIALS'
673                                     },
674                                     {
675                                     condition: {
676                                     state: {
677                                     status: 'new',
678                                     stateClientHub:
'availableValidSSOPin'
679                                     },
680                                     context : {
681                                     registrationReadOnly :true,
682                                     credentialsByClientHub : false
683                                     }
684                                     },
685                                     action:
'SCR_ENTER_CREDENTIALS'
686                                     },
687
688
```

```
689                                     {
690                                     condition: {
691                                     state: {
692                                     status: 'new',
693                                   //stateClientHub: 'notAvailable' |
'availableValidSSOPin' | 'skipped' | 'error'
694                                     stateAfaria:
'initializationFailed'
695                                     }
696                                     },
697                                     action: 'SCR_REGISTRATION'
698                                     },
699
700                                     {
701                                     condition: {
702                                     state: {
703                                     status: 'new',
704                                   //stateClientHub: 'notAvailable' |
'availableValidSSOPin' | 'skipped' | 'error'
705                                     }
706                                     },
707                                     action: 'SCR_REGISTRATION'
708                                     },
709
710                                     {
711                                     condition: {
712                                     state: {
713                                     status: 'new',
714                                   //stateClientHub: 'notAvailable' |
'availableValidSSOPin' | 'skipped' | 'error'
715                                     stateAfaria:
'initializationFailed'
716                                     }
717                                     },
```

```
718                                     action: 'SCR_REGISTRATION'
719                                     },
720
721                                     {
722                                     condition: {
723                                     state: {
724                                     secureStoreOpen: false,
725                                     status: 'registered',
726                                     defaultPasscodeUsed: true,
727                                     //
defaultPasscodeAllowed: true,
728                                     }
729                                     },
730                                     action:
'SCR_SET_PASSCODE_OPT_OFF'
731                                     },
732                                     {
733                                     condition: {
734                                     state: {
735                                     secureStoreOpen: false,
736                                     status: 'registered',
737                                     defaultPasscodeUsed: false,
738                                     defaultPasscodeAllowed: true,
739                                     }
740                                     },
741                                     action:
'SCR_SET_PASSCODE_OPT_ON'
742                                     },
743                                     {
744                                     condition: {
745                                     state: {
746                                     secureStoreOpen: false,
747                                     status: 'registered',
```

```
748                                     //
defaultPasscodeAllowed: false,
749                                     }
750                                     },
751                                     action:
'SCR_SET_PASSCODE_MANDATORY'
752                                     },
753
754
755                                     {
756                                     condition: {
757                                     state: {
758                                    //secureStoreOpen: false, //TODO
clarify
759                                     status: 'fullRegistered',
760                                     stateClientHub:
'availableInvalidSSOPin'
761                                     }
762                                     },
763                                     action: 'SCR_SSOPIN_CHANGE'
764                                     },
765                                     {
766                                     condition: {
767                                     state: {
768                                     secureStoreOpen: true,
769                                     status: 'fullRegistered',
770                                     stateClientHub: 'notAvailable'
771                                     }
772                                     },
773                                     action: onFullRegistered
774                                     },
775                                     {
776                                     condition: {
777                                     state: {
```

```
778                                     secureStoreOpen: true,
779                                     status: 'fullRegistered',
780                                     stateClientHub:
'availableValidSSOPin'
781                                     }
782                                     },
783                                     action: onFullRegistered
784                                     },
785                                     {
786                                     condition: {
787                                     state: {
788                                     secureStoreOpen: true,
789                                     status: 'fullRegistered',
790                                     stateClientHub: 'skipped'
791                                     }
792                                     },
793                                     action: onFullRegistered
794                                     },
795
796
797
798                                     ];
799
800         this.screenEvents = {
801             'SCR_SSOPIN_SET': {
802             onsubmit: function(context){
803                 utils.logJSON(context,
'logonCore.setSSOPasscode');
804                 logonCore.setSSOPasscode(onCoreResult,
onSSOPasscodeSetError, context);
805             },
806             oncancel: onCancelSSOPin,
807             onerror: onFlowError,
```

```
808             onforgot: onForgotSsoPin,
809             onskip: onSkipSsoPin
810             },
811
812             'SCR_ENTER_AFARIA_CREDENTIAL' : {
813             onsubmit: function(context){
814                 utils.logJSON(context,
'logonCore.setAfariaCredential');
815                 logonCore.setAfariaCredential(onCoreResult,
onSetAfariaCredentialError, context);
816             }
817             },
818
819             'SCR_SSOPIN_CHANGE': {
820             onsubmit: function(context){
821                 utils.logJSON(context,
'logonCore.setSSOPasscode');
822                 logonCore.setSSOPasscode(onCoreResult,
onSSOPasscodeSetError, context);
823             },
824             oncancel: onSkipSsoPin,
825             onerror: onFlowError,
826             onforgot: onForgotSsoPin
827             },
828
829             'SCR_UNLOCK': {
830             onsubmit: onUnlockSubmit,
831             oncancel: noOp,
832             onerror: onFlowError,
833             onforgot: onForgotAppPasscode,
834             onerrorack: onErrorAck
835             },
836
837             'SCR_REGISTRATION':  {
```

```
838             onsubmit: onRegSubmit,
839             oncancel: onCancelRegistration,
840             onerror: onFlowError,
841             onbackbutton: onRegistrationBackButton
842             },
843
844             'SCR_ENTER_CREDENTIALS' : {
845             onsubmit: onRegSubmit,
846             oncancel: onCancelRegistration,
847             onerror: onFlowError
848             },
849             'SCR_SET_PASSCODE_OPT_ON': {
850             onsubmit: onCreatePasscodeSubmit,
851             oncancel: noOp,
852             onerror: onFlowError,
853             ondisable: showScreen('SCR_SET_PASSCODE_OPT_OFF'),
854             onerrorack: noOp
855             },
856             'SCR_SET_PASSCODE_OPT_OFF': {
857             onsubmit: callPersistWithDefaultPasscode,
858             oncancel: noOp,
859             onerror: onFlowError,
860             onenable: showScreen('SCR_SET_PASSCODE_OPT_ON'),
861             onerrorack: noOp
862             },
863             'SCR_SET_PASSCODE_MANDATORY': {
864             onsubmit: onCreatePasscodeSubmit,
865             oncancel: noOp,
866             onerror: onFlowError,
867             onerrorack: noOp
868             },
869
```

```
870
871
872         };
873
874
875         utils.log('flow constructor return');
876     }
877
878
879
880     var ChangePasswordFlow = function
ChangePasswordFlow(logonCore, logonView, onCoreResult,
onFlowSuccess, onFlowError, onFlowCancel) {
881         //wrapped into a function to defer evaluation of the
references to flow callbacks
882
883         this.name = 'changePasswordFlowBuilder';
884
885
886         // internal methods
887
888         var callUnlockFlow = function(){
889             utils.log(this.name + ' triggered unlock');
890             registerOrUnlock(onCoreResult,onFlowError);
891         }
892
893         var onChangePasswordSubmit = function(context){
894             utils.logJSON(context, 'logonCore.changePassword');
895             // this logonCore call does not return with context
896             logonCore.changePassword(onPasswordChanged,
onFlowError, context);
897         }
898
899
```

```
900         var onPasswordChanged = function(){
901             utils.log('onPasswordChanged');
902             logonCore.getContext(onFlowSuccess, onFlowError);
903         }
904
905         // exported properties
906         this.stateTransitions = [
907                                     {
908                                     condition: {
909                                     state: {
910                                     secureStoreOpen: false,
911                                     }
912                                     },
913                                     action: callUnlockFlow,
914                                     },
915                                     {
916                                     condition: {
917                                     state: {
918                                     secureStoreOpen: true,
919                                     }
920                                     },
921                                     action: 'SCR_CHANGE_PASSWORD'
922                                     },
923
924                                     ];
925
926         this.screenEvents = {
927             'SCR_CHANGE_PASSWORD': {
928             onsubmit: onChangePasswordSubmit,
929             oncancel: onFlowCancel,
930             onerror: onFlowError
931             }
```

```
932         };
933
934
935         utils.log('flow constructor return');
936     }
937
938     var ManagePasscodeFlow = function
ManagePasscodeFlow(logonCore, logonView, onCoreResult,
onFlowSuccess, onFlowError, onFlowCancel) {
939         //wrapped into a function to defer evaluation of the
references to flow callbacks
940
941         this.name = 'managePasscodeFlowBuilder';
942
943         // internal methods
944         var showScreen = function(screenId) {
945             return function(coreContext) {
946                 logonView.showScreen(screenId,
this.screenEvents[screenId], coreContext);
947             }.bind(this);
948         }.bind(this);
949
950
951         var callSetPasscode = function(context){
952             utils.logJSON(context, 'logonCore.changePasscode');
953             logonCore.changePasscode(
954                                     onCoreResult,
955                                     onChangePasscodeError,
956                                     context)
957         }
958
959         var callChangePasscode = function(context){
960             utils.logJSON(context, 'logonCore.changePasscode');
961             context.passcode = context.newPasscode;
```

```
962             logonCore.changePasscode(
963                                         onCoreResult,
964                                         onChangePasscodeError,
965                                         context)
966         }
967
968         var onChangePasscodeError = function(error) {
969             utils.logJSON("onChangePasscodeError: " +
JSON.stringify(error));
970
logonView.showNotification(getSecureStoreErrorText(error));
971         }
972
973         var noOp = function() { }
974
975         var callDisablePasscode = function(context){
976             utils.logJSON(context,
'logonCore.disablePasscode');
977             context.passcode = null;
978             logonCore.changePasscode(
979                                         onCoreResult,
980                                         onFlowError,
981                                         context)
982         }
983
984         var callGetContext = function(){
985             utils.log('logonCore.getContext');
986             logonCore.getContext(onCoreResult, onFlowError);
987         }
988
989         var onPasscodeEnable = function(context){
990           utils.logJSON(context, this.name + ' onPasscodeEnable:
');
```

```
991           //logonCore.changePasscode(onFlowSuccess, onFlowError,
context);
992              onFlowError();
993          }
994
995          // exported properties
996          this.stateTransitions = [
997                                       {
998                                       condition: {
999                                       state: {
1000                                      secureStoreOpen: true,
1001                                      },
1002                                      context: null
1003                                      },
1004                                      action: callGetContext
1005                                      },
1006                                      {
1007                                      condition: {
1008                                      state: {
1009                                      secureStoreOpen: false,
1010                                      }
1011                                      },
1012                                      action: onFlowError
1013                                      },
1014                                      {
1015                                      condition: {
1016                                      state: {
1017                                      secureStoreOpen: true,
1018                                      defaultPasscodeUsed: true,
1019                                      //
defaultPasscodeAllowed: true,
1020                                      }
1021                                      },
```

```
1022                                    action:
'SCR_MANAGE_PASSCODE_OPT_OFF'
1023                                    },
1024                                    {
1025                                    condition: {
1026                                    state: {
1027                                    secureStoreOpen: true,
1028                                    defaultPasscodeUsed: false,
1029                                    defaultPasscodeAllowed: true,
1030                                    }
1031                                    },
1032                                    action:
'SCR_MANAGE_PASSCODE_OPT_ON'
1033                                    },
1034                                    {
1035                                    condition: {
1036                                    state: {
1037                                    secureStoreOpen: true,
1038                                    //defaultPasscodeUsed:
[DONTCARE],
1039                                    defaultPasscodeAllowed: false,
1040                                    }
1041                                    },
1042                                    action:
'SCR_MANAGE_PASSCODE_MANDATORY'
1043                                    },
1044
1045
1046                                    ];
1047
1048        this.screenEvents = {
1049            'SCR_MANAGE_PASSCODE_OPT_ON': {
1050            onsubmit: onFlowSuccess,
1051            oncancel: onFlowSuccess,
```

```
1052            onerror: onFlowError,
1053            ondisable:
showScreen('SCR_CHANGE_PASSCODE_OPT_OFF'),
1054            onchange: showScreen('SCR_CHANGE_PASSCODE_OPT_ON')
1055            },
1056            'SCR_MANAGE_PASSCODE_OPT_OFF': {
1057            onsubmit: onFlowSuccess,
1058            oncancel: onFlowSuccess,
1059            onerror: onFlowError,
1060            onenable: showScreen('SCR_SET_PASSCODE_OPT_ON')
1061            },
1062            'SCR_MANAGE_PASSCODE_MANDATORY': {
1063            onsubmit: onFlowSuccess,
1064            oncancel: onFlowSuccess,
1065            onerror: onFlowError,
1066            onchange:
showScreen('SCR_CHANGE_PASSCODE_MANDATORY')
1067            },
1068
1069
1070            'SCR_SET_PASSCODE_OPT_ON': {
1071            onsubmit: callSetPasscode,
1072            oncancel: onFlowCancel,
1073            onerror: onFlowError,
1074            ondisable: showScreen('SCR_SET_PASSCODE_OPT_OFF'),
1075            onerrorack: noOp
1076            },
1077            'SCR_SET_PASSCODE_OPT_OFF': {
1078            onsubmit: callDisablePasscode,
1079            oncancel: onFlowCancel,
1080            onerror: onFlowError,
1081            onenable: showScreen('SCR_SET_PASSCODE_OPT_ON'),
1082            onerrorack: noOp
```

```
1083            },
1084            'SCR_CHANGE_PASSCODE_OPT_ON': {
1085            onsubmit: callChangePasscode,
1086            oncancel: onFlowCancel,
1087            onerror: onFlowError,
1088            ondisable:
showScreen('SCR_CHANGE_PASSCODE_OPT_OFF'),
1089            onerrorack: noOp
1090            },
1091            'SCR_CHANGE_PASSCODE_OPT_OFF': {
1092            onsubmit: callDisablePasscode,
1093            oncancel: onFlowCancel,
1094            onerror: onFlowError,
1095            onenable: showScreen('SCR_CHANGE_PASSCODE_OPT_ON'),
1096            onerrorack: noOp
1097            },
1098            'SCR_CHANGE_PASSCODE_MANDATORY': {
1099            onsubmit: callChangePasscode,
1100            oncancel: onFlowCancel,
1101            onerror: onFlowError,
1102            onerrorack: noOp
1103            },
1104
1105        };
1106
1107
1108        utils.log('flow constructor return');
1109    }
1110
1111    var ShowRegistrationFlow = function
ShowRegistrationFlow(logonCore, logonView, onCoreResult,
onFlowSuccess, onFlowError, onFlowCancel) {
1112        //wrapped into a function to defer evaluation of the
references to flow callbacks
```

```
1113
1114        this.name = 'showRegistrationFlowBuilder';
1115
1116        var showRegistrationInfo = function(context) {
1117            logonView.showScreen('SCR_SHOW_REGISTRATION',
this.screenEvents['SCR_SHOW_REGISTRATION'], context);
1118        }.bind(this);
1119
1120        var callGetContext = function(){
1121            utils.log('logonCore.getContext');
1122            logonCore.getContext(onCoreResult, onFlowError);
1123        }
1124
1125        // exported properties
1126        this.stateTransitions = [
1127                                    {
1128                                    condition: {
1129                                    state: {
1130                                    secureStoreOpen: true,
1131
1132                                    },
1133                                    context: null
1134                                    },
1135                                    action: callGetContext
1136                                    },
1137                                    {
1138                                    condition: {
1139                                    secureStoreOpen: true,
1140                                    },
1141                                    action: showRegistrationInfo
1142                                    }
1143
```

```
1144                                    ];
1145
1146        this.screenEvents = {
1147            'SCR_SHOW_REGISTRATION': {
1148            oncancel: onFlowSuccess,
1149            onerror: onFlowError
1150            }
1151        };
1152
1153
1154        utils.log('flow constructor return');
1155    }
1156
1157    // === flow launcher methods
==================================
1158
1159
1160    var resume = function (onsuccess, onerror) {
1161
1162        if(!_oLogonCore) {
1163            utils.log('FlowRunner.run MAFLogon is not
initialized');
1164
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1165            return;
1166        }
1167
1168        var onUnlockSuccess = function(){
1169            _oLogonCore.onEvent(onsuccess, onerror, 'RESUME');
1170        }
1171
1172        var onGetStateSuccess = function(state) {
```

```
1173            //call registration flow only if the status is
fullregistered in case of resume, so logon screen will not loose its
input values
1174            if (state.status == 'fullRegistered') {
1175                registerOrUnlock(onUnlockSuccess, onerror);
1176            }
1177        }
1178
1179        getState(onGetStateSuccess, onerror);
1180    }
1181
1182
1183    var get = function (onsuccess, onerror, key) {
1184
1185        if(!_oLogonCore) {
1186            utils.log('FlowRunner.run MAFLogon is not
initialized');
1187
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1188            return;
1189        }
1190
1191        var onUnlockSuccess = function(){
1192           _oLogonCore.getSecureStoreObject(onsuccess, onerror,
key);
1193        }
1194
1195        registerOrUnlock(onUnlockSuccess, onerror);
1196    }
1197
1198
1199
1200    var set = function (onsuccess, onerror, key, value) {
```

```
1201
1202        if(!_oLogonCore) {
1203            utils.log('FlowRunner.run MAFLogon is not
initialized');
1204
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1205            return;
1206        }
1207
1208        var onUnlockSuccess = function(){
1209           _oLogonCore.setSecureStoreObject(onsuccess, onerror,
key, value);
1210        }
1211
1212        registerOrUnlock(onUnlockSuccess, onerror);
1213    }
1214
1215
1216
1217    var lock = function (onsuccess, onerror) {
1218        if(!_oLogonCore) {
1219            utils.log('FlowRunner.run MAFLogon is not
initialized');
1220
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1221            return;
1222        }
1223
1224        _oLogonCore.lockSecureStore(onsuccess, onerror);
1225    }
1226
1227    var getState = function (onsuccess, onerror) {
1228        if(!_oLogonCore) {
```

```
1229            utils.log('FlowRunner.run MAFLogon is not
initialized');
1230
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1231            return;
1232        }
1233
1234        _oLogonCore.getState(onsuccess, onerror);
1235    }
1236
1237    var wrapCallbackWithQueueNext = function(callback) {
1238        return function() {
1239            callback.apply(this, arguments);
1240             if (flowqueue) {
1241                flowqueue.runNextFlow();
1242            }
1243        }
1244    }
1245
1246
1247    var registerOrUnlock = function(onsuccess, onerror) {
1248        if(!_oLogonCore) {
1249            utils.log('FlowRunner.run MAFLogon is not
initialized');
1250
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1251            return;
1252        }
1253
1254        var callbacks = {
1255            "onsuccess" : wrapCallbackWithQueueNext(onsuccess),
1256            "onerror" : wrapCallbackWithQueueNext(onerror)
```

```
1257        }
1258        var flowRunner = new FlowRunner(callbacks, _oLogonView,
_oLogonCore, RegistrationFlow);
1259
1260
1261        if (flowqueue) {
1262            flowqueue.add(flowRunner);
1263        }
1264        else {
1265            flowRunner.run();
1266        }
1267    }
1268
1269    var changePassword = function(onsuccess, onerror) {
1270        if(!_oLogonCore) {
1271            utils.log('FlowRunner.run MAFLogon is not
initialized');
1272
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1273            return;
1274        }
1275
1276        var onUnlockSuccess = function(){
1277            var callbacks = {
1278                "onsuccess" :
wrapCallbackWithQueueNext(onsuccess),
1279                "onerror" : wrapCallbackWithQueueNext(onerror)
1280            }
1281            var innerFlowRunner = new FlowRunner(callbacks,
_oLogonView, _oLogonCore, ChangePasswordFlow);
1282
1283            if (flowqueue) {
1284                flowqueue.add(innerFlowRunner);
```

```
1285            }
1286            else {
1287                innerFlowRunner.run();
1288            }
1289        }
1290
1291        registerOrUnlock(onUnlockSuccess, onerror);
1292    }
1293
1294
1295    var forgottenPasscode = function(onsuccess, onerror) {
1296        if(!_oLogonCore) {
1297            utils.log('FlowRunner.run MAFLogon is not
initialized');
1298
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1299            return;
1300        }
1301
1302        var onUnlockSuccess = function(){
1303        var callbacks = {
1304                "onsuccess" :
wrapCallbackWithQueueNext(onsuccess),
1305                "onerror" : wrapCallbackWithQueueNext(onerror)
1306            }
1307            var innerFlowRunner = new FlowRunner(callbacks,
_oLogonView, _oLogonCore, MockFlow);
1308            if (flowqueue) {
1309                flowqueue.add(innerFlowRunner);
1310            }
1311            else {
1312                innerFlowRunner.run();
1313            }
```

```
1314        }
1315
1316        registerOrUnlock(onUnlockSuccess, onerror);
1317    }
1318
1319    var managePasscode = function(onsuccess, onerror) {
1320        if(!_oLogonCore) {
1321            utils.log('FlowRunner.run MAFLogon is not
initialized');
1322
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1323            return;
1324        }
1325
1326        var onUnlockSuccess = function(){
1327        var callbacks = {
1328                "onsuccess" :
wrapCallbackWithQueueNext(onsuccess),
1329                "onerror" : wrapCallbackWithQueueNext(onerror)
1330            }
1331            var innerFlowRunner = new FlowRunner(callbacks,
_oLogonView, _oLogonCore, ManagePasscodeFlow);
1332            if (flowqueue) {
1333                flowqueue.add(innerFlowRunner);
1334            }
1335            else {
1336                innerFlowRunner.run();
1337            }
1338        }
1339
1340        registerOrUnlock(onUnlockSuccess, onerror);
1341    }
1342
```

```
1343    var showRegistrationData = function(onsuccess, onerror) {
1344        if(!_oLogonCore) {
1345            utils.log('FlowRunner.run MAFLogon is not
initialized');
1346
onerror(errorWithDomainCodeDescription("MAFLogon","2","MAFLogon is
not initialized"));
1347            return;
1348        }
1349
1350        var onUnlockSuccess = function(){
1351        var callbacks = {
1352                "onsuccess" :
wrapCallbackWithQueueNext(onsuccess),
1353                "onerror" : wrapCallbackWithQueueNext(onerror)
1354            }
1355            var innerFlowRunner = new FlowRunner(callbacks,
_oLogonView, _oLogonCore, ShowRegistrationFlow);
1356            if (flowqueue) {
1357                flowqueue.add(innerFlowRunner);
1358            }
1359            else {
1360                innerFlowRunner.run();
1361            }
1362        }
1363
1364        registerOrUnlock(onUnlockSuccess, onerror);
1365    }
1366
1367    var getSecureStoreErrorText = function(error) {
1368      utils.logJSON('LogonController.getSecureStoreErrorText: '
+ JSON.stringify(error));
1369
1370        var errorText;
```

```
1371
1372        if(error.errorCode === '14' && error.errorDomain ===
'MAFSecureStoreManagerErrorDomain')
1373            errorText = "ERR_PASSCODE_TOO_SHORT";
1374       else if(error.errorCode === '10' && error.errorDomain ===
'MAFSecureStoreManagerErrorDomain')
1375            errorText = "ERR_PASSCODE_REQUIRES_DIGIT";
1376       else if(error.errorCode === '13' && error.errorDomain ===
'MAFSecureStoreManagerErrorDomain')
1377            errorText = "ERR_PASSCODE_REQUIRES_UPPER";
1378       else if(error.errorCode === '11' && error.errorDomain ===
'MAFSecureStoreManagerErrorDomain')
1379            errorText = "ERR_PASSCODE_REQUIRES_LOWER";
1380       else if(error.errorCode === '12' && error.errorDomain ===
'MAFSecureStoreManagerErrorDomain')
1381            errorText = "ERR_PASSCODE_REQUIRES_SPECIAL";
1382       else if(error.errorCode === '15' && error.errorDomain ===
'MAFSecureStoreManagerErrorDomain')
1383            errorText = "ERR_PASSCODE_UNDER_MIN_UNIQUE_CHARS";
1384        else {
1385            errorText = "ERR_SETPASSCODE_FAILED";
1386        }
1387
1388        return errorText;
1389    }
1390
1391    var getSSOPasscodeSetErrorText = function(error) {
1392
utils.logJSON('LogonController.getSSOPasscodeSetErrorText: ' +
JSON.stringify(error));
1393
1394        var errorText;
1395
1396        if (error.errorDomain === 'MAFLogonCoreErrorDomain') {
1397            if (error.errorCode === '16') {
```

```
1398                errorText = "ERR_SSO_PASSCODE_SET_ERROR";
1399            }
1400        }
1401
1402        return errorText;
1403    }
1404
1405    var getRegistrationErrorText = function(error) {
1406        utils.logJSON('LogonController.getRegistrationErrorText:
' + JSON.stringify(error));
1407
1408        var errorText;
1409
1410        if (error.errorDomain === 'MAFLogonCoreErrorDomain') {
1411            if (error.errorCode === '80003') {
1412                errorText = "ERR_REG_FAILED_WRONG_SERVER";
1413            }
1414            //in case of wrong application id
1415            else if (error.errorCode === '404') {
1416                errorText = "ERR_REG_FAILED";
1417            }
1418            else if (error.errorCode === '401') {
1419                errorText = "ERR_REG_FAILED_UNATHORIZED";
1420            }
1421            else {
1422                errorText = "ERR_REG_FAILED";
1423            }
1424        }
1425
1426        return errorText;
1427    }
1428
```

```
1429    var getRegistrationCancelError = function(error) {
1430
utils.logJSON('LogonController.getRegistrationCancelError: ' +
JSON.stringify(error));
1431
1432        var errorText;
1433
1434        errorText = "ERR_REGISTRATION_CANCEL";
1435
1436        return errorText;
1437    }
1438
1439    var errorWithDomainCodeDescription = function(domain, code,
description) {
1440        var error = {
1441        errorDomain: domain,
1442        errorCode: code,
1443        errorMessage: description
1444        };
1445
1446        return error;
1447    }
1448
1449
1450    // ==================== exported (public) members
====================
1451
1452    /**
1453     * The Logon plugin provides screen flows to register an app
with an SAP Mobile Platform server.<br/>
1454     * <br/>
1455     * The logon plugin is a component of the SAP Mobile
Application Framework (MAF), exposed as a Cordova plugin. The basic
```

```
1456     * 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

1457     * stores those values in its own secure data vault. This data
vault is separate from the one provided with the

1458     * 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

1459     * server to receive an application connection ID for a
particular app. The application connection ID must be sent

1460     * along with each request that is proxied through the SAP
Mobile Platform 3.0 server to the OData producer.<br/>

1461     * <br/>

1462     * <b>Adding and Removing the Logon Plugin</b><br/>

1463     * The Logon plugin is added and removed using the

1464     * <a href="http://cordova.apache.org/docs/en/edge/
guide_cli_index.md.html#The%20Command-line%20Interface">Cordova
CLI</a>.<br/>

1465     * <br/>

1466     * To add the Logon plugin to your project, use the following
command:<br/>

1467     * cordova plugin add <full path to directory containing
Kapsel plugins>\logon<br/>

1468     * <br/>

1469     * To remove the Logon plugin from your project, use the
following command:<br/>

1470     * cordova plugin rm com.sap.mp.cordova.plugins.logon

1471     *

1472     * @namespace

1473     * @alias Logon

1474     * @memberof sap

1475     */

1476    module.exports = {

1477

1478        /**

1479       * Initialization method to set up the Logon plugin.  This
will register the application with the SMP server and also
authenticate the user
```

```
1480         * with servers on the network.  This step must be done
first prior to any attempt to communicate with the SMP server.
1481         *
1482         * @method
1483        * @param {sap.Logon~successCallback} successCallback The
function that is invoked if initialization is successful.  The
current
1484         * context is passed to this function as the parameter.
1485         * @param {sap.Logon~errorCallback} errorCallback The
function that is invoked in case of an error.
1486         * @param {string} applicationId The unique ID of the
application.  Must match the application ID on the SAP Mobile
Platform server.
1487         * @param {object} [context] The context with default
values for application registration.  See {@link
sap.Logon~successCallback} for the structure
1488        * of the context object.  Note that all properties of the
context object are optional, and you only need to specify the
properties
1489         * 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
1490         * a chance to override these values during runtime.
1491         * @param {string} [logonView="com/sap/mp/logon/iabui"]
The cordova module ID of a custom renderer for the logon,
1492         * implementing the [showScreen(), close()] interface.
Please use the defaul module unless you are absolutely sure that you
can provide your own
1493         * custom implementation.  Please refer to JavaScript
files inside your Kapsel project's plugins\logon\www\common\modules\
folder as example.
1494         * @example
1495         * // a custom UI can be loaded here
1496         * var logonView = sap.logon.IabUi;
1497         *
1498         * // The app ID
1499         * var applicationId = "someAppID";
1500         *
```

```
1501         * // You only need to specify the fields for which you
want to set the default.   These values are optional because they
will be
1502         * // used to prefill the fields on Logon's UI screen.
1503         * var defaultContext = {
1504         *  "serverHost" : "defaultServerHost.com"
1505         *    "https" : false,
1506         *    "serverPort" : "8080",
1507         *    "user" : "user1",
1508         *    "password" : "Zzzzzz123",
1509         *    "communicatorId" : "REST",
1510         *    "securityConfig" : "sec1",
1511         *    "passcode" : "Aaaaaa123",
1512         *    "unlockPasscode" : "Aaaaaa123"
1513         * };
1514         *
1515         * var app_context;
1516         *
1517         * var successCallback = function(context){
1518         *     app_context = context;
1519         * }
1520         *
1521         * var errorCallback = function(errorInfo){
1522         *     alert("error: " + JSON.stringify(errorInfo));
1523         * }
1524         * sap.Logon.init(successCallback, errorCallback,
applicationId, defaultContext, logonView);
1525         */
1526    init: init,
1527
1528        /**
1529         * The application ID with which {@link sap.Logon.init}
was called.  It is available here so it is easy to access later.
1530         * @example
```

```
1531         * // After calling the init function
1532         * alert("The app ID for this app is: " +
sap.Logon.applicationId);
1533         */
1534    applicationId: null,
1535        /**
1536         * Direct reference to the logon core object used by the
Logon plugin.  This is needed to perform more complex operations
that
1537         * are not generally needed by applications. <br/>
1538        * There are several functions that can be accessed on the
core object:<br/>
1539         *         
getState(successCallback,errorCallback) returns the state object of
the application to the success callback in the form of a JavaScript
object.<br/>
1540         *         
getContext(successCallback,errorCallback) returns the context object
of the application to the success callback in the form of a
JavaScript object.<br/>
1541         *         
deleteRegistration(successCallback,errorCallback) deletes the
application's registration from the SAP Mobile Platform server and
removes<br/>
1542         *
           &
nbsp;    application data on device.<br/>
1543         * @example
1544         * var successCallback = function(result){
1545         *     alert("Result: " + JSON.stringify(result));
1546         * }
1547         * var errorCallback = function(errorInfo){
1548         *     alert("Error: " + JSON.stringify(errorInfo));
1549         * }
1550         *
sap.Logon.core.getState(successCallback,errorCallback);
1551         *
sap.Logon.core.getContext(successCallback,errorCallback);
1552         *
sap.Logon.core.deleteRegistration(successCallback,errorCallback);
```

```
1553         */
1554    core: _oLogonCore, //Must be updated after init
1555
1556        /**
1557         * Get an  (JSON serializable) object from the DataVault
for a given key.
1558         * @method
1559         * @param {sap.Logon~getSuccessCallback} onsuccess The
function that is invoked
1560        * upon success.  It is called with the resulting object as
a single parameter.
1561         * This can be null or undefined, if no object is defined
for the given key.
1562         * @param {sap.Logon~errorCallback} onerror The function
to invoke in case of error.
1563         * @param {string} key The key with which to query the
DataVault.
1564         * @example
1565         * var errorCallback = function(errorInfo){
1566         *     alert("Error: " + JSON.stringify(errorInfo));
1567         * }
1568         * var getSuccess = function(value){
1569         *     alert("value retrieved from the store: " +
JSON.stringify(value));
1570         * }
1571         * var setSuccess = function(){
1572         *
sap.Logon.get(getSuccess,errorCallback,'someKey');
1573         * }
1574         * sap.Logon.set(setSuccess,errorCallback,'someKey',
'some string (could also be an object).');
1575         */
1576    get: get,
1577
1578        /**
1579         * Set an (JSON serializable) object in the DataVault.
```

```
1580         * @method
1581         * @param {sap.Logon~successCallbackNoParameters}
onsuccess The function to invoke upon success.
1582         * onsuccess will be called without parameters for this
method.
1583        * @param {sap.Logon~errorCallback} onerror The function
to invoke in case of error.
1584         * @param {string} key The key to store the provided
object on.
1585        * @param {object} value The object to be set on the given
key.  Must be JSON serializable (ie:
1586         * cannot contain circular references).
1587         * @example
1588         * var errorCallback = function(errorInfo){
1589         *     alert("Error: " + JSON.stringify(errorInfo));
1590         * }
1591         * var getSuccess = function(value){
1592         *     alert("value retrieved from the store: " +
JSON.stringify(value));
1593         * }
1594         * var setSuccess = function(){
1595         *
sap.Logon.get(getSuccess,errorCallback,'someKey');
1596         * }
1597         * sap.Logon.set(setSuccess,errorCallback,'someKey',
'some string (could also be an object).');
1598         */
1599    set: set,
1600
1601        /**
1602         * Locks the Logon plugin's secure data vault.
1603         * @method
1604         * @param {sap.Logon~successCallbackNoParameters}
onsuccess The function to invoke upon success.
1605        * @param {sap.Logon~errorCallback} onerror The function
to invoke in case of error.
```

```
1606          * @example
1607          * var errorCallback = function(errorInfo){
1608          *     alert("Error: " + JSON.stringify(errorInfo));
1609          * }
1610          * var successCallback = function(){
1611          *     alert("Locked!");
1612          * }
1613          * sap.Logon.lock(successCallback,errorCallback);
1614          */
1615    lock: lock,
1616
1617        /**
1618         * Unlock the Logon plugin's secure data vault if it has
been locked (due to being inactive, or
1619        * {@link sap.Logon.lock} being called), then the user is
prompted for the passcode to unlock the
1620          * application.<br/>
1621          * If the application is already unlocked, then nothing
will be done.<br/>
1622        * If the application has passcode disabled, then passcode
prompt will not be necessary.
1623          * In all cases if an error does not occur, the success
callback is invoked with the current logon context
1624          * as the parameter.
1625          * @method
1626          * @param {sap.Logon~successCallback} onsuccess - The
callback to call if the screen flow succeeds.
1627        * onsuccess will be called with the current logon context
as a single parameter.
1628        * @param {sap.Logon~errorCallback} onerror - The callback
to call if the screen flow fails.
1629          * @example
1630          * var errorCallback = function(errorInfo){
1631          *     alert("Error: " + JSON.stringify(errorInfo));
1632          * }
```

```
1633         * var successCallback = function(context){
1634         *     alert("Registered and unlocked.  Context: " +
JSON.stringify(context));
1635         * }
1636         * sap.Logon.unlock(successCallback,errorCallback);
1637         */
1638    unlock: registerOrUnlock,
1639
1640        /**
1641         * This is an alias for registerOrUnlock.  Calling this
function is equivalent
1642        * to calling {@link sap.Logon.unlock} since both of them
are alias to registerOrUnlock.
1643         * @method
1644         * @private
1645         */
1646    registerUser: registerOrUnlock,
1647
1648        /**
1649         * This function registers the user and creates a new
unlocked DataVault to store the registration
1650         * information.<br/>
1651         * If the user has already been registered, but the
application is locked (due to being inactive, or
1652        * {@link sap.Logon.lock} being called), then the user is
prompted for the passcode to unlock the
1653         * application.<br/>
1654         * If the application is already unlocked, then nothing
will be done.<br/>
1655         * In all cases if an error does not occur, the success
callback is invoked with the current logon context
1656         * as the parameter.
1657         * @method
1658         * @param {sap.Logon~successCallback} onsuccess - The
callback to call if the screen flow succeeds.
```

```
1659         * onsuccess will be called with the current logon context
as a single parameter.
1660         * @param {sap.Logon~errorCallback} onerror - The callback
to call if the screen flow fails.
1661          * @example
1662          * var errorCallback = function(errorInfo){
1663          *     alert("Error: " + JSON.stringify(errorInfo));
1664          * }
1665          * var successCallback = function(context){
1666          *     alert("Registered and unlocked.  Context: " +
JSON.stringify(context));
1667          * }
1668          *
sap.Logon.registerOrUnlock(successCallback,errorCallback);
1669          * @private
1670          */
1671     registerOrUnlock: registerOrUnlock,
1672
1673         /**
1674          * This method will launch the UI screen for application
users to manage and update the data vault passcode or,
1675          * if the SMP server's Client Passcode Policy allows it,
enable or disable the passcode to the data vault.
1676          *
1677          * @method
1678          * @param {sap.Logon~successCallbackNoParameters}
onsuccess - The function to invoke upon success.
1679         * @param {sap.Logon~errorCallback} onerror - The function
to invoke in case of error.
1680          * @example
1681          * var errorCallback = function(errorInfo){
1682          *     alert("Error: " + JSON.stringify(errorInfo));
1683          * }
1684          * var successCallback = function(context){
1685          *     alert("Passcode successfully managed.");
```

```
1686         * }
1687         *
sap.Logon.managePasscode(successCallback,errorCallback);
1688         */
1689    managePasscode: managePasscode,
1690
1691        /**
1692         * This method will launch the UI screen for application
users to manage and update the back-end passcode that Logon stores in
the
1693         * data vault that is used to authenticate the client to
the server.
1694         *
1695         * @method
1696         * @param {sap.Logon~successCallbackNoParameters}
onsuccess - The callback to call if the screen flow succeeds.
1697         * onsuccess will be called without parameters for this
method.
1698        * @param {sap.Logon~errorCallback} onerror The function
that is invoked in case of an error.
1699         * @example
1700         * var errorCallback = function(errorInfo){
1701         *     alert("Error: " + JSON.stringify(errorInfo));
1702         * }
1703         * var successCallback = function(context){
1704         *     alert("Password successfully changed.");
1705         * }
1706         *
sap.Logon.changePassword(successCallback,errorCallback);
1707         */
1708    changePassword: changePassword,
1709
1710        /**
1711         * Calling this method will show a screen which displays
the current registration settings for the application.
1712         * @method
```

```
1713         * @param {sap.Logon~successCallbackNoParameters}
onsuccess - The callback to call if the screen flow succeeds.
1714         * onsuccess will be called without parameters for this
method.
1715        * @param {sap.Logon~errorCallback} onerror The function
that is invoked in case of an error.
1716         * @example
1717         * var errorCallback = function(errorInfo){
1718         *     alert("Error: " + JSON.stringify(errorInfo));
1719         * }
1720         * var successCallback = function(context){
1721         *     alert("The showRegistrationData screenflow was
successful.");
1722         * }
1723         *
sap.Logon.showRegistrationData(successCallback,errorCallback);
1724         */
1725    showRegistrationData: showRegistrationData
1726    };
1727
1728    /**
1729     * Callback function that is invoked in case of an error.
1730     *
1731     * @callback sap.Logon~errorCallback
1732     *
1733    * @param {Object} errorObject Depending on the origin of the
error the object can take several forms.
1734     * (Unfortunately the error object structure and content is
not uniform among the platforms, this will
1735     * probably change in the future.)
1736     *
1737     * Errors originating from the logon plugin have only an
'errorKey' property.
1738     * The possible values for 'errorKey':
1739     *
```

```
1740     * ERR_CHANGE_TIMEOUT_FAILED
1741     * ERR_FORGOT_SSO_PIN
1742     * ERR_INIT_FAILED
1743     * ERR_INVALID_ACTION
1744     * ERR_INVALID_STATE
1745     * ERR_PASSCODE_REQUIRES_DIGIT
1746     * ERR_PASSCODE_REQUIRES_LOWER
1747     * ERR_PASSCODE_REQUIRES_SPECIAL
1748     * ERR_PASSCODE_REQUIRES_UPPER
1749     * ERR_PASSCODE_TOO_SHORT
1750     * ERR_PASSCODE_UNDER_MIN_UNIQUE_CHARS
1751     * ERR_REGISTRATION_CANCEL
1752     * ERR_REG_FAILED
1753     * ERR_REG_FAILED_UNATHORIZED
1754     * ERR_REG_FAILED_WRONG_SERVER
1755     * ERR_SETPASSCODE_FAILED
1756     * ERR_SET_AFARIA_CREDENTIAL_FAILED
1757     * ERR_SSO_PASSCODE_SET_ERROR
1758     * ERR_UNKNOWN_SCREEN_ID
1759     * ERR_UNLOCK_FAILED
1760     * ERR_USER_CANCELLED
1761     *
1762     * Errors originating in the logon core (either iOS or
Android) have the following properties: 'errorCode',
1763     * 'errorMessage', and 'errorDomain'.
1764     * The 'errorCode' is just a number uniquely identifying the
error.  The 'errorMessage'
1765    * property is a string with more detailed information of what
went wrong.  The 'errorDomain' property specifies
1766     * the domain that the error occurred in.
1767     *
1768     * On iOS the 'errorDomain' property of the core errors can
take the following values: MAFLogonCoreErrorDomain,
```

```
MAFSecureStoreManagerErrorDomain, and
MAFLogonCoreCDVPluginErrorDomain.

1769     *

1770     * In the MAFLogonCoreErrorDomain the following errors are
thrown (throwing methods in paren):

1771     *

1772     *  3   errMAFLogonErrorCommunicationManagerError
(register, update settings, delete, change backend password)

1773     *  9   errMAFLogonErrorCouldNotDecideCommunicator
(register)

1774     *  11  errMAFLogonErrorOperationNotAllowed
(all)

1775     *  12  errMAFLogonErrorInvalidServerHost
(register)

1776     *  13  errMAFLogonErrorInvalidBackendPassword
(changeBackendPassword)

1777     *  15  errMAFLogonErrorUploadTraceFailed
(uploadTrace)

1778     *  16  errMAFLogonErrorInvalidMCIMSSOPin
(setMCIMSSOPin)

1779     *  18  errMAFLogonErrorCertificateKeyError
(register)

1780     *  19  errMAFLogonErrorCertificateError
(register)

1781     *  20  errMAFLogonErrorAfariaInvalidCredentials
(setAfariaCredentialWithUser)

1782     *

1783     * In the MAFSecureStoreManagerErrorDomain the following
errors are thrown (throwing methods in paren):

1784     *

1785     *  0   errMAFSecureStoreManagerErrorUnknown    (persist,
unlock, changePasscode, delete, getContext)

1786     *  1   errMAFSecureStoreManagerErrorAlreadyExists
(persist)

1787     *  2   errMAFSecureStoreManagerErrorDataTypeError  (unlock,
getContext)

1788     *  3   errMAFSecureStoreManagerErrorDoesNotExist   (unlock,
persist, getContext)

1789     *  4   errMAFSecureStoreManagerErrorInvalidArg unlock,
(persist, getContext)
```

```
1790     *  5   errMAFSecureStoreManagerErrorInvalidPassword
(unlock)

1791     *  6   errMAFSecureStoreManagerErrorLocked
(getContext)

1792     *  7   errMAFSecureStoreManagerErrorOutOfMemory    (persist,
unlock, changePasscode, delete, getContext)

1793     *  8   errMAFSecureStoreManagerErrorPasswordExpired
(unlock, getContext)

1794     *  9   errMAFSecureStoreManagerErrorPasswordRequired
(persist, changePasscode)

1795     *  10  errMAFSecureStoreManagerErrorPasswordRequiresDigit
(persist, changePasscode)

1796     *  11  errMAFSecureStoreManagerErrorPasswordRequiresLower
(persist, changePasscode)

1797     *  12
errMAFSecureStoreManagerErrorPasswordRequiresSpecial    (persist,
changePasscode)

1798     *  13  errMAFSecureStoreManagerErrorPasswordRequiresUpper
(persist, changePasscode)

1799     *  14  errMAFSecureStoreManagerErrorPasswordUnderMinLength
(persist, changePasscode)

1800     *  15
errMAFSecureStoreManagerErrorPasswordUnderMinUniqueChars
(persist, changePasscode)

1801     *  16  errMAFSecureStoreManagerErrorDeleted    (unlock)

1802     *

1803     * In the MAFLogonCoreCDVPluginErrorDomain the following
errors are thrown:

1804     *

1805     *  1 (init failed)

1806     *  2 (plugin not initialized)

1807     *  3 (no input provided)

1808     *

1809     * On Android the 'errorDomain' property of the core errors
can take the following values: MAFLogonCoreErrorDomain and
MAFLogonCoreCDVPluginErrorDomain.

1810     * There are no logon specific error codes, the 'errorCode'
property only wraps the error values from the underlying libraries.

1811     */
```

```
1812
1813    /**
1814     * Callback function that is invoked upon successfully
registering or unlocking or retrieving the context.
1815     *
1816     * @callback sap.Logon~successCallback
1817     *
1818     * @param {Object} context An object containing the current
logon context.  Two properties of particular importance
1819     * are applicationEndpointURL, and applicationConnectionId.
1820     * The context object contains the following properties:<br/
>
1821     * "registrationContext": {<br/>
1822     *
        "serverHost": Host
of the server.<br/>
1823     *         "domain":
Domain for server. Can be used in case of SAP Mobile Platform
communication.<br/>
1824     *
        "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/
>
1825     *         "https":
Marks whether the server should be accessed in a secure way.<br/>
1826     *
        "serverPort": Port
of the server.<br/>
1827     *         "user":
Username in the backend.<br/>
1828     *
        "password":
Password for the backend user.<br/>
1829     *         "farmId":
FarmId of the server. Can be nil. Used in case of Relay server or
SiteMinder.<br/>
1830     *
        "communicatorId":
Id of the communicator manager that will be used for performing the
logon. Possible values: IMO / GATEWAY / REST<br/>
```

```
1831     *
        "securityConfig":
Security configuration. If nil, the default configuration is
used.<br/>

1832     *
        "mobileUser":
Mobile User. Used in case of IMO manual user creation.<br/>

1833     *
        "activationCode":
Activation Code. Used in case of IMO manual user creation.<br/>

1834     *
        "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/>

1835     *
        "gatewayPingPath":
The custom path of the ping URL on the gateway. Used in case of
Gateway only registration mode.<br/>

1836     * }<br/>

1837     * "applicationEndpointURL": Contains the application
endpoint URL after a successful registration.<br/>

1838     * "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/>

1839    * "afariaRegistration": manual / automatic / certificate<br/
>

1840     * "policyContext": Contains the password policy for the
secure store {<br/>

1841     *
        "alwaysOn":<br/>

1842     *
        "alwaysOff":<br/>

1843     *
        "defaultOn":<br/>

1844     *
        "hasDigits":<br/>

1845     *
        "hasLowerCaseLetter
s":<br/>

1846     *
        "hasSpecialLetters"
:<br/>
```

```
1847     *
        "hasUpperCaseLetter
s":<br/>

1848     *
        "defaultAllowed":<b
r/>

1849     *
        "expirationDays":<b
r/>

1850     *
        "lockTimeout":<br/>

1851     *
        "minLength":<br/>

1852     *
        "minUniqueChars":<b
r/>

1853     *
        "retryLimit":<br/>

1854     * }<br/>

1855     * "registrationReadOnly": specifies whether context values
are coming from clientHub / afaria<br/>

1856     * "policyReadOnly": specifies whether passcode policy is
coming from afaria<br/>

1857     * "credentialsByClientHub": specifies whether credentials
are coming from clientHub

1858     */

1859

1860    /**

1861     * Callback function that will be invoked with no
parameters.

1862     *

1863     * @callback sap.Logon~successCallbackNoParameters

1864     */

1865

1866    /**

1867     * Callback function that is invoked upon successfully
retrieving an object from the DataVault.

1868     *

1869     * @callback sap.Logon~getSuccessCallback
```

```
1870     *
1871     * @param {Object} value The object that was stored with the
given key.  Can be null or undefined if no object was stored
1872     * with the given key.
1873     */
1874
1875
```

Using the AuthProxy Plugin

The 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.

Domain Whitelisting
Kapsel plugins support Apache Cordova's domain whitelisting model. Whitelisting allows you to control access to external network resources. Apache Cordova whitelisting allows you to whitelist individual network resources (URLs), for example, http://www.google.com.

For information about the whitelist rules, see *http://docs.phonegap.com/en/3.3.0/guide_appdev_whitelist_index.md.html*.

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 128	Create certificate source description object for a certificate from a keystore file.
sap.AuthProxy.CertificateFromLogonManager on page 129	Create a certificate source description object for certificates from logon manager.
sap.AuthProxy.CertificateFromStore on page 130	Create a certificate source description object for certificates from the system keystore.

Members

Name	Description
ERR_CERTIFICATE_ALIAS_NOT_FOUND on page 131	Constant indicating the certificate with the given alias could not be found.
ERR_CERTIFICATE_FILE_NOT_EXIST on page 131	Constant indicating the certificate file could not be found.
ERR_CERTIFICATE_INVALID_FILE_FORMAT on page 131	Constant indicating incorrect certificate file format.
ERR_CLIENT_CERTIFICATE_VALIDATION on page 131	Constant indicating the provided certificate failed validation on the server side.
ERR_DOMAIN_WHITELIST_REJECTION on page 132	Constant indicating cordova domain whitelist rejection error while sending request to server.
ERR_FILE_CERTIFICATE_SOURCE_UNSUPPORTED on page 132	Constant indicating the certificate from file is not supported on the current platform.
ERR_GET_CERTIFICATE_FAILED on page 132	Constant indicating failure in getting the certificate.
ERR_HTTP_TIMEOUT on page 132	Constant indicating timeout error while connecting to the server.
ERR_INVALID_PARAMETER_VALUE on page 133	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 133	Constant indicating the logon manager certifciate method is not available.
ERR_LOGON_MANAGER_CORE_NOT_AVAILABLE on page 133	Constant indicating the logon manager core library is not available.
ERR_MISSING_PARAMETER on page 133	Constant indicating the operation failed because of a missing parameter.

ERR_NO_SUCH_ACTION on page 134	Constant indicating there is no such Cordova action for the current service.
ERR_SERVER_CERTIFICATE_ VALIDATION on page 134	Constant indicating the server certificate failed validation on the client side.
ERR_SERVER_REQUEST_FAILED on page 134	Constant indicating the server request failed.
ERR_SYSTEM_CERTIFICATE_SOURCE_UNSUPPORTED on page 135	Constant indicating the certificate from the system keystore is not supported on the current platform.
ERR_UNKNOWN on page 135	Constant indicating the operation failed with unknown error.

Methods

Name	Description
deleteCertificateFromStore(successCB, [errorCB], certificateKey) on page 135	Delete a cached certificate from the keychain.
generateODataHttpClient() on page 136	Generates an OData client that uses the AuthProxy plugin to make requests.
get(url, header, successCB, errorCB, [user], [password], [timeout], [certSource]) on page 137	Send an HTTP(S) GET request to a remote server.
sendRequest(method, url, header, requestBody, successCB, errorCB, [user], [password], [timeout], [certSource]) on page 139	Send an HTTP(S) request to a remote server.

Type Definitions

Name	Description
deleteCertificateSuccessCallback on page 141	Callback function that is invoked upon successfully deleting a certificate from the store.
errorCallback(errorObject) on page 141	Callback function that is invoked in case of an error.
successCallback(serverResponse) on page 142	Callback function that is invoked upon a response from the server.

Source

authproxy.js, line 27 on page 144.

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 225 on page 152.

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 282 on page 154.

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 254 on page 153.

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 141.

Syntax
<constant> `ERR_CERTIFICATE_ALIAS_NOT_FOUND` : number

Source
authproxy.js, line 90 on page 147.

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 141.

Syntax
<constant> `ERR_CERTIFICATE_FILE_NOT_EXIST` : number

Source
authproxy.js, line 98 on page 147.

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 141.

Syntax
<constant> `ERR_CERTIFICATE_INVALID_FILE_FORMAT` : number

Source
authproxy.js, line 106 on page 147.

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 141.

Syntax
<constant> `ERR_CLIENT_CERTIFICATE_VALIDATION` : number

Source
authproxy.js, line 122 on page 148.

ERR_DOMAIN_WHITELIST_REJECTION member
Constant indicating cordova domain whitelist rejection error while sending request to server.

Used as a possible value for the errorCode in *sap.AuthProxy~errorCallback* on page 141.

Syntax
<constant> `ERR_DOMAIN_WHITELIST_REJECTION` : number

Source
authproxy.js, line 172 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 141.

Syntax
<constant> `ERR_FILE_CERTIFICATE_SOURCE_UNSUPPORTED` : number

Source
authproxy.js, line 74 on page 146.

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 141.

Syntax
<constant> `ERR_GET_CERTIFICATE_FAILED` : number

Source
authproxy.js, line 114 on page 148.

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 141.

Syntax
<constant> `ERR_HTTP_TIMEOUT` : number

Source
authproxy.js, line 164 on page 149.

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 141.

Syntax
<constant> `ERR_INVALID_PARAMETER_VALUE` : number

Source
authproxy.js, line 48 on page 145.

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 141.

Syntax
<constant> `ERR_LOGON_MANAGER_CERTIFICATE_METHOD_NOT_AVAILABLE` : number

Source
authproxy.js, line 156 on page 149.

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 141.

Syntax
<constant> `ERR_LOGON_MANAGER_CORE_NOT_AVAILABLE` : number

Source
authproxy.js, line 148 on page 149.

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 141.

Syntax
<constant> `ERR_MISSING_PARAMETER` : number

Source
authproxy.js, line 56 on page 145.

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 141.

Syntax
<constant> `ERR_NO_SUCH_ACTION` : number

Source
authproxy.js, line 66 on page 146.

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 141.

Syntax
<constant> `ERR_SERVER_CERTIFICATE_VALIDATION` : number

Source
authproxy.js, line 131 on page 148.

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 141.

Syntax
<constant> `ERR_SERVER_REQUEST_FAILED` : number

Source
authproxy.js, line 139 on page 148.

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 141.

Syntax
<constant> `ERR_SYSTEM_CERTIFICATE_SOURCE_UNSUPPORTED` : number

Source
authproxy.js, line 82 on page 146.

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 141.

Syntax
<constant> `ERR_UNKNOWN` : number

Source
authproxy.js, line 40 on page 145.

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 141		Callback method upon success.
errorCB	*sap.AuthProxy~errorCallback* on page 141	(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 637 on page 168.

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 734 on page 172.

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 139 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 142		Callback method invoked upon a response from the server.
errorCB	*sap.AuthProxy~errorCallback* on page 141		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 timeout is 60 seconds. A value of 0 means there is no timeout.
certSource	Object	(optional)	Certificate description object.It can be one of *sap.AuthProxy#CertificateFromFile* on page 128, *sap.AuthProxy#CertificateFromStore* on page 130, or *sap.AuthProxy#CertificateFromLogonManager* on page 129.

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 617 on page 167.

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]@host-name[: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 142		Callback method invoked upon a response from the server.
errorCB	*sap.AuthProxy~errorCallback* on page 141		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 timeout is 60 seconds. A value of 0 means there is no timeout.
certSource	Object	(optional)	Certificate description object.It can be one of *sap.AuthProxy#CertificateFromFile* on page 128, *sap.AuthProxy#CertificateFromStore* on page 130, or *sap.AuthProxy#CertificateFromLogonManager* on page 129.

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 467 on page 162.

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

Syntax
`deleteCertificateSuccessCallback`()

Source
authproxy.js, line 820 on page 175.

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 125 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 816 on page 174.

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 818 on page 174.

Source code

authproxy.js

```
1       // 3.0.2-SNAPSHOT
2       var exec = require('cordova/exec');
3
```

```
4       /**
5        * The AuthProxy plugin provides the ability to make HTTPS
requests with mutual authentication.<br/>
6        * <br/>
7        * The regular XMLHttpRequest does not
8       * support mutual authentication.  The AuthProxy plugin allows
you to specify a certificate to include in an HTTPS request
9        * to identify the client to the server.  This allows the
server to verify the identity of the client.  An example of where
you
10       * might need mutual authenticaion is the onboarding process
to register with an application, or, to access an
11      * OData producer. This occurs mostly in Business to Business
(B2B) applications. This is different from most business to
12       * consumer (B2C) web sites where it is only the server that
authenticates itself to the client with a certificate.<br/>
13       * <br/>
14       * <b>Adding and Removing the AuthProxy Plugin</b><br/>
15       * The AuthProxy plugin is added and removed using the
16       * <a href="http://cordova.apache.org/docs/en/edge/
guide_cli_index.md.html#The%20Command-line%20Interface">Cordova
CLI</a>.<br/>
17       * <br/>
18       * To add the AuthProxy plugin to your project, use the
following command:<br/>
19       * cordova plugin add <path to directory containing Kapsel
plugins>\authproxy<br/>
20       * <br/>
21      * To remove the AuthProxy plugin from your project, use the
following command:<br/>
22       * cordova plugin rm com.sap.mp.cordova.plugins.authproxy
23       * @namespace
24       * @alias AuthProxy
25       * @memberof sap
26       */
27      var AuthProxy = function () {};
28
```

```
29
30      /**
31       * Constant definitions for registration methods
32       */
33
34      /**
35       * Constant indicating the operation failed with unknown
error. Used as a possible value for the
36       * errorCode in {@link sap.AuthProxy~errorCallback}.
37       * @constant
38       * @type number
39       */
40      AuthProxy.prototype.ERR_UNKNOWN = -1;
41
42      /**
43       * Constant indicating the operation failed due to an invalid
parameter (for example, a string was passed where a number was
44       * required). Used as a possible value for the errorCode in
{@link sap.AuthProxy~errorCallback}.
45       * @constant
46       * @type number
47       */
48      AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE = -2;
49
50      /**
51       * Constant indicating the operation failed because of a
missing parameter. Used as a possible value for the
52       * errorCode in {@link sap.AuthProxy~errorCallback}.
53       * @constant
54       * @type number
55       */
56      AuthProxy.prototype.ERR_MISSING_PARAMETER = -3;
57
58      /**
```

```
59      * Constant indicating there is no such Cordova action for the
current service.  When a Cordova plugin calls into native

60       * code it specifies an action to perform.  If the action
provided by the JavaScript is unknown to the native code this

61       * error occurs.  This error should not occur as long as
authproxy.js is unmodified. Used as a possible

62       * value for the errorCode in {@link
sap.AuthProxy~errorCallback}.

63       * @constant

64       * @type number

65       */

66      AuthProxy.prototype.ERR_NO_SUCH_ACTION = -100;

67

68      /**

69       * Constant indicating the certificate from file is not
supported on the current platform. Used as a possible value for the

70       * errorCode in {@link sap.AuthProxy~errorCallback}.

71       * @constant

72       * @type number

73       */

74      AuthProxy.prototype.ERR_FILE_CERTIFICATE_SOURCE_UNSUPPORTED
= -101;

75

76      /**

77       * Constant indicating the certificate from the system
keystore is not supported on the current platform. Used as a possible
value

78       * for the errorCode in {@link
sap.AuthProxy~errorCallback}.

79       * @constant

80       * @type number

81       */

82
AuthProxy.prototype.ERR_SYSTEM_CERTIFICATE_SOURCE_UNSUPPORTED =
-102;

83

84      /**
```

```
85       * Constant indicating the certificate with the given alias
could not be found. Used as a possible value for the
86       * errorCode in {@link sap.AuthProxy~errorCallback}.
87       * @constant
88       * @type number
89       */
90      AuthProxy.prototype.ERR_CERTIFICATE_ALIAS_NOT_FOUND = -104;
91
92      /**
93       * Constant indicating the certificate file could not be
found. Used as a possible value for the
94       * errorCode in {@link sap.AuthProxy~errorCallback}.
95       * @constant
96       * @type number
97       */
98      AuthProxy.prototype.ERR_CERTIFICATE_FILE_NOT_EXIST = -105;
99
100     /**
101      * Constant indicating incorrect certificate file format.
Used as a possible value for the
102      * errorCode in {@link sap.AuthProxy~errorCallback}.
103      * @constant
104      * @type number
105      */
106     AuthProxy.prototype.ERR_CERTIFICATE_INVALID_FILE_FORMAT =
-106;
107
108     /**
109      * Constant indicating failure in getting the certificate.
Used as a possible value for the
110      * errorCode in {@link sap.AuthProxy~errorCallback}.
111      * @constant
112      * @type number
113      */
```

```
114     AuthProxy.prototype.ERR_GET_CERTIFICATE_FAILED = -107;
115
116     /**
117      * Constant indicating the provided certificate failed
validation on the server side. Used as a possible value for the
118      * errorCode in {@link sap.AuthProxy~errorCallback}.
119      * @constant
120      * @type number
121      */
122     AuthProxy.prototype.ERR_CLIENT_CERTIFICATE_VALIDATION =
-108;
123
124     /**
125      * Constant indicating the server certificate failed
validation on the client side.  This is likely because the server
certificate
126      * is self-signed, or not signed by a well-known certificate
authority.  This constant is used as a possible value for the
127      * errorCode in {@link sap.AuthProxy~errorCallback}.
128      * @constant
129      * @type number
130      */
131     AuthProxy.prototype.ERR_SERVER_CERTIFICATE_VALIDATION =
-109;
132
133     /**
134      * Constant indicating the server request failed. Used as a
possible value for the
135      * errorCode in {@link sap.AuthProxy~errorCallback}.
136      * @constant
137      * @type number
138      */
139     AuthProxy.prototype.ERR_SERVER_REQUEST_FAILED = -110;
140
141     /**
```

```
142      * Constant indicating the logon manager core library is not
available.  Getting this error code means you tried

143      * to use Logon plugin features (for example, a certificate
from Logon) without adding the Logon plugin to the app.

144      * A possible value for the errorCode in {@link
sap.AuthProxy~errorCallback}.

145      * @constant

146      * @type number

147      */

148     AuthProxy.prototype.ERR_LOGON_MANAGER_CORE_NOT_AVAILABLE =
-111;

149

150     /**

151     * Constant indicating the logon manager certifciate method is
not available. Used as a possible value for the

152      * errorCode in {@link sap.AuthProxy~errorCallback}.

153      * @constant

154      * @type number

155      */

156
AuthProxy.prototype.ERR_LOGON_MANAGER_CERTIFICATE_METHOD_NOT_AVAILA
BLE = -112;

157

158     /**

159      * Constant indicating timeout error while connecting to the
server. Used as a possible value for the

160      * errorCode in {@link sap.AuthProxy~errorCallback}.

161      * @constant

162      * @type number

163      */

164     AuthProxy.prototype.ERR_HTTP_TIMEOUT = -120;

165

166     /**

167      * Constant indicating cordova domain whitelist rejection
error while sending request to server. Used as a possible value for
the
```

```
168      * errorCode in {@link sap.AuthProxy~errorCallback}.
169      * @constant
170      * @type number
171      */
172     AuthProxy.prototype.ERR_DOMAIN_WHITELIST_REJECTION = -121;
173
174     /**
175     * Constant indicating a missing required parameter message.
Used as a possible value for the description
176      * in (@link sap.AuthProxy~errorCallback}.
177      * @constant
178      * @type string
179      * @private
180      */
181     AuthProxy.prototype.MSG_MISSING_PARAMETER = "Missing a
required parameter: ";
182
183     /**
184     * Constant indicating invalid parameter value message.  Used
as a possible value for the description
185      * in (@link sap.AuthProxy~errorCallback}.
186      * @constant
187      * @type string
188      * @private
189      */
190     AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE = "Invalid
Parameter Value for parameter: ";
191
192     /**
193      * Create certificate source description object for a
certificate from a keystore file.  The keystore file must be of type
PKCS12
194      * (usually a .p12 extention) since that is the only
certificate file type that can contain a private key (a private key
is needed
```

```
195      * to authenticate the client to the server).  You might want
to use this method if you know the desired certificate resides in a

196      * file on the filesystem.

197      * @class

198      * @param {string} Path The Path of the keystore file.<br/>For
iOS clients, it first tries to load the

199      *                 relative file path from the application's
Documents folder. If it fails, it then tries

200      *              to load the file path from application's main
bundle. In addition, before trying

201      *              to load the certificate from the file system,
the iOS client first checks whether the

202      *               specified certificate key already exists in
the key store. If it does, it loads

203      *                 the existing certificate from key store,
instead of loading the certificate from

204      *                 file system.<br/>

205      *               For Android clients, the filepath is first
treated as an absolute path. If the certificate

206      *              is not found, then the filepath is treated as
relative to the root of the sdcard.

207      * @param {string} Password The password of the keystore.

208      * @param {string} CertificateKey A unique key (aka: alias)
that is used to locate the certificate.

209      * @example

210      * // Create the certificate source description object.

211      * var fileCert = new
sap.AuthProxy.CertificateFromFile("directory/certificateName.p12",
"certificatePassword", "certificateKey");

212      * // callbacks

213      * var successCB = function(serverResponse){

214      *     alert("Status: " +
JSON.stringify(serverResponse.status));

215      *     alert("Headers: " +
JSON.stringify(serverResponse.headers));

216      *     alert("Response: " +
JSON.stringify(serverResponse.response));

217      * }
```

```
218      * var errorCB = function(errorObject){
219      *     alert("Error making request: " +
JSON.stringify(errorObject));
220      * }
221     * // Make the request with the certificate source description
object.
222      * sap.AuthProxy.sendRequest("POST", "https://hostname",
headers, "THIS IS THE BODY", successCB, errorCB, null, null, 0,
fileCert);
223      *
224      */
225     AuthProxy.prototype.CertificateFromFile = function (Path,
Password, CertificateKey) {
226         this.Source = "FILE";
227         this.Path = Path;
228         this.Password = Password;
229         this.CertificateKey = CertificateKey;
230     };
231
232     /**
233      * Create a certificate source description object for
certificates from the system keystore.  You might want to use a
certificate
234     * from the system keystore if you know the user's device will
have the desired certificate installed on it.<br/>
235      * On Android, sending a request with a certificate from the
system store results in UI being shown for the user to pick
236      * the certificate to use (the certificate with the alias
matching the given CertificateKey is pre-selected).
237      * @class
238      * @param {string} CertificateKey A unique key (aka: alias)
that is used to locate the certificate.
239      * @example
240      * // Create the certificate source description object.
241      * var systemCert = new
sap.AuthProxy.CertificateFromStore("certificatekey");
242      * // callbacks
```

```
243       * var successCB = function(serverResponse){
244       *     alert("Status: " +
JSON.stringify(serverResponse.status));
245       *     alert("Headers: " +
JSON.stringify(serverResponse.headers));
246       *     alert("Response: " +
JSON.stringify(serverResponse.response));
247       * }
248       * var errorCB = function(errorObject){
249       *     alert("Error making request: " +
JSON.stringify(errorObject));
250       * }
251      * // Make the request with the certificate source description
object.
252       * sap.AuthProxy.sendRequest("POST", "https://hostname",
headers, "THIS IS THE BODY", successCB, errorCB, null, null, 0,
systemCert);
253       */
254      AuthProxy.prototype.CertificateFromStore = function
(CertificateKey) {
255          this.Source = "SYSTEM";
256          this.CertificateKey = CertificateKey;
257      };
258
259
260      /**
261       * Create a certificate source description object for
certificates from logon manager.  Using the resulting certificate
source description
262       * object on subsequent calls to AuthProxy.sendRequest or
AuthProxy.get will cause AuthProxy to retrieve a certificate from
Logon Manager
263       * to use for client authentication. The appID parameter is
used to indicate which application's certificate to use.<br/>
264       * Note that to use a certificate from Logon Manager, the
application must have already registered with the server using a
certificate from Afaria.
265       * @class
```

```
266       * @param {string} appID application identifier
267       * @example
268       * // Create the certificate source description object.
269       * var logonCert = new
sap.AuthProxy.CertificateFromLogonManager("applicationID");
270       * // callbacks
271       * var successCB = function(serverResponse){
272       *     alert("Status: " +
JSON.stringify(serverResponse.status));
273       *     alert("Headers: " +
JSON.stringify(serverResponse.headers));
274       *     alert("Response: " +
JSON.stringify(serverResponse.response));
275       * }
276       * var errorCB = function(errorObject){
277       *     alert("Error making request: " +
JSON.stringify(errorObject));
278       * }
279      * // Make the request with the certificate source description
object.
280       * sap.AuthProxy.sendRequest("POST", "https://hostname",
headers, "THIS IS THE BODY", successCB, errorCB, null, null, 0,
logonCert);
281       */
282      AuthProxy.prototype.CertificateFromLogonManager = function
(appID) {
283          this.Source = "LOGON";
284          this.AppID = appID;
285      };
286
287
288      /**
289       * Verifies that a certificate source description object
(created with {@link sap.AuthProxy#CertificateFromFile},
290       * {@link sap.AuthProxy#CertificateFromStore}, or {@link
sap.AuthProxy#CertificateFromLogonManager}) has all the required
fields and that the values
```

```
291      * for those fields are the correct type.  This function
verifies only the certificate description object, not the certificate
itself.  So, for example,

292      * if the certificate source description object was created
with {@link sap.AuthProxy#CertificateFromFile} and has a String for
the filepath and a

293      * String for the key/alias, <b>this function considers it
valid even if no certificate actually exists on the filesystem</b>.
If the certificate

294      * source description object is valid but the certificate
itself is not, then an error occurs during the call to {@link
sap.AuthProxy#get} or

295      * {@link sap.AuthProxy#sendRequest}.

296      * @param {object} certSource The certificate source
object.

297      * @param {sap.AuthProxy~errorCallback} errorCB The error
callback invoked if the certificate source is not valid.  Will have
an object with 'errorCode'

298      * and 'description' properties.

299      * @example

300      * var notValidCert = {};

301      * var errorCallback = function(error){

302      *     alert("certificate not valid!\nError code: " +
error.errorCode + "\ndescription: " + error.description);

303      * }

304      * var isCertValid =
sap.AuthProxy.validateCertSource(notValidCert, errorCallback);

305      * if( isCertValid ){

306      *     // do stuff with the valid certificate source
description object

307      * } else {

308     *    // at this point we know the cert is not valid, and the
error callback is invoked with extra information.

309      * }

310      *

311      *

312      * Developers are not expected to call this function.

313      * @private
```

```
314      */
315     AuthProxy.prototype.validateCertSource = function
(certSource, errorCB) {
316         if (!certSource) {
317             // The certificate is not present, so just ignore
it.
318             return true;
319         }
320
321         // errorCB required.
322         // First check this one. We may need it to return
errors
323         if (errorCB && (typeof errorCB !== "function")) {
324             console.log("AuthProxy Error: errorCB is not a
function");
325             return false;
326         }
327
328         try {
329             // First check whether it is an object
330             if (typeof certSource !== "object") {
331                 errorCB({
332                     errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
333                     description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certSource"
334                 });
335                 return false;
336             }
337
338             if (certSource.Source === "FILE") {
339                 if (!certSource.Path) {
340                     errorCB({
341                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
```

```
342                         description:
AuthProxy.prototype.MSG_MISSING_PARAMETER + "keystore path"
343                     });
344                     return false;
345                 }
346
347                 if (typeof certSource.Path !== "string") {
348                     errorCB({
349                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
350                         description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "keystore path"
351                     });
352                     return false;
353                 }
354
355                 if (!certSource.CertificateKey) {
356                     errorCB({
357                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
358                         description:
AuthProxy.prototype.MSG_MISSING_PARAMETER + "certificate key"
359                     });
360                     return false;
361                 }
362
363             if (typeof certSource.CertificateKey !== "string")
{
364                     errorCB({
365                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
366                         description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certificate key"
367                     });
368                     return false;
```

```
369                 }
370             } else if (certSource.Source === "SYSTEM") {
371                 if (!certSource.CertificateKey) {
372                     errorCB({
373                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
374                         description:
AuthProxy.prototype.MSG_MISSING_PARAMETER + "certificate key"
375                     });
376                     return false;
377                 }
378
379                 if (typeof certSource.CertificateKey !== "string")
{
380                     errorCB({
381                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
382                         description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certificate key"
383                     });
384                     return false;
385                 }
386             } else if (certSource.Source === "LOGON") {
387                 if (!certSource.AppID) {
388                     errorCB({
389                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
390                         description:
AuthProxy.prototype.MSG_MISSING_PARAMETER + "AppID"
391                     });
392                     return false;
393                 }
394
395                 if (typeof certSource.AppID !== "string") {
396                     errorCB({
```

```
397                         errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
398                         description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "AppID"
399                     });
400                     return false;
401                 }
402             } else {
403                 errorCB({
404                     errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
405                     description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certSource"
406                 });
407                 return false;
408             }
409
410             return true;
411         } catch (ex) {
412             errorCB({
413                 errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
414                 description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "certSource"
415             });
416         }
417     };
418
419
420     /**
421      * Send an HTTP(S) request to a remote server.  This function
is the centerpiece of the AuthProxy plugin.  It will handle
422      * mutual authentication if a certificate source is
provided.
423      * The success callback is invoked upon any response from the
server.  Even responses not generally considered to be
```

```
424     * successful (such as 404 or 500 status codes) will result in
the success callback being invoked.  The error callback

425      * is reserved for problems that prevent the AuthProxy from
creating the request or contacting the server.  It is therefore

426     * important to always check the status property on the object
given to the success callback.

427      * @param {string} method Standard HTTP request method
name.

428      * @param {string} url The HTTP URL with format http(s)://
[user:password]@hostname[:port]/path.

429      * @param {Object} header HTTP header to send to the server.
This is an Object. Can be null.

430     * @param {string} requestBody Data to send to the server with
the request. Can be null.

431      * @param {sap.AuthProxy~successCallback} successCB Callback
method invoked upon a response from the server.

432      * @param {sap.AuthProxy~errorCallback} errorCB Callback
method invoked in case of failure.

433      * @param {string} [user] User ID for basic authentication.

434      * @param {string} [password] User password for basic
authentication.

435      * @param {number} [timeout] Timeout setting in seconds.
Default timeout is 60 seconds.  A value of 0 means there is no
timeout.

436      * @param {Object} [certSource] Certificate description
object. It can be one of {@link sap.AuthProxy#CertificateFromFile},

437      * {@link sap.AuthProxy#CertificateFromStore}, or {@link
sap.AuthProxy#CertificateFromLogonManager}.

438      * @return {function} A JavaScript function object to abort
the operation.  Calling the abort function results in neither the
success or error

439     * callback being invoked for the original request (excepting
the case where the success or error callback was invoked before
calling the

440      * abort function).  Note that the request itself cannot be
unsent, and the server will still receive the request - the
JavaScript will just

441      * not know the results of that request.

442      * @example

443      * // callbacks
```

```
444       * var successCB = function(serverResponse){
445       *     alert("Status: " +
JSON.stringify(serverResponse.status));
446       *     alert("Headers: " +
JSON.stringify(serverResponse.headers));
447       *     alert("Response: " +
JSON.stringify(serverResponse.response));
448       * }
449       * var errorCB = function(errorObject){
450       *     alert("Error making request: " +
JSON.stringify(errorObject));
451       * }
452       * // To send a post request to the server, call the method
453       * var abortFunction = sap.AuthProxy.sendRequest("POST",
"http://www.google.com", null, "THIS IS THE BODY", successCB,
errorCB);
454       * // An example of aborting the request
455       * abortFunction();
456       *
457      * // To send a post request to the server with headers, call
the method
458       * sap.AuthProxy.sendRequest("POST", url, {HeaderName :
"Header value"}, "THIS IS THE BODY", successCB, errorCB);
459       *
460       * // To send a post request to the server with basic
authentication, call the method
461      * sap.AuthProxy.sendRequest("POST", url, headers, "THIS IS
THE BODY", successCB, errorCB, "username", "password");
462       *
463       * // To send a post request to the server with mutual
authentication, call the method
464       * sap.AuthProxy.sendRequest("POST", "https://hostname",
headers, "THIS IS THE BODY", successCB, errorCB, null,
465       *     null, 0, new
sap.AuthProxy.CertificateFromLogonManager("theAppId"));
466       */
```

```
467     AuthProxy.prototype.sendRequest = function (method, url,
header, requestBody, successCB, errorCB, user, password, timeout,
certSource) {
468
469         // errorCB required.
470         // First check this one. We may need it to return
errors
471         if (!errorCB || (typeof errorCB !== "function")) {
472             console.log("AuthProxy Error: errorCB is not a
function");
473           // if error callback is invalid, throw an exception to
notify the caller
474             throw new Error("AuthProxy Error: errorCB is not a
function");
475         }
476
477         // method required
478         if (!method) {
479             console.log("AuthProxy Error: method is required");
480             errorCB({
481                 errorCode:
AuthProxy.prototype.ERR_MISSING_PARAMETER,
482                 description:
AuthProxy.prototype.MSG_MISSING_PARAMETER + "method"
483             });
484             return;
485         }
486
487
488         // We only support GET, POST, HEAD, PUT, DELETE, PATCH
method
489         if (method !== "GET" && method !== "POST" && method !==
"HEAD" && method !== "PUT" && method !== "DELETE" && method !==
"PATCH") {
490           console.log("Invalid Parameter Value for parameter: "
+ method);
491             errorCB({
```

```
492                 errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
493                 description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "method"
494             });
495             return;
496         }
497
498
499         // url required
500         if (!url) {
501             console.log("AuthProxy Error: url is required");
502             errorCB({
503                 errorCode:
AuthProxy.prototype.ERR_MISSING_PARAMETER,
504                 description:
AuthProxy.prototype.MSG_MISSING_PARAMETER + "url"
505             });
506             return;
507         }
508
509
510         // successCB required
511         if (!successCB) {
512             console.log("AuthProxy Error: successCB is
required");
513             errorCB({
514                 errorCode:
AuthProxy.prototype.ERR_MISSING_PARAMETER,
515                 description:
AuthProxy.prototype.MSG_MISSING_PARAMETER + "successCB"
516             });
517             return;
518         }
519
```

```
520
521         if (typeof successCB !== "function") {
522             console.log("AuthProxy Error: successCB is not a
function");
523             errorCB({
524                 errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
525                 description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "successCB"
526             });
527             return;
528         }
529
530
531         if (user && typeof user !== "string") {
532             errorCB({
533                 errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
534                 description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "user"
535             });
536             return;
537         }
538
539
540         if (password && typeof password !== "string") {
541             errorCB({
542                 errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
543                 description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "password"
544             });
545             return;
546         }
547
```

```
548
549         if (timeout && typeof timeout !== "number") {
550             errorCB({
551                 errorCode:
AuthProxy.prototype.ERR_INVALID_PARAMETER_VALUE,
552                 description:
AuthProxy.prototype.MSG_INVALID_PARAMETER_VALUE + "timeout"
553             });
554             return;
555         }
556
557         if (!this.validateCertSource(certSource, errorCB)) {
558             return;
559         }
560
561
562         try {
563             var client = new Client(method, url, header,
requestBody, successCB, errorCB, user, password, timeout,
certSource);
564             return client.send();
565         } catch (ex) {
566             errorCB({
567                 errorCode: AuthProxy.prototype.ERR_UNKNOWN,
568                 description: ex.message
569             });
570         }
571
572     };
573
574     /**
575      * Send an HTTP(S) GET request to a remote server.  This is a
convenience function that simply calls {@link
sap.AuthProxy#sendRequest}
```

```
576      * with "GET" as the method and null for the request body.
All given parameters are passed as-is to sap.AuthProxy.sendRequest.

577      * The success callback is invoked upon any response from the
server.  Even responses not generally considered to be

578      * successful (such as 404 or 500 status codes) will result in
the success callback being invoked. The error callback

579      * is reserved for problems that prevent the AuthProxy from
creating the request or contacting the server.  It is, therefore,

580      * important to always check the status property on the object
given to the success callback.

581      * @param {string} url The URL against which to make the
request.

582      * @param {Object} header HTTP header to send to the server.
This is an Object. Can be null.

583      * @param {sap.AuthProxy~successCallback} successCB Callback
method invoked upon a response from the server.

584      * @param {sap.AuthProxy~errorCallback} errorCB Callback
method invoked in case of failure.

585      * @param {string} [user] User ID for basic authentication.

586      * @param {string} [password] User password for basic
authentication.

587      * @param {number} [timeout] Timeout setting in seconds.
Default timeout is 60 seconds.  A value of 0 means there is no
timeout.

588      * @param {Object} [certSource] Certificate description
object. It can be one of {@link sap.AuthProxy#CertificateFromFile},

589      * {@link sap.AuthProxy#CertificateFromStore}, or {@link
sap.AuthProxy#CertificateFromLogonManager}.

590      * @return {function} A JavaScript function object to abort
the operation.  Calling the abort function results in neither the
success or error

591      * callback being invoked for the original request (excepting
the case where the success or error callback was invoked before
calling the

592      * abort functino).  Note that the request itself cannot be
unsent, and the server will still receive the request - the
JavaScript will just

593      * not know the results of that request.

594      * @example

595      * var successCB = function(serverResponse){
```

```
596      *     alert("Status: " +
JSON.stringify(serverResponse.status));

597      *     alert("Headers: " +
JSON.stringify(serverResponse.headers));

598      *     if (serverResponse.responseText){

599      *         alert("Response: " +
JSON.stringify(serverResponse.responseText));

600      *     }

601      * }

602      * var errorCB = function(errorObject){

603      *     alert("Error making request: " +
JSON.stringify(errorObject));

604      * }

605      * // To send a GET request to server, call the method

606      * var abortFunction = sap.AuthProxy.get("http://
www.example.com", null, successCB, errorCB);

607      * // An example of aborting the request

608      * abortFunction();

609     * // To send a GET request to the server with headers, call
the method

610     * sap.AuthProxy.get("http://www.example.com", {HeaderName :
"Header value"}, successCB, errorCB);

611      * // To send a GET request to the server with basic
authentication, call the method

612      * sap.AuthProxy.get("https://www.example.com", headers,
successCB, errorCB, "username", "password");

613      * // To send a GET request to the server with mutual
authentication, call the method

614      * sap.AuthProxy.get("https://www.example.com", headers,
successCB, errorCB, null, null, 0,

615      *     new
sap.AuthProxy.CertificateFromLogonManager("theAppId"));

616      */

617     AuthProxy.prototype.get = function (url, header, successCB,
errorCB, user, password, timeout, certSource) {

618         return this.sendRequest("GET", url, header, null,
successCB, errorCB, user, password, timeout, certSource);

619     };
```

```
620
621     /**
622      * Delete a cached certificate from the keychain. iOS clients
always checks the cached certificate first to see if it is available
before
623      * loading the certificate from the file system. If the cached
certificate is no longer valid, use this method to delete it from the
keychain.
624      * <br/><b>Only supported on iOS platform, NOT Android.</
b>
625      * @param {sap.AuthProxy~deleteCertificateSuccessCallback}
successCB Callback method upon success.
626      * @param {sap.AuthProxy~errorCallback} [errorCB] Callback
method upon failure.
627      * @param {string} certificateKey The key of the certificate
to be deleted.
628      * @example
629      * var successCB = function(){
630      *     alert("certificate successfully deleted.");
631      * }
632      * var errorCB = function(error){
633      *     alert("error deleting certificate: " +
JSON.stringify(error));
634      * }
635      * sap.AuthProxy.deleteCertificateFromStore(successCB,
errorCB, "certificateKeyToDelete");
636      */
637     AuthProxy.prototype.deleteCertificateFromStore = function
(successCB, errorCB, certificateKey) {
638         cordova.exec(successCB, errorCB, "AuthProxy",
"deleteCertificateFromStore", [certificateKey]);
639     };
640
641     /**
642      * @private
643      */
```

```
644     var Client = function (method, url, header, requestBody,
successCB, errorCB, user, password, timeout, certSource) {
645
646         //ios plugin parameter does not support object type,
convert Header and CertSource to JSON string
647         if (device.platform === "iOS" || (device.platform &&
device.platform.indexOf("iP") === 0)) {
648             if (header) {
649                 header = JSON.stringify(header);
650             }
651             if (certSource) {
652                 certSource = JSON.stringify(certSource);
653             }
654         }
655
656         this.Method = method;
657         this.Url = url;
658         this.Header = header;
659         this.RequestBody = requestBody;
660         this.SuccessCB = successCB;
661         this.ErrorCB = errorCB;
662         this.User = user;
663         this.Password = password;
664         this.Timeout = timeout;
665         this.CertSource = certSource;
666         this.IsAbort = false;
667
668         this.abort = function () {
669             this.IsAbort = true;
670         };
671
672
673         this.send = function () {
```

```
674
675             var args = [this.Method, this.Url, this.Header,
this.RequestBody, this.User, this.Password, this.Timeout,
this.CertSource];
676
677             var me = this;
678
679             var successCallBack = function (data) {
680                 if (me.IsAbort === true) {
681                     return;
682                 }
683
684                 successCB(data);
685             };
686
687             var errorCallBack = function (data) {
688                 if (me.IsAbort === true) {
689                     return;
690                 }
691
692                 errorCB(data);
693             };
694
695             exec(successCallBack, errorCallBack, "AuthProxy",
"sendRequest", args);
696
697             return this.abort;
698         };
699     };
700
701     /**
702     * Generates an OData client that uses the AuthProxy plugin to
make requests.  This is useful if you are using Datajs, but want
```

```
703      * to make use of the certificate features of AuthProxy.
Datajs is a javascript library useful for accessing OData services.
704     * Datajs has a concept of an HttpClient, which does the work
of making the request.  This function generates an HttpClient that
705      * you can specify to Datajs so you can provide client
certificates for requests.  If you want to use the generated HTTP
client
706     * for all future Datajs requests, you can do that by setting
the OData.defaultHttpClient property to the return value of this
707      * function.  Once that is done, then doing OData stuff with
Datajs is almost exactly the same, but you can add a
708      * certificateSource to a request.
709      * @example
710      * OData.defaultHttpClient =
sap.AuthProxy.generateODataHttpClient();
711      *
712      * // Using a certificate from file, for example.
713      * fileCert = new sap.AuthProxy.CertificateFromFile("mnt/
sdcard/cert.p12", "password", "certKey");
714      *
715     * // This is the same request object you would have created
if you were just using Datajs, but now
716      * // you can add the extra 'certificateSource' property.
717      * var createRequest = {
718      *     requestUri: "http://www.example.com/stuff/etc/
example.svc",
719      *     certificateSource : fileCert,
720      *     user : "username",
721      *     password : "password",
722      *     method : "POST",
723      *     data:
724      *     {
725      *          Description: "Created Record",
726      *          CategoryName: "Created Category"
727      *     }
728      * }
```

```
729      *
730      * // Use Datajs to send the request.
731      * OData.request( createRequest, successCallback,
failureCallback );
732      *
733      */
734     AuthProxy.prototype.generateODataHttpClient = function () {
735        var httpClient = {
736             request: function (request, success, error) {
737               var url, requestHeaders, requestBody, statusCode,
statusText, responseHeaders;
738              var responseBody, requestTimeout, requestUserName,
requestPassword, requestCertificate;
739                 var client, result;
740
741                 url = request.requestUri;
742                 requestHeaders = request.headers;
743                 requestBody = request.body;
744
745                 var successCB = function (data) {
746                     var response = {
747                         requestUri: url,
748                         statusCode: data.status,
749                         statusText: data.status,
750                         headers: data.headers,
751                         body: (data.responseText ?
data.responseText : data.responseBase64)
752                     };
753
754                     if (response.statusCode >= 200 &&
response.statusCode <= 299) {
755                         if (success) {
756                             success(response);
757                         }
```

```
758                     } else {
759                         if (error) {
760                             error({
761                                 message: "HTTP request failed",
762                                 request: request,
763                                 response: response
764                             });
765                         }
766                     }
767                 };
768 
769                 var errorCB = function (data) {
770                     if (error) {
771                         error({
772                             message: data
773                         });
774                     }
775                 };
776 
777                 if (request.timeoutMS) {
778                     requestTimeout = request.timeoutMS / 1000;
779                 }
780 
781                 if (request.certificateSource) {
782                     requestCertificate =
request.certificateSource;
783                 }
784 
785                 if (request.user) {
786                     requestUserName = request.user;
787                 }
788 
```

```
789                 if (request.password) {
790                     requestPassword = request.password;
791                 }
792
793                 client =
AuthProxy.prototype.sendRequest(request.method || "GET", url,
requestHeaders, requestBody, successCB, errorCB, requestUserName,
requestPassword, requestTimeout, requestCertificate);
794
795                 result = {};
796                 result.abort = function () {
797                     client.abort();
798
799                     if (error) {
800                         error({
801                             message: "Request aborted"
802                         });
803                     }
804                 };
805                 return result;
806             }
807         };
808         return httpClient;
809     };
810
811     var AuthProxyPlugin = new AuthProxy();
812
813     module.exports = AuthProxyPlugin;
814
815
816     /**
817      * Callback function that is invoked in case of an error.
818      *
```

```
819      * @callback sap.AuthProxy~errorCallback
820      *
821      * @param {Object} errorObject An object containing two
properties: 'errorCode' and 'description.'
822      * The 'errorCode' property corresponds to one of the {@link
sap.AuthProxy} constants.  The 'description'
823     * property is a string with more detailed information of what
went wrong.
824      *
825      * @example
826      * function errorCallback(errCode) {
827     *   //Set the default error message. Used if an invalid code
is passed to the
828      *    //function (just in case) but also to cover the
829      *    //sap.AuthProxy.ERR_UNKNOWN case as well.
830      *    var msg = "Unkown Error";
831      *    switch (errCode) {
832      *       case sap.AuthProxy.ERR_INVALID_PARAMETER_VALUE:
833      *          msg = "Invalid parameter passed to method";
834      *          break;
835      *       case sap.AuthProxy.ERR_MISSING_PARAMETER:
836      *          msg = "A required parameter was missing";
837      *          break;
838      *       case sap.AuthProxy.ERR_HTTP_TIMEOUT:
839      *          msg = "The request timed out";
840      *          break;
841      *    };
842      *    //Write the error to the log
843      *    console.error(msg);
844      *    //Let the user know what happened
845      *    navigator.notification.alert(msg, null, "AuthProxy
Error", "OK");
846      * };
847      */
```

```
848
849     /**
850      * Callback function that is invoked upon a response from the
server.
851      *
852      * @callback sap.AuthProxy~successCallback
853      *
854      * @param {Object} serverResponse An object containing the
response from the server.  Contains a 'headers' property,
855      * a 'status' property, and a 'responseText' property.<br/>
856      * 'headers' is an object containing all the headers in the
response.<br/>
857      * 'status' is an integer corresponding to the HTTP status
code of the response.  It is important to check the status of
858      * the response, since <b>this success callback is invoked
upon any response from the server</b> - including responses that
are
859      * not normally thought of as successes (for example, the
status code could be 404 or 500).<br/>
860      * 'responseText' is a string containing the body of the
response.
861      */
862
863     /**
864      * Callback function that is invoked upon successfully
deleting a certificate from the store.
865      *
866      * @callback sap.AuthProxy~deleteCertificateSuccessCallback
867      */
```

Using the 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.

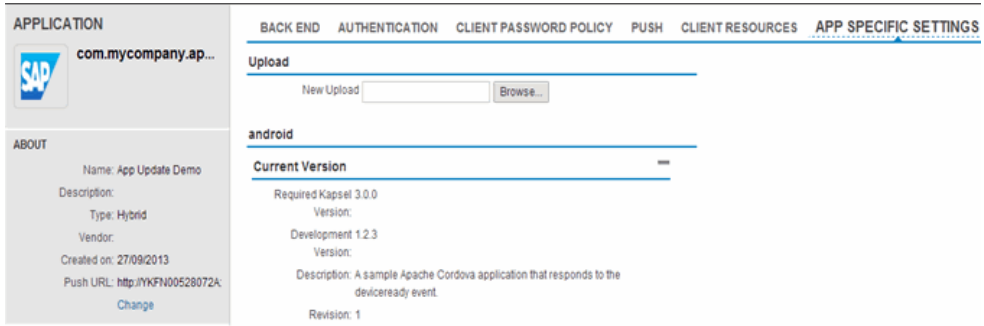
The AppUpdate plugin requires no developer programming, but includes a JavaScript API for customizing the way that application updates occur. The AppUpdate plugin operates in a default mode unless you handle the provided callback APIs.

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.



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

Domain Whitelisting

Kapsel plugins support Apache Cordova's domain whitelisting model. Whitelisting allows you to control access to external network resources. Apache Cordova whitelisting allows you to whitelist individual network resources (URLs), for example, http://www.google.com.

For information about the whitelist rules, see *http://docs.phonegap.com/en/3.3.0/guide_appdev_whitelist_index.md.html*.

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 185	Add a listener for an AppUpdate event.
reloadApp() on page 186	Replaces the app resources with any newly downloaded resources.
removeEventListener(eventname, f) on page 186	Removes a listener for an AppUpdate event.

reset() on page 187	Removes all local updates and loads the original web assets bundled with the app.
update() on page 187	Force an update check.

Events

Name	Description
checking on page 187	Event fired when AppUpdate is checking for an update.
downloading on page 188	Event fired when AppUpdate has found an update and is starting the download.
error on page 188	Event fired when AppUpdate encounters an error while checking for an update or downloading an update.
noupdate on page 189	Event fired when AppUpdate finds no available updates on server.
progress on page 189	Event fired when AppUpdate has made progress downloading the update.
updateready on page 190	Event fired when AppUpdate has a newly downloaded update available.

Source
appupdate.js, line 85 on page 195.

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 134 on page 197.

reloadApp() method

Replaces the app resources with any newly downloaded resources.

Syntax

<static> `reloadApp`()

Example

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

Source

appupdate.js, line 109 on page 196.

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 154 on page 197.

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 121 on page 196.

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 92 on page 195.

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 160 on page 197.

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 164 on page 198.

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 169 on page 198.

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 162 on page 197.

progress event
Event fired when AppUpdate has made progress downloading the update.

Properties

Name	Type	Default	Description
type	string	undefined	The name of the event.Value will be progress.
lengthComputable	boolean	undefined	Specifies whether or not the total size is known.
loaded	int	undefined	The number of bytes transferred so far.
total	int	undefined	The total number of bytes of content that will be transferred.If total size is unknown, this value is zero.

Type
object

Example

```
sap.AppUpdate.addEventListener('progress', function(e) {
    if (e.lengthComputable) {
         var percent = Math.round(e.loaded / e.total * 100);
         console.log("Progress " + percent);
    }
});
```

Since
3.0.2

Source
appupdate.js, line 167 on page 198.

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 updateready.
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 171 on page 198.

Source code

appupdate.js

```
1       // 3.0.2-SNAPSHOT
2       var exec = require('cordova/exec'),
3           channel = require('cordova/channel'),
4          logonFired = false, // Flag to determine if logon manager is done
5           promptActive = false, // Flag to prevent prompt from displaying more than once
6          bundle = null; // Internationalization. Loaded with device ready
7
8
9       // Event channels for AppUpdate
10      var channels = {
11          'checking': channel.create('checking'),
12          'noupdate': channel.create('noupdate'),
13          'downloading': channel.create('downloading'),
14          'progress': channel.create('progress'),
15          'error': channel.create('error'),
16          'updateready': channel.create('updateready')
17      };
18
19      // Holds the dom 0 handlers that are registered for the channels
20      var domZeroHandlers = {};
21
22      // Private callback that plugin calls for events
23      var _eventHandler = function (event) {
24          if (event.type) {
25              if (event.type in channels) {
26                  channels[event.type].fire(event);
27              }
28          }
```

```
29      };
30
31      /** @namespace sap */
32
33      /**
34       * Used to provide server-based updates to the application
content.
35       * <br/><br/>
36      * The AppUpdate plugin updates the contents of the www folder
of deployed Kapsel
37       * applications.  After an application successfully does a
logon to an SAP Mobile Platform 3
38       * server, the AppUpdate plugin is able to download an
available update. See Uploading Hybrid Apps in user documentation
39       * for information on how to upload an update to SAP Mobile
Platform 3 server.
40       * <br/><br/>
41       * After an update is completely downloaded, the application
user is
42       * prompted to install the update and restart the
application.  They can decline
43       * if they wish.
44       * <br/><br/>
45       * Once an update is installed, the application's revision
number is updated.
46       * <br/><br/>
47       * <b>Adding and Removing the AppUpdate Plugin</b><br/>
48       * The AppUpdate plugin is added and removed using the
49       * <a href="http://cordova.apache.org/docs/en/edge/
guide_cli_index.md.html#The%20Command-line%20Interface">Cordova
CLI</a>.<br/>
50       * <br/>
51       * To add the AppUpdate plugin to your project, use the
following command:<br/>
52       * cordova plugin add <path to directory containing Kapsel
plugins>\appupdate<br/>
53       * <br/>
```

```
54       * To remove the AppUpdate plugin from your project, use the
following command:<br/>
55       * cordova plugin rm com.sap.mp.cordova.plugins.appupdate
56       * <br/><br/>
57       *
58       * <b>Hybrid App Revision Preference</b><br/>
59       * This is an optional preference that tells the AppUdate
plugin if the local
60      * assets are uploaded to the server, and at what number.  If
this preference is
61       * not provided, the default revision is 0.
62       * In your config.xml file you can add the following
preference:<br/>
63       * <preference name="hybridapprevision" value="1" />
64      <br/>
65      <br/>
66       * This means that the local assets in your www folder are
uploaded to the server
67       * and the server is reporting revision 1 for them.  This
allows the application
68       * to receive a delta update when revision 2 is available
instead of a full update.
69       * <br/><br/>
70       *
71       * <b>Caveats</b><br/>
72       * It is important to test that your update has valid HTML,
Javascript, and CSS.
73       * Otherwise, the update could prevent the application from
functioning correctly,
74      * and may no longer be updateable.  You can test the updated
application in a
75      * separate simulator or additional test device.  You can also
validate your
76       * Javascript with tools like <a href="http://
www.jslint.com">JSLint</a>, or
77       * <a href="http://www.jshint.com">JSHint</a>.
```

```
78      * You can validate CSS with <a href="http://csslint.net">CSS
Lint</a>.
79       * <br/><br/>
80       *
81       * @namespace
82       * @alias AppUpdate
83       * @memberof sap
84       */
85      module.exports = {
86          /**
87           * Force an update check.  By default updates are done
automatically during logon and resume.
88           * See events for what will be fired during this
process.
89           * @example
90           * sap.AppUpdate.update();
91           */
92          update: function () {
93              // Abort if logon event has not yet fired
94              if (logonFired) {
95                  sap.Logon.unlock(function (connectionInfo) {
96                      //Add application ID required for REST call
97                      connectionInfo.applicationId =
sap.Logon.applicationId;
98
99                      exec(_eventHandler, null, 'AppUpdate',
'update', [connectionInfo]);
100                 });
101             }
102         },
103
104         /**
105          * Replaces the app resources with any newly downloaded
resources.
```

```
106          * @example
107          * sap.AppUpdate.reloadApp();
108          */
109         reloadApp: function () {
110             exec(null, null, 'AppUpdate', 'reloadApp', []);
111         },
112
113         /**
114          * Removes all local updates and loads the original web
assets bundled with the app. Call this after delete registration.
115         * Reset calls error callback if called during the update
process.
116          * @example
117          * sap.Logon.core.deleteRegistration(function() {
118          *     sap.AppUpdate.reset();
119          * }, function() {});
120          */
121         reset: function (successCallback, errorCallback) {
122             exec(successCallback, errorCallback, 'AppUpdate',
'reset', []);
123         },
124
125         /**
126         * Add a listener for an AppUpdate event.  See events for
available event names.
127          * @param {string} eventname Name of the app update
event.
128          * @param {function} f Function to call when event is
fired.
129          * @example
130        * sap.AppUpdate.addEventListener('checking', function(e)
{
131          *     console.log("Checking for update");
132          * });
133          */
```

```
134         addEventListener: function (eventname, f) {
135             if (eventname in channels) {
136                 channels[eventname].subscribe(f);
137             }
138         },
139 
140         /**
141         * Removes a listener for an AppUpdate event.  See events
for available event names.
142          * @param {string} eventname Name of the app update
event.
143          * @param {function} f Function that was registered.
144          * @example
145          * // Adding the listener
146          * var listener = function(e) {
147          *     console.log("Checking for update");
148          * });
149          * sap.AppUpdate.addEventListener('checking',
listener);
150          *
151          * // Removing the listener
152          * sap.AppUpdate.removeEventListener('checking',
listener);
153          */
154         removeEventListener: function (eventname, f) {
155             if (eventname in channels) {
156                 channels[eventname].unsubscribe(f);
157             }
158         }
159 
160         /**
161          * Event fired when AppUpdate is checking for an
update.
162          *
```

```
163          * @event sap.AppUpdate#checking
164          * @type {object}
165         * @property {string} type - The name of the event.  Value
will be checking.
166          * @example
167         * sap.AppUpdate.addEventListener('checking', function(e)
{
168          *     console.log("Checking for update");
169          * });
170          */
171
172         /**
173          * Event fired when AppUpdate finds no available updates
on server.
174          *
175          * @event sap.AppUpdate#noupdate
176          * @type {object}
177         * @property {string} type - The name of the event.  Value
will be noupdate.
178          * @example
179         * sap.AppUpdate.addEventListener('noupdate', function(e)
{
180          *     console.log("No update");
181          * });
182          */
183
184         /**
185          * Event fired when AppUpdate has found an update and is
starting the download.
186          *
187          * @event sap.AppUpdate#downloading
188          * @type {object}
189         * @property {string} type - The name of the event.  Value
will be downloading.
190          * @example
```

```
191          * sap.AppUpdate.addEventListener('downloading',
function(e) {
192          *     console.log("Downloading update");
193          * });
194          */
195
196
197         /**
198          * Event fired when AppUpdate has made progress
downloading the update.
199          *
200          * @since 3.0.2
201          * @event sap.AppUpdate#progress
202          * @type {object}
203         * @property {string} type - The name of the event.  Value
will be progress.
204          * @property {boolean} lengthComputable - Specifies
whether or not the total size is known.
205          * @property {int} loaded - The number of bytes
transferred so far.
206          * @property {int} total - The total number of bytes of
content that will be transferred.  If total size is unknown, this
value is zero.
207          * @example
208         * sap.AppUpdate.addEventListener('progress', function(e)
{
209          *     if (e.lengthComputable) {
210          *         var percent = Math.round(e.loaded / e.total *
100);
211          *           console.log("Progress " + percent);
212          *     }
213          * });
214          */
215
216         /**
```

```
217          * Event fired when AppUpdate encounters an error while
checking for an update or downloading an update.

218          * The status code and status message are provided with
this event.

219          *

220          * @event sap.AppUpdate#error

221          * @type {object}

222         * @property {string} type - The name of the event.  Value
will be error.

223          * @property {int} statusCode - The http error code.

224          * @property {string} statusMessage - The http status
message.

225          * @example

226          * sap.AppUpdate.addEventListener('error', function(e)
{

227         *     console.log("Error downloading update. statusCode:
" + e.statusCode + " statusMessage: " + e.statusMessage);

228          * });

229          */

230

231         /**

232          * Event fired when AppUpdate has a newly downloaded
update available.

233          * A default handler is already added to
sap.AppUpdate.onupdateready that will ask the user to reload the
app.

234          * When handling this event you should call
sap.AppUpdate.reloadApp() to apply the downloaded update.

235          *

236          * @event sap.AppUpdate#updateready

237          * @type {object}

238         * @property {string} type - The name of the event.  Value
will be updateready.

239          * @property {int} revision - The revision that was
downloaded.

240          * @example

241          *
```

```
242          * // This will listen for updateready event.
243         * // Note: Use sap.AppUpdate.onupdateready if you want to
override the default handler.
244          * sap.AppUpdate.addEventListener('updateready',
function(e) {
245          *     console.log("Update ready");
246          * });
247          *
248          * // Override default handler so that we automatically
load the update
249          * // without first prompting the user for permission,
250          * sap.AppUpdate.onupdateready = function(e) {
251          *     // No notification just reload
252          *     console.log("Apply application update...");
253          *     sap.AppUpdate.reloadApp();
254          * };
255          *
256         * // Override default handler with custom prompt to warn
the user that the
257          * // application is ready to update.
258          * sap.AppUpdate.onupdateready = function() {
259          *     console.log("Confirming application updateâ€¦");
260          *     navigator.notification.confirm('Update
Available',
261          *         function(buttonIndex) {
262          *             if (buttonIndex === 2) {
263         *                 console.log("Applying application updateâ
€¦");
264          *                     sap.AppUpdate.reloadApp();
265          *             }
266          *         },
267          *         "Update", ["Later", "Relaunch Now"]);
268          * };
269          */
```

```
270     };
271
272     // Add getter/setter for DOM0 style events
273     for (var type in channels) {
274         function defineSetGet(eventType) {
275             module.exports.__defineGetter__("on" + eventType,
function () {
276                 return domZeroHandlers[eventType];
277             });
278
279             module.exports.__defineSetter__("on" + eventType,
function (val) {
280                 // Remove current handler
281                 if (domZeroHandlers[eventType]) {
282                   module.exports.removeEventListener(eventType,
domZeroHandlers[eventType]);
283                 }
284
285                 // Add new handler
286                 if (val) {
287                     domZeroHandlers[eventType] = val;
288                     module.exports.addEventListener(eventType,
domZeroHandlers[eventType]);
289                 }
290             });
291         }
292
293         defineSetGet(type);
294     }
295
296     // Add default update ready implementation
297     module.exports.onupdateready = function () {
298         if (!promptActive) {
299             promptActive = true;
```

```
300
301             var onConfirm = function (buttonIndex) {
302                 promptActive = false;
303                 if (buttonIndex === 2) {
304                     // Only reload if we are unlocked
305                     sap.Logon.unlock(function (connectionInfo)
{
306                         //Add application ID required for REST
call
307                         connectionInfo.applicationId =
sap.Logon.applicationId;
308
309                         module.exports.reloadApp();
310                     });
311                 }
312             }
313
314             if (!bundle) {
315                 // Load required translations
316                 var i18n =
require('com.sap.mp.cordova.plugins.i18n.i18n');
317                 bundle = i18n.load({
318                     path: "plugins/
com.sap.mp.cordova.plugins.appupdate/www"
319                 });
320             }
321
322             window.navigator.notification.confirm(
323                 bundle.get("update_available"),
324                 onConfirm,
325                 bundle.get("update"), [bundle.get("later"),
bundle.get("relaunch_now")]);
326         }
327     }
```

```
328
329     // When logon is ready an update check is started
330     document.addEventListener("onSapLogonSuccess", function ()
{
331         logonFired = true;
332         module.exports.update();
333     }, false);
334
335     document.addEventListener("onSapResumeSuccess",
module.exports.update, false);
336
```

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.

Security for the log upload connection to SAP Mobile Platform Server is provided by using the security profile associated with the Application ID.

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");});
```

Domain Whitelisting

Kapsel plugins support Apache Cordova's domain whitelisting model. Whitelisting allows you to control access to external network resources. Apache Cordova whitelisting allows you to whitelist individual network resources (URLs), for example, http://www.google.com.

For information about the whitelist rules, see *http://docs.phonegap.com/en/3.3.0/guide_appdev_whitelist_index.md.html*.

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. Client requests to upload a log file are authenticated based on the application security profile. 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.

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)

If the log level is set to DEBUG, the application captures all log messages.

If you set the log level to INFO, the application captures INFO, WARN, and ERROR log messages.

If you set the log level to WARN, the application captures WARN and ERROR log messages.

If you set the log level to ERROR, the application captures only Error log messages.

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.

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.

The expected work flow, with the current architecture consists of the following:

1) If a user has an issue that needs to be analyzed by an administrator or developer, the user reports the issue as appropriate.

2) The administrator, or developer, enables the log collection for the user on the SAP Mobile Platform server.

3) The administrator lets the user know that he, or she, can upload log file.

4) The user uploads thelog file to the server, and the administrator gets the uploaded log file in the Management Cockpit.

5) The administrator sends the file to the developer to debug.

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.

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

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. To upload a log file automatically, settings plugin is required. 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 NONE is set in the Management Cockpit, Logger sets it to default log level.

Adding and Removing the Logger Plugin

Add or remove the Logger plugin 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 214	Constant variable for Debug log level.
Logger#ERROR on page 214	Constant variable for Error log level.
Logger#INFO on page 214	Constant variable for Information log level.
Logger#WARN on page 215	Constant variable for Warning log level.

Methods

Name	Description
debug(message, [tag], [successCallback], [errorCallback]) on page 215	Add a debug message to the log.
error(message, [tag], [successCallback], [errorCallback]) on page 216	Add an error message to the log.
getLogLevel(successCallback, [errorCallback]) on page 217	Get log level.
info(message, [tag], [successCallback], [errorCallback]) on page 219	Add an info message to the log.
setLogLevel(level, [successCallback], [errorCallback]) on page 220	Set log level.

upload(successCallback, errorCallback) on page 222	Upload a log file, with log entries, to SAP Mobile Platform server.
warn(message, [tag], [successCallback], [errorCallback]) on page 223	Add a warning message to the log.

Source
logger.js, line 69 on page 228.

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 354 on page 238.

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 324 on page 237.

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 344 on page 238.

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 334 on page 238.

debug(message, [tag], [successCallback], [errorCallback]) method

Add a debug message to the log.

This function logs messages with the '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 value added to the log entry used to indicate the source of the message (for example, SMP_LOGGER, SMP_AUTHPROXY).

successCallback	function	(optional)	Callback function called when the message has been successfully added to the log.No object will be passed to success callback.
errorCallback	function	(optional)	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.

Example

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

Source

logger.js, line 83 on page 228.

error(message, [tag], [successCallback], [errorCallback]) method

Add an error message to the log.

This function logs messages with the '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 value added to the log entry used to indicate the source of the message (for example, SMP_LOGGER, SMP_AUTHPROXY).
successCallback	function	(optional)	Callback function called when the message has been successfully added to the log.No object will be passed to success callback.
errorCallback	function	(optional)	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.

Example

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

Source

logger.js, line 152 on page 231.

getLogLevel(successCallback, [errorCallback]) method

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.

Syntax

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

Parameters

Name	Type	Argument	Description
successCallback	function		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.
errorCallback	function	(optional)	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.

Example

```
sap.Logger.getLogLevel(successCallback, errorCallback);

function successCallback(logLevel) {
    alert("Log level is " + logLevel);
}

function errorCallback() {
    alert("Failed to get log level");
}
```

Source
logger.js, line 228 on page 234.

info(message, [tag], [successCallback], [errorCallback]) method
Add an info message to the log.

This function logs messages with the '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 value added to the log entry used to indicate the source of the message (for example, SMP_LOGGER, SMP_AUTHPROXY).
successCallback	function	(optional)	Callback function called when the message has been successfully added to the log.No object will be passed to success callback.

errorCallback	function	(optional)	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.

Example

```
sap.Logger.info("info message", "INFO_TAG");
```

Source

logger.js, line 106 on page 229.

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 messages

WARN : ERROR and WARN messages

INFO : ERROR, WARN and INFO

DEBUG : ERROR, WARN, INFO and DEBUG

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.

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)	Callback function called when the log level has been successfully set.No object will be passed to success callback.
errorCallback	function	(optional)	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.

Example

```
sap.Logger.setLogLevel(sap.Logger.DEBUG, successCallback,
errorCallback);

function successCallback() {
    alert("Log level set");
}

function errorCallback() {
```

```
    alert("Failed to set log level");
}
```

Source
logger.js, line 175 on page 232.

upload(successCallback, errorCallback) method
Upload a log file, with log entries, to SAP Mobile Platform server.

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.

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.

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.

Syntax
<static> `upload`(*successCallback*, *errorCallback*)

Parameters

Name	Type	Description
successCallback	function	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)

errorCallback	function	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.

Example

```
sap.Logger.upload(successCallback, errorCallback);

function successCallback() {
    alert("Upload Successful");
}

function errorCallback(e) {
    alert("Upload Failed. Status: " + e.statusCode + ", Message: " +
e.statusMessage);
}
```

Source

logger.js, line 259 on page 235.

warn(message, [tag], [successCallback], [errorCallback]) method

Add a warning message to the log.

This function logs messages with the '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 value added to the log entry used to indicate the source of the message (for example, SMP_LOGGER, SMP_AUTHPROXY).

successCallback	function	(optional)	Callback function called when the message has been successfully added to the log.No object will be passed to success callback.
errorCallback	function	(optional)	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.

Example

```
sap.Logger.warn("warn message", "WARN_TAG");
```

Source

logger.js, line 129 on page 230.

Source code

logger.js

```
1       // 3.0.2-SNAPSHOT
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        *
```

```
8        * The Logger plugin has ERROR, WARN, INFO, and DEBUG log
levels and log messages are captured based on the configured and
selected log level.

9        * A Kapsel application can be set to these log levels by
programmatic control, and by the administrator changing a setting on
the server.

10       * For Android and iOS, the default log level is ERROR, so by
default only ERROR level logs are captured.

11       * sap.Logger.setLogLevel() method is used to set other
levels. If you want to get log messages at all log levels,

12       * you must set the log level to DEBUG. (DEBUG < INFO < WARN <
ERROR) <br>

13       * If the log level is set to DEBUG, the application captures
all log messages. <br>

14       * If you set the log level to INFO, the application captures
INFO, WARN, and ERROR log messages. <br>

15       * If you set the log level to WARN, the application captures
WARN and ERROR log messages. <br>

16      * If you set the log level to ERROR, the application captures
only Error log messages.

17       * <br><br>

18       *

19       * Using the provided sap.Logger.upload() method allows
developers to upload a log file to SAP Mobile Platform Server,

20       * where an administrator can view them and remotely set the
appropriate log level to control the amount of information

21       * that is written to the log. When the sap.Logger.upload()
method is triggered, a log file will be uploaded.

22       * If the Log Upload checkbox is selected in the Management
Cockpit, the client can upload a log file by calling
sap.Logger.upload().

23       * 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.

24       * To support manual uploading of the log, you should
implement a button or some other mechanism that calls
sap.Logger.upload() when needed.

25       * <br>

26      * 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.
```

```
27       * <br>
28       * The expected work flow, with the current architecture
consists of the following: <br>
29      * 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>
30       * 2) The administrator, or developer, enables the log
collection for the user on the SAP Mobile Platform server.<br>
31       * 3) The administrator lets the user know that he, or she,
can upload log file. <br>
32       * 4) The user uploads thelog file to the server, and the
administrator gets the uploaded log file in the Management
Cockpit.<br>
33       * 5) The administrator sends the file to the developer to
debug.
34       * <br><br>
35       *
36       * Currently, on iOS, if the current log level is ERROR
(default level), only ERROR level messages are displayed on the
console
37      * even if other log level messages are generated. But if the
current log level is DEBUG, INFO, or WARN,
38       * all generated log messages, regardless of log level, are
displayed on the console. <br>
39      * On Android, if the current log level is ERROR, only ERROR
level messages are displayed in the Android logcat view (console).
40       * if log level is INFO, then ERROR, WARN and INFO level
messages are displayed in the Android logcat view (console).
41       *
42       * 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
43       * 3) Call sap.Logger.upload() after a logon success event,
for example, when the app is launched or resumed and logon is
successful.
44      * The Settings plugin retrieves the selected log level(type)
from the Management Cockpit on the server,
45      * sets the log level to Logger plugin, and then automatically
uploads a log file to the server.
```

```
46       * 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.

47       * To upload a log file automatically, settings plugin is
required.

48       * 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).

49       * 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.

50       * Since the Kapsel Logger plugin supports only DEBUG, INFO,
WARN, and ERROR, the Logger plugin implicitly matches FATAL to ERROR,
and PATH to DEBUG.

51       * If NONEÂ isÂ setÂ inÂ the Management Cockpit,
LoggerÂ setsÂ itÂ toÂ defaultÂ log level.

52       * <br><br>

53       * <b>Adding and Removing the Logger Plugin</b><br>

54       * Add or remove the Logger plugin using the

55       * <a href="http://cordova.apache.org/docs/en/edge/
guide_cli_index.md.html#The%20Command-line%20Interface">Cordova
CLI</a>.<br>

56       * <br>

57       * To add the Logger plugin to your project, use the following
command:<br>

58       * Cordova plugin add <path to directory containing Kapsel
plugins>\logger<br>

59       * <br>

60       * To remove the Logger plugin from your project, use the
following command:<br>

61       * cordova plugin rm com.sap.mp.cordova.plugins.logger

62       * <br>

63       *

64       * @namespace

65       * @alias Logger

66       * @memberof sap

67       */

68
```

```
69      Logger = function () {
70          /**
71           * Formating for message
72           * @private
73           */
74          var format = function (message) {
75              if ((message === null) || (message === undefined))
{
76                  return "";
77              }
78
79              return message.toString();
80          }
81
82
83          /**
84           * Add a debug message to the log.
85           * This function logs messages with the 'DEBUG' log
level.
86           *
87           * @memberof sap.Logger
88           * @method debug
89           * @param {String} message Log message to be logged.
90          * @param {String} [tag]   Tag value added to the log entry
used to indicate the source of the message (for example, SMP_LOGGER,
SMP_AUTHPROXY).
91           * @param {function} [successCallback]  Callback function
called when the message has been successfully added to the log.
92          *                                  No object will be passed
to success callback.
93           * @param {function} [errorCallback]    Callback function
called when an error occurs while adding the message to the log.
94           *                                      Since Kapsel Logger
native code will always call the success callback function, the
95           *                                   errorCallback function
will be executed by Cordova if an error or exception occurs
```

```
96          *                                   while making the call to
the plugin.
97           * @public
98           * @memberof sap.Logger
99           * @example
100          * sap.Logger.debug("debug message", "DEBUG_TAG");
101          */
102         this.debug = function (message, tag, successCallback,
errorCallback) {
103             exec(successCallback, errorCallback, "Logging",
"logDebug", [format(message), tag]);
104         }
105
106         /**
107          * Add an info message to the log.
108          * This function logs messages with the 'INFO' log
level.
109          *
110          * @memberof sap.Logger
111          * @method info
112          * @param {String} message Log message to be logged.
113         * @param {String} [tag]   Tag value added to the log entry
used to indicate the source of the message (for example, SMP_LOGGER,
SMP_AUTHPROXY).
114         * @param {function} [successCallback]  Callback function
called when the message has been successfully added to the log.
115         *                                   No object will be passed
to success callback.
116         * @param {function} [errorCallback]    Callback function
called when an error occurs while adding the message to the log.
117          *                                     Since Kapsel Logger
native code will always call the success callback function, the
118         *                                    errorCallback function
will be executed by Cordova if an error or exception occurs
119         *                                   while making the call to
the plugin.
120          * @public
```

```
121          * @memberof sap.Logger
122          * @example
123          * sap.Logger.info("info message", "INFO_TAG");
124          */
125         this.info = function (message, tag, successCallback,
errorCallback) {
126             exec(successCallback, errorCallback, "Logging",
"logInfo", [format(message), tag]);
127         }
128
129         /**
130          * Add a warning message to the log.
131          * This function logs messages with the 'WARN' log
level.
132          *
133          * @memberof sap.Logger
134          * @method warn
135          * @param {String} message Log message to be logged.
136         * @param {String} [tag]   Tag value added to the log entry
used to indicate the source of the message (for example, SMP_LOGGER,
SMP_AUTHPROXY).
137         * @param {function} [successCallback]  Callback function
called when the message has been successfully added to the log.
138         *                                     No object will be passed
to success callback.
139         * @param {function} [errorCallback]    Callback function
called when an error occurs while adding the message to the log.
140         *                                       Since Kapsel Logger
native code will always call the success callback function, the
141         *                                      errorCallback function
will be executed by Cordova if an error/exception occurs
142         *                                     while making the call to
the plugin.
143          * @public
144          * @memberof sap.Logger
145          * @example
146          * sap.Logger.warn("warn message", "WARN_TAG");
```

```
147          */
148         this.warn = function (message, tag, successCallback,
errorCallback) {
149             exec(successCallback, errorCallback, "Logging",
"logWarning", [format(message), tag]);
150         }
151
152         /**
153          * Add an error message to the log.
154          * This function logs messages with the 'ERROR' log
level.
155          *
156          * @memberof sap.Logger
157          * @method error
158          * @param {String} message Log message to be logged.
159         * @param {String} [tag]   Tag value added to the log entry
used to indicate the source of the message (for example, SMP_LOGGER,
SMP_AUTHPROXY).
160         * @param {function} [successCallback]  Callback function
called when the message has been successfully added to the log.
161         *                                     No object will be passed
to success callback.
162         * @param {function} [errorCallback]    Callback function
called when an error occurs while adding the message to the log.
163         *                                       Since Kapsel Logger
native code will always call the success callback function, the
164         *                                      errorCallback function
will be executed by Cordova if an error or exception occurs
165        *                                     while making the call to
the plugin.
166          * @public
167          * @memberof sap.Logger
168          * @example
169          * sap.Logger.error("error message", "ERROR_TAG");
170          */
171         this.error =  function (message, tag, successCallback,
errorCallback) {
```

```
172             exec(successCallback, errorCallback, "Logging",
"logError", [format(message), tag]);

173         }

174

175         /**

176          * Set log level.

177          * This function sets the log level for logging. <br>

178          * Coverage of logging data in each log level:  DEBUG <
INFO < WARN < ERROR. <br>

179          * Following is the expected behavior to cover log
messages at specific log levels: <br>

180          *    ERROR : only ERROR messages <br>

181          *    WARN  : ERROR and WARN messages <br>

182          *    INFO  : ERROR, WARN and INFO <br>

183          *    DEBUG : ERROR, WARN, INFO and DEBUG <br>

184          * For example, if you want to get all log messages, you
need to set the log to the 'Debug' level.

185         * If the WARN level is set, logging data contains WARN and
ERROR messages. <br>

186          * Default log level is ERROR.

187          *

188          * @memberof sap.Logger

189          * @method setLogLevel

190          * @param {String} level Log level to set [DEBUG, INFO,
WARN, ERROR]

191         * @param {function} [successCallback] Callback function
called when the log level has been successfully set.

192         *                                  No object will be passed
to success callback.

193          * @param {function} [errorCallback]   Callback function
called when an error occurs while setting the log level.

194          *                                      Since Kapsel Logger
native code will always call the success callback function, the

195         *                                   errorCallback function
will be executed by Cordova if an error or exception occurs

196         *                                  while making the call to
the plugin.
```

```
197          * @memberof sap.Logger
198          * @example
199          * sap.Logger.setLogLevel(sap.Logger.DEBUG,
successCallback, errorCallback);
200          *
201          * function successCallback() {
202          *     alert("Log level set");
203          * }
204          *
205          * function errorCallback() {
206          *     alert("Failed to set log level");
207          * }
208          */
209         this.setLogLevel = function (level, successCallback,
errorCallback) {
210             if (level.toLowerCase() === "fatal")
211                 level = "ERROR";
212             else if (level.toLowerCase() === "path")
213                 level = "DEBUG";
214             else if (level.toLowerCase() === "warning")
215                 level = "WARN";
216             else if (level.toLowerCase() === "debug")
217                 level = "DEBUG";
218             else if (level.toLowerCase() === "info")
219                 level = "INFO";
220             else if (level.toLowerCase() === "error")
221                 level = "ERROR";
222             else if (level.toLowerCase() === "none")
223                 level = "ERROR";
224
225             exec(successCallback, errorCallback, "Logging",
"setLogLevel", [level]);
226         }
```

```
227
228         /**
229          * Get log level.
230          * This function gets the current log level.
231          * Use this function to know what kind of log level
messages can be generated and affected at the current log level.
232          *
233          * @memberof sap.Logger
234          * @method getLogLevel
235          * @param {function} successCallback Callback function
called when the log level has been successfully retrieved.
236         *                                  When the current log level
is successfully retrieved, it is fired with the current log level.
[DEBUG, INFO, WARN, ERROR]
237         *                                    Log level of String type
will be passed to success callback.
238          *                                       Default log level is
ERROR.
239          * @param {function} [errorCallback] Callback function
called when an error occurs while getting the current log level.  For
this method, error callback is optional.
240         *                                  Since Kapsel Logger native
code will always call the success callback function, the
241          *                                     errorCallback function
will be executed by Cordova if an error or exception occurs
242         *                                    while making the call to
the plugin.
243          * @memberof sap.Logger
244          * @example
245          * sap.Logger.getLogLevel(successCallback,
errorCallback);
246          *
247          * function successCallback(logLevel) {
248          *     alert("Log level is " + logLevel);
249          * }
250          *
251          * function errorCallback() {
```

```
252          *     alert("Failed to get log level");
253          * }
254          */
255         this.getLogLevel = function(successCallback,
errorCallback) {
256             exec(successCallback, errorCallback, "Logging",
"getLogLevel",[]);
257         }
258
259         /**
260          * Upload a log file, with log entries, to SAP Mobile
Platform server.<br>
261         * This function uploads a log file, which is helpful for
collecting logging data from the app to trace bugs and issues.
262         * It uploads a log file, which contains log entries based
on log level.
263          * Developers can access the log data in the Management
Cockpit and/or a specific folder in installed server
directly.<br><br>
264          *
265          * On iOS, the uploaded log messages are filtered by the
log level at upon upload.
266          * For example, when you upload a log file with an ERROR
log level, the uploaded log messages contain only ERROR log level
messages.
267         * When you upload log files with an INFO level, uploaded
log messages contain ERROR, WARN, and INFO log level messages.
268          *
269          * <br><br>
270         * On Android, generated log messages are filtered "at the
log level."
271         * In other words, the already generated and filtered log
messages at another log level are not affected by the current log
level.
272         * 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.
273         * 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.
274          *
275          * @memberof sap.Logger
276          * @method upload
277          * @param {function} successCallback Callback function
called when a log file is successfully uploaded to the server.
278          *                                     When a log file is
successfully uploaded, it is fired. (with http statusCode and
statusMessage for success)
279          * @param {function} errorCallback   Callback function
called when an error occurs while uploading a log file to the
server.
280         *                                If there is a connectivity
error, such as an HTTP error, or unknown server error,
281          *                                    it is fired with http
statusCode and statusMessage for error.
282          * @public
283          * @memberof sap.Logger
284          * @example
285          * sap.Logger.upload(successCallback, errorCallback);
286          *
287          * function successCallback() {
288          *     alert("Upload Successful");
289          * }
290          *
291          * function errorCallback(e) {
292         *    alert("Upload Failed. Status: " + e.statusCode + ",
Message: " + e.statusMessage);
293          * }
294          */
295         this.upload =  function (successCallback, errorCallback)
{
296             sap.Logon.core.getState (function (result) {
297                     if(result.status === "new") {
```

```
298                             errorCallback({statusCode : 0,
statusMessage : "Logon in " + result.status + " state. Registration
is required!!"});

299                         } else {

300                          sap.Logon.unlock(function (connectionInfo)
{

301                                 //Add application ID required for
REST call

302                                   connectionInfo.applicationId =
sap.Logon.applicationId;

303                                exec(successCallback, errorCallback,
"Logging", "uploadLog", [connectionInfo]);

304                                   }, function () {

305                                   errorCallback({statusCode : 0,
statusMessage : "Logon failed"});

306                             });

307                         }

308                     },

309                     function (){

310                      errorCallback({statusCode : 0, statusMessage :
"Failed to get current logon state."});

311                     }

312                 );

313             }

314         }

315

316         /**

317         * Constant variable for Error log level. It contains "ERROR"
string.

318          * @memberof sap.Logger

319          * @constant

320          * @type String

321          * @example

322          * sap.Logger.setLogLevel(sap.Logger.ERROR);

323          */

324         Logger.prototype.ERROR = "ERROR";
```

```
325
326     /**
327     * Constant variable for Warning log level. It contains "WARN"
string.
328      * @memberof sap.Logger
329      * @constant
330      * @type String
331      * @example
332      * sap.Logger.setLogLevel(sap.Logger.WARN);
333      */
334     Logger.prototype.WARN = "WARN";
335
336     /**
337      * Constant variable for Information log level. It contains
"INFO" string.
338      * @memberof sap.Logger
339      * @constant
340      * @type String
341      * @example
342      * sap.Logger.setLogLevel(sap.Logger.INFO);
343      */
344     Logger.prototype.INFO = "INFO";
345
346     /**
347     * Constant variable for Debug log level. It contains "DEBUG"
string.
348      * @memberof sap.Logger
349      * @constant
350      * @type String
351      * @example
352      * sap.Logger.setLogLevel(sap.Logger.DEBUG);
353      */
354     Logger.prototype.DEBUG = "DEBUG";
```

```
355
356     module.exports = new Logger();
357
```

Using the 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*.

Domain Whitelisting

Kapsel plugins support Apache Cordova's domain whitelisting model. Whitelisting allows you to control access to external network resources. Apache Cordova whitelisting allows you to whitelist individual network resources (URLs), for example, http://www.google.com.

For information about the whitelist rules, see *http://docs.phonegap.com/en/3.3.0/guide_appdev_whitelist_index.md.html*.

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 Developers Console.
2. Click **Create Project** to create an API project.
3. Enter a project name and click **Create** .

   A page displays your project ID and project number.
4. Record the project number for later use as the GCM sender ID.
5. Enable the GCM service.
   a) In the sidebar on the left, select **APIs & auth.**

b) In the displayed list of APIs, turn the **Google Cloud Messaging for Android** toggle to ON.

6. Obtain an API key.

a) In the sidebar on the left, select **APIs & auth** > **Credentials**.

b) Under **Public API access**, click **Create new key**.

c) In the **Create a new key** dialog, click **Android key**.

d) In the resulting configuration dialog, supply one SHA1 fingerprint and the package name for your app, separated by a semicolon. For example, `45:B5:E4:6F:36:AD:0A:98:94:B4:02:66:2B:12:17:F2:56:26:A0:E0;com.myexample`.

To get the value for the SHA1 fingerprint, follow the instructions in the *console help* .

e) Click **Create**.

f) Record the API key for later use to perform authentication in your application server.

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. This is the application or bundled identifier that the application developer assigns or generates during application development. The administrator uses the Application ID to register the application to SAP Mobile Platform Server, and the client application code uses the Application ID while sending requests to the server.
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`.

Header Structure (SAP Mobile Platform and later)	Consists of
`<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 `UILaunchImage-File` 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.

Header Structure (SAP Mobile Platform and later)	Consists of
<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 254	This method checks for any notifications received while the application was not running in the foreground.
getBadgeNumber(callback) on page 255	Used to fetch the badge count for the application.
registerForNotificationTypes(types, successCallback, errorCallback, notificationlistenerfunc, [senderId]) on page 256	Function called by the application to register notification types to receive.
resetBadge(callback) on page 257	Used to reset the badge count for the application.
setBadgeNumber(number, callback) on page 258	Used to set the badge count for the application.
unregisterForNotificationTypes(callback) on page 258	Function called by the application to unregister from future notifications.

Type Definitions

Name	Description
callback([devtok]) on page 259	This method updates the application with the new device token in the SAP Mobile Platform server.

Source
push.js, line 29 on page 261.

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 354 on page 273.

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 261 on page 270.

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 at-tempt fails.

notificationlistener-func	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 214 on page 268.

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 298 on page 271.

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 callback 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 279 on page 270.

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 243 on page 269.

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 319 on page 272.

Source code

push.js

```
1       // 3.0.2-SNAPSHOT
2       var exec = require("cordova/exec");
```

```
3
4       /**
5        * The push plugin provides an abstraction layer over the
6        * <a href="http://developer.android.com/google/gcm/
index.html">Google Cloud Messaging for Android (GCM)</a>
7        * and
8        * <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>.
9        * <br/><br/>
10       * A notification can be sent to a device registered with an
application through a
11       * rest call at <pre>http://SMP_3.0_SERVER:8080/
Notifications/application_registration_id</pre>
12       * <br/><br/>
13       * <b>Adding and Removing the Push Plugin</b><br/>
14       * The Push plugin is added and removed using the
15       * <a href="http://cordova.apache.org/docs/en/edge/
guide_cli_index.md.html#The%20Command-line%20Interface">Cordova
CLI</a>.<br/>
16       * <br/>
17       * To add the Push plugin to your project, use the following
command:<br/>
18       * cordova plugin add <path to directory containing Kapsel
plugins>\push<br/>
19       * <br/>
20       * To remove the Push plugin from your project, use the
following command:<br/>
21       * cordova plugin rm com.sap.mp.cordova.plugins.push
22       * <br/>
23       *
24       * @namespace
25       * @alias Push
26       * @memberof sap
27       */
28
```

```
29      module.exports = {
30
31
32
33          /**
34           * Helper method for handling failure callbacks. It is
configured as a failure callback in <code> call_native() </code>
35           *
36           *
37           * @param {msg} Error message with the cause of failure
38           *
39           * @private
40           * @name failure
41           * @function
42           */
43
44      failure: function (msg) {
45          sap.Logger.debug("Javascript Callback Error: " +
msg,"PUSHJS",function(m){},function(m){});
46
47      },
48          /**
49           * Helper method for handling push registration success
callbacks. It is configured as a failure callback in <code>
call_native() </code>
50           *
51           *
52           * @param {msg} Error message with the cause of failure
53           *
54           * @private
55           * @name pushregsuccsss
56           * @function
57           */
58
```

```
59      pushregsuccsss: function(msg) {
60          sap.Logger.debug("Javascript Callback Success
","PUSHJS",function(m){},function(m){});
61      },
62
63          /**
64           * Helper method for calling native methods
65           *
66           * @param {function} callback
67           * @param {string} Name of the action to invoke on the
plugin
68           * @param {array} List of arguments
69           * @private
70           * @name call_native
71           * @function
72           */
73      call_native: function (callback, name, args) {
74
75          if(arguments.length == 2) {
76              args = []
77          }
78          ret = exec(
79                  callback,              /**  Called when signature
capture is successful */
80                      sap.Push.failure,            /** Called when
signature capture encounters an error */
81                  'SMPPushPlugin',        /**  Tell Cordova that we
want to run "PushNotificationPlugin" */
82                   name,                   /**  Tell the plugin the
action we want to perform */
83                  args);                 /**  List of arguments to
the plugin */
84          return ret;
85      },
86
```

```
87          /**
88           * Helper method to check if platform is iOS.
89           *
90           * @return {bool} Whether the current platform is iOS or
not.
91           * @private
92           * @name isPlatformIOS
93           * @function
94           */
95      isPlatformIOS: function () {
96          return device.platform == "iPhone" || device.platform ==
"iPad" || device.platform == "iPod touch" || device.platform ==
"iOS"
97      },
98          /**
99           * Function called by the application to get connection
information.
100          *
101          * @param {string} [types] Types of notifications the
application wants to receive. The different types of notifications
are expressed in <code>notificationType</code>
102          * @param {string} [successCB] Success callback to call
when registration is successful.
103         * @param {string} [errorCB] Error callback to call when
registration attempt fails.
104          * @private
105          * @memberof sap.Push
106          * @function getConnectionSettings
107          * @example
108          * sap.Push.getConnectionSettings(function(){
109          * sap.Logger.debug("getting Connection
Settings","PUSHJS",function(m){},function(m){});
110          *  console.log("getting Connection Settings");
111          *  sap.Push.registerForNotification(types,
successCallback, errorCallback, notificationListenerFunc,
senderId );
112          **/
```

```
113         getConnectionSettings : function (successCB, errorCB) {
114
115
116             if (sap.Settings.isInitialized == true)
117             {
118                 /*It is already initialized */
119                 successCB();
120
121             } else {
122                 sap.Settings.isInitialized = true;
123                 var pd ="";
124                 sap.Logon.unlock(function (connectionInfo) {
125                                     var userName =
connectionInfo["registrationContext"]["user"];
126                                     var password  =
connectionInfo["registrationContext"]["password"];
127                                     var applicationConnectionId =
connectionInfo["applicationConnectionId"];
128                                     var securityConfig =
connectionInfo["registrationContext"]["securityConfig"];
129                                     var endpoint =
connectionInfo["applicationEndpointURL"];
130                                     var keySSLEnabled = "false";
131                                     var splitendpoint =
endpoint.split("/");
132                                     if (splitendpoint[0] ==
"https:")
133                                     {
134                                         keySSLEnabled="true";
135                                     }
136                                     if (securityConfig == null) {
137                                         securityConfig = "";
138                                     }
139                                     var burl = splitendpoint[2];
140                                     var appId = splitendpoint[3];
```

```
141                                     pd = appId+userName+password;
142                                     sap.Settings.store = new
sap.EncryptedStorage("SettingsStore", pd);
143                                     connectionData = {
144
"keyMAFLogonOperationContextConnectionData": {
145
"keyMAFLogonConnectionDataApplicationSettings":
146                                                 {
147
"DeviceType":device.platform,
148
"DeviceModel":device.model,
149
"ApplicationConnectionId":applicationConnectionId
150                                                 },
151
"keyMAFLogonConnectionDataBaseURL":burl
152                                             },
153
"keyMAFLogonOperationContextApplicationId":appId,
154
"keyMAFLogonOperationContextBackendUserName":userName,
155
"keyMAFLogonOperationContextBackendPassword":password,
156
"keyMAFLogonOperationContextSecurityConfig":securityConfig,
157
"keySSLEnabled":keySSLEnabled
158                                     };
159
sap.Settings.start(connectionData,
160                                                         function(mesg)
{
161
sap.Settings.isInitialized = true;
162
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
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.
```

```
187         * @param {string} [errorCallback] Error callback to call
when registration attempt fails.

188         * @param {string} [notificationlistenerfunc] The function
that receives the notification for processing by the application.

189          * @param {string} [senderId] The sender ID that is used
for GCM registration. For other platforms it is null.

190          * @private

191          * @memberof sap.Push

192          * @function registerForNotificationTypes

193          * @example

194          * regid = "211112269206";

195          * function registerSuccess(mesg){}

196          * function registerFailure(mesg) {}

197          * function ProcessNotification(mesg){}

198          *
sap.Push.registerForNotificationTypes(sap.Push.notificationType.bad
ge | sap.Push.notificationType.sound |
sap.Push.notificationType.alert, registerSuccess, registerFailure,
ProcessNotification, regid);

199          */

200

201     registerForNotification: function (types, successCallback,
errorCallback, notificationListenerFunc, senderId ) {

202         if(device.platform == "iPhone" || device.platform ==
"iPad" || device.platform == "iPod touch" || device.platform == "iOS"
|| device.platform == "Android") {

203             sap.Push.RegisterSuccess = successCallback;

204             sap.Push.RegisterFailed  = errorCallback;

205             sap.Push.ProcessNotificationForUser =
notificationListenerFunc;

206             sap.Push.call_native(sap.Push.pushregsuccsss,
"registerForNotificationTypes", [types, senderId]);

207

208         }

209

210     },

211
```

```
212         /* Core APIS */
213
214         /**
215          * Function called by the application to register
notification types to receive.
216          *
217          * @param {string} types Types of notifications the
application wants to receive. The different types of notifications
are expressed in <code>notificationType</code>
218        *               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).
219          * @param {string} successCallback Success callback to
call when registration is successful.
220          * @param {string} errorCallback Error callback to call
when registration attempt fails.
221         * @param {string} notificationlistenerfunc The function
that receives the notification for processing by the application.
222          * @param {string} [senderId] The sender ID that is used
for GCM registration. For other platforms it is null.
223          * @public
224          * @memberof sap.Push
225          * @function registerForNotificationTypes
226          * @example
227          * regid = "211112269206";
228          * function registerSuccess(mesg){}
229          * function registerFailure(mesg) {}
230          * function ProcessNotification(mesg){}
231          *
sap.Push.registerForNotificationTypes(sap.Push.notificationType.bad
ge | sap.Push.notificationType.sound |
sap.Push.notificationType.alert, registerSuccess, registerFailure,
ProcessNotification, regid);
232          */
233     registerForNotificationTypes: function (types,
successCallback, errorCallback, notificationListenerFunc, senderId )
{
234         sap.Push.getConnectionSettings(function(){
```

```
235                                         sap.Logger.debug("getting
Connection Settings","PUSHJS",function(m){},function(m){});

236                                     console.log("getting Connection
Settings");

237
sap.Push.registerForNotification(types,  successCallback,
errorCallback, notificationListenerFunc, senderId );

238                                          },

239                                          function(){});

240     },

241

242

243         /**

244          * Function called by the application to unregister from
future notifications.

245          *

246          * @param {function} callback 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.

247          * @public

248          * @memberof sap.Push

249          * @function unregisterForNotificationTypes

250          * @example

251          * function unregCallback(mesg){}

252          *
sap.Push.unregisterForNotificationTypes(unregCallback);

253          */

254

255     unregisterForNotificationTypes: function (callbak) {

256         if(device.platform == "iPhone" || device.platform ==
"iPad" || device.platform == "iPod touch" || device.platform == "iOS"
|| device.platform == "Android") {

257
sap.Push.call_native(callbak,"unregisterForNotification");

258         }

259     },
```

```
260
261         /**
262         * 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.
263          *
264          * @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 into the
example for the deail on how to use them.
265          * @public
266          * @memberof sap.Push
267          * @function getBadgeNumber
268          * @example
269          * function getBadgeNumCallback(data) { badgecount =
data["badgecount"];}
270          * sap.Push.getBadgeNumber(getBadgeNumCallback);
271          */
272     getBadgeNumber: function(callback)
273         {
274             if (sap.Push.isPlatformIOS()) {
275                 sap.Push.call_native(callback,
"getBadgeNumber");
276             }
277         },
278
279         /**
280         * 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.
281          *
282         * @param {number} number The badge count to set for the
application.
283          * @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
purpose.
```

```
284          * @public
285          * @memberof sap.Push
286          * @function setBadgeNumber
287          * @example
288          * function badgeCallback(mesg){}
289          * badgenum = 10;
290          * sap.Push.setBadgeNumber(badgenum, badgeCallback);
291          */
292     setBadgeNumber: function (number, callback) {
293         if (sap.Push.isPlatformIOS()) {
294             sap.Push.call_native(callback, "setBadgeNumber",
[number]);
295         }
296     },
297
298         /**
299         * 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.
300          *
301          * @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.
302          * @public
303          * @memberof sap.Push
304          * @function resetBadge
305          * @example
306          * function badgeCallback(mesg){}
307          * sap.Push.resetBadge(badgeCallback);
308          */
309     resetBadge: function (callback) {
310         if (sap.Push.isPlatformIOS()) {
311             sap.Push.call_native(callback, "resetBadge");
```

```
312         }
313     },
314
315
316
317
318
319         /**
320         * This method updates the application with the new device
token in the SAP Mobile Platform server.
321          *
322         * @param {string} [devtok] The device token received from
the APNS/GCM device registration.
323          * @public
324         * @callback {function} [callback] The callback function
that is called with the registration result.
325          * @memberof sap.Push
326          * @example
327          * function callback(mesg) {}
328          * devToken ="123123213213";//sample device token
329          * sap.Push.updateWithDeviceToken(devToken, callback);
330          */
331
332     updateWithDeviceToken:  function (devtok, callback) {
333         if (sap.Push.isPlatformIOS() || device.platform ==
"Android" ) {
334             sap.Push.call_native(callback,
"updateWithDeviceToken", [devtok]);
335         }
336     },
337
338         /**
339         * This method checks for any notifications received while
the application was not running in the foreground. Application
developer can call this
```

```
340         * 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.

341          * @param {function} callback 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.

342          * @memberof sap.Push

343          * @example

344          * function processBackgroudMessage(mesg){

345          *

346          * }

347          * function checkBackgroundNotification() {

348          *
sap.Push.checkForNotification(processBackgroudMessage);

349          * }

350          * document.addEventListener("onSapLogonSuccess",
checkBackgroundNotification, false);

351          * document.addEventListener("onSapResumeSuccess",
checkBackgroundNotification, false);

352          **/

353

354     checkForNotification: function(callback) {

355         if (sap.Push.isPlatformIOS() || device.platform ==
"Android" ) {

356             sap.Push.call_native(callback,
"checkForNotification");

357         }

358     },

359

360         /**

361        * This is an internal function, which is called when there
is a push notification.

362          * @private

363          **/

364     ProcessNotification: function(message) {

365         if (sap.Push.ProcessNotificationForUser == null )
```

```
366         {
367             console.log("No Processing function provided");
368             sap.Logger.debug("Notification listener function is
not registered. Register it by calling
registerForNotificationTypes","PUSHJS",function(m){},function(m)
{});
369         } else {
370             sap.Push.ProcessNotificationForUser(message);
371         }
372     },
373         /**
374          * This is an internal function, which is automatically
called when the plugin is initialized. Used only for android.
375          * @private
376          **/
377     initPlugin: function(callback) {
378         if ( device.platform == "Android")
379         {
380             args = [];
381             exec(
382                 callback,
383                 function(){ sap.Logger.debug("Plugin
Initialization","PUSHJS",function(m){},function(m){}); } ,
384                 'SMPPushPlugin',
385                 "initPlugin",
386                 args);
387         }
388     }
389
390     };
391
392
393     /**
394      * Local private variables
```

```
395      */
396     module.exports.RegisterSuccess = null;
397     module.exports.RegisterFailed = null;
398     module.exports.ProcessNotificationForUser = null;
399     /**
400      * Enum for types of push notification.
401      * @enum {number}
402      * @private
403      */
404     module.exports.notificationType = {
405         /** Disable all notifications */
406     NONE: 0,
407         /** Set badge count on app icon */
408     BADGE: 1,
409         /** Play sounds on receiving notification */
410     SOUNDS: 2,
411         /** Show alert on receiving notification */
412     ALERT: 4
413     };
414
415
416
417
418     document.addEventListener('deviceready',
module.exports.initPlugin, false);
419
420
```

Using the 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 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).

Deleting of Encrypted Storage for Security Reasons

The EncryptedStorage plugin receives a notification from the Login plugin in the event that the Login plugin's data vault is deleted. This can occur when the user forgets their password while unlocking the application, violates a password policy set on the server, or explictly deletes the registration. The EncryptedStorage plugin then generates an `OnEncryptedStorageErased` event which is a notification that the encrypted storage on the device (the database the application uses for secure storage of application data) has been cleared for security reasons.

Domain Whitelisting

Kapsel plugins support Apache Cordova's domain whitelisting model. Whitelisting allows you to control access to external network resources. Apache Cordova whitelisting allows you to whitelist individual network resources (URLs), for example, http://www.google.com.

For information about the whitelist rules, see *http://docs.phonegap.com/en/3.3.0/guide_appdev_whitelist_index.md.html*.

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 280	This constant is the length of the largest string that is guaranteed to be successfully stored on Android.
ERROR_BAD_PASSWORD on page 280	This error code indicates that the operation failed due to an incorrect password.
ERROR_GREATER_THAN_MAXIMUM_SIZE on page 281	This error indicates that the string was too large to store.
ERROR_INVALID_PARAMETER on page 281	This error code indicates an invalid parameter was provided.
ERROR_UNKNOWN on page 281	This error code indicates an unknown error occurred.
SIMPLE_STRING_MAXIMUM_LENGTH on page 281	This constant is the length of the largest string that can successfully be stored on Android.

Methods

Name	Description
clear(successCallback, errorCallback) on page 282	This function removes all items from the store.
deleteStore(successCallback, errorCallback) on page 282	This function deletes a store that has been created with EncryptedStorage.
getItem(key, successCallback, errorCallback) on page 283	This function gets the value corresponding to the given key.
key(index, successCallback, errorCallback) on page 284	This function gets the key corresponding to the given index.
length(successCallback, errorCallback) on page 285	This function gets the length of the store.
removeItem(key, successCallback, errorCallback) on page 285	This function removes the item corresponding to the given key.
setItem(key, value, successCallback, errorCallback) on page 286	This function sets an item with the given key and value.

Type Definitions

Name	Description
errorCallback(errorCode) on page 287	Callback function that is invoked in case of an error.
getItemSuccessCallback(value) on page 288	
keySuccessCallback(key) on page 289	
lengthSuccessCallback(length) on page 289	
successCallback on page 290	Callback function that is invoked on a successful call to a function that does not need to return anything.

Source
encryptedstorage.js, line 37 on page 292.

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 337 on page 303.

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 309 on page 302.

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 317 on page 302.

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 301 on page 301.

ERROR_UNKNOWN member
This error code indicates an unknown error occurred.

Syntax
<constant> `ERROR_UNKNOWN`

Source
encryptedstorage.js, line 294 on page 301.

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 325 on page 302.

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 290	If successful, the successCallback is invoked with no parameters.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 287	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 228 on page 299.

deleteStore(successCallback, errorCallback) method
This function deletes a store that has been created with EncryptedStorage.

This allows for the creation of a new store with the same name and a different password.

Syntax
<static> `deleteStore`(*successCallback*, *errorCallback*)

Parameters

Name	Type	Description

successCallback	*sap.EncryptedStorage~successCallback* on page 290	If successful, the successCallback is invoked with no parameters.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 287	If there is an error, the errorCallback is invoked with an ErrorInfo object as the parameter.

Example

```
var successCallback = function() {
   alert("Store deleted!");
}
var errorCallback = function(error) {
   alert("An error occurred: " + JSON.stringify();
}
ks = new sap.EncryptedStorage("storename", "password");
ks.deleteStore(successCallback, errorCallback);
```

Source

encryptedstorage.js, line 260 on page 300.

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 288	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 287	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 117 on page 294.

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 289	If successful, the successCallback is invoked with the key as the parameter.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 287	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 78 on page 293.

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 289	If successful, the successCallback is invoked with the length of the store as the parameter.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 287	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 45 on page 292.

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 290	If successful, the successCallback is invoked with no parameters.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 287	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 194 on page 298.

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 281 and *sap.EncryptedStorage#COMPLEX_STRING_MAXIMUM_LENGTH* on page 280 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 290	If successful, the successCallback is invoked with no parameters.
errorCallback	*sap.EncryptedStorage~errorCallback* on page 287	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 153 on page 296.

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 281, *sap.EncryptedStorage#ERROR_INVALID_PARAMETER* on page 281, *sap.EncryptedStorage#ERROR_BAD_PASSWORD* on page 280, or *sap.EncryptedStorage#ERROR_GREATER_THAN_MAXIMUM_SIZE* on page 281.

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 359 on page 304.

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 357 on page 304.

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 355 on page 303.

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 353 on page 303.

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 351 on page 303.

Source code

encryptedstorage.js

```
1       var argscheck = require('cordova/argscheck'),

2           exec = require("cordova/exec");

3

4       /**

5        * The EncryptedStorage class is used as a secure local
store.  The EncryptedStorage API is based on the

6        * W3C web storage API, but has two major differences: it is
asynchronous, and it has a constructor with

7        * a password.<br/>

8        * <br/>

9       * Note: There is a security flaw on some versions of Android
with the Pseudo Random Number Generation.

10       * The first time the native code of this plugin runs it
applies the fix for this issue.  However, the

11      * fix needs to be applied before any use of Java Cryptography
Architecture primitives.  Therefore, it

12      * is a good idea to run this plugin (call a function that has
a native component: length, key, getItem,

13       * setItem, removeItem, clear) before using any other
security-related plugin, to protect yourself
```

```
14       * against the possibility that the other plugin does not
apply this fix. No other Kapsel plugins are

15      * affected, so you need not do this on their behalf.  For more
details about the security flaw, see

16       * <a href="http://android-developers.blogspot.com/2013/08/
some-securerandom-thoughts.html">

17       * http://android-developers.blogspot.com/2013/08/some-
securerandom-thoughts.html</a><br/>

18       * <br/>

19      * <b>Adding and Removing the EncryptedStorage Plugin</b><br/
>

20       * The EncryptedStorage plugin is added and removed using
the

21       * <a href="http://cordova.apache.org/docs/en/edge/
guide_cli_index.md.html#The%20Command-line%20Interface">Cordova
CLI</a>.<br/>

22       * <br/>

23      * To add the EncryptedStorage plugin to your project, use the
following command:<br/>

24       * cordova plugin add <path to directory containing Kapsel
plugins>\encryptedstorage<br/>

25       * <br/>

26       * To remove the EncryptedStorage plugin from your project,
use the following command:<br/>

27       * cordova plugin rm
com.sap.mp.cordova.plugins.encryptedstorage

28       * @namespace

29       * @alias EncryptedStorage

30       * @memberof sap

31      * @param {String} storeName The name of the store to create.
All stores with different names

32       * act independently, while stores with the same name (and
same password) act as the same store.

33       * If null or undefined is passed, an empty string is used.

34       * @param {String} password The password of the store to
create.  If null or undefined is passed,

35       * an empty string is used.

36       */
```

```
37      EncryptedStorage = function (storeName, password) {
38          // private variables
39          var that = this;
40          var storagePassword = password ? password : "";
41          var storageName = storeName ? storeName : "";
42
43          // privileged functions
44
45          /**
46          * This function gets the length of the store.  The length
of a store
47           * is the number of key/value pairs that are in the
store.
48           * @param {sap.EncryptedStorage~lengthSuccessCallback}
successCallback If successful,
49           * the successCallback is invoked with the length of the
store as
50           * the parameter.
51           * @param {sap.EncryptedStorage~errorCallback}
errorCallback If there is an error,
52          * the errorCallback is invoked with an ErrorInfo object as
the parameter.
53           * @memberof sap.EncryptedStorage
54           * @function length
55           * @instance
56           * @example
57           * var store = new sap.EncryptedStorage("storeName",
"storePassword");
58           * var successCallback = function(length) {
59           *    alert("Length is " + length);
60           * }
61           * var errorCallback = function(error) {
62           *    alert("An error occurred: " +
JSON.stringify(error));
63           * }
```

```
64            * store.length(successCallback, errorCallback);
65            */
66           this.length = function (successCallback, errorCallback)
{
67               try{
68                   argscheck.checkArgs('FF',
'EncryptedStorage.length', arguments);
69               }catch(ex){
70                   errorCallback(this.ERROR_INVALID_PARAMETER);
71                   return;
72               }
73
74               cordova.exec(successCallback, errorCallback,
"EncryptedStorage",
75                   "length", [storageName, storagePassword]);
76           }
77
78           /**
79            * This function gets the key corresponding to the given
index.
80           * @param {number} index The index of the store for which
to get the key.
81          * Valid indices are integers from zero (the first index),
up to, but not including,
82          * the length of the store.  If the index is out of bounds,
then the success
83            * callback is invoked with null as the parameter.
84            * @param {sap.EncryptedStorage~keySuccessCallback}
successCallback If successful,
85            * the successCallback is invoked with the key as the
parameter.
86            * @param {sap.EncryptedStorage~errorCallback}
errorCallback If there is an error,
87          * the errorCallback is invoked with an ErrorInfo object as
the parameter.
88            * @memberof sap.EncryptedStorage
89            * @function key
```

```
90           * @instance
91           * @example
92          * // This example shows how to get the key for the last
item.
93           * var store = new sap.EncryptedStorage("storeName",
"storePassword");
94           * var errorCallback = function( error ){
95           *    alert("An error occurred: " +
JSON.stringify(error));
96           * }
97           * var keySuccessCallback = function(key) {
98           *    alert("Last key is " + key);
99           * }
100          * var lengthSuccessCallback = function(length) {
101          *    store.key(length - 1, keySuccessCallback,
errorCallback);
102          * }
103          * store.length(lengthSuccessCallback, errorCallback);
104          */
105         this.key = function (index, successCallback,
errorCallback) {
106             try{
107               argscheck.checkArgs('NFF', 'EncryptedStorage.key',
arguments);
108             }catch(ex){
109                errorCallback(this.ERROR_INVALID_PARAMETER);
110                return;
111             }
112
113             cordova.exec(successCallback, errorCallback,
"EncryptedStorage",
114                 "key", [storageName, storagePassword, index]);
115         }
116
117         /**
```

```
118         * This function gets the value corresponding to the given
key.  If there is no

119          * item with the given key, then the success callback is
invoked with null as

120          * the parameter.

121         * @param {String} key The key of the item for which to get
the value.  If null or undefined is

122          * passed, "null" is used.

123          * @param {sap.EncryptedStorage~getItemSuccessCallback}
successCallback If successful,

124          * the successCallback is invoked with the value as the
parameter (or null if the key

125          * did not exist).

126          * @param {sap.EncryptedStorage~errorCallback}
errorCallback If there is an error,

127          * the errorCallback is invoked with an ErrorInfo object
as the parameter.

128          * @memberof sap.EncryptedStorage

129          * @function getItem

130          * @instance

131          * @example

132          * var store = new sap.EncryptedStorage("storeName",
"storePassword");

133          * var successCallback = function(value) {

134          *    alert("Value is " + value);

135          * }

136          * var errorCallback = function(error) {

137          *    alert("An error occurred: " +
JSON.stringify(error));

138          * }

139          * store.getItem("theKey", successCallback,
errorCallback);

140          */

141         this.getItem = function (key, successCallback,
errorCallback) {

142             try{
```

```
143                 argscheck.checkArgs('SFF',
'EncryptedStorage.getItem', arguments);
144             }catch(ex){
145                 errorCallback(this.ERROR_INVALID_PARAMETER);
146                 return;
147             }
148
149             cordova.exec(successCallback, errorCallback,
"EncryptedStorage",
150                 "getItem", [storageName, storagePassword,
key]);
151         }
152
153         /**
154          * This function sets an item with the given key and
value.  If no item exists with
155         * the given key, then a new item is created.  If an item
does exist with the
156          * the given key, then its value is overwritten with the
given value.<br/>
157          * <br/>
158         * Note: On Android there is a size limit on the string to
be stored.  See
159          * {@link
sap.EncryptedStorage#SIMPLE_STRING_MAXIMUM_LENGTH} and {@link
sap.EncryptedStorage#COMPLEX_STRING_MAXIMUM_LENGTH}
160          * for more details.
161         * @param {String} key The key of the item to set.  If null
or undefined is passed,
162          * "null" is used.
163         * @param {String} value The value of the item to set.  If
null or undefined is passed,
164          * "null" is used.
165          * @param {sap.EncryptedStorage~successCallback}
successCallback If successful,
166          * the successCallback is invoked with no parameters.
```

```
167          * @param {sap.EncryptedStorage~errorCallback}
errorCallback If there is an error,
168         * the errorCallback is invoked with an ErrorInfo object
as the parameter.
169          * @memberof sap.EncryptedStorage
170          * @function setItem
171          * @instance
172          * @example
173          * var store = new sap.EncryptedStorage("storeName",
"storePassword");
174          * var successCallback = function() {
175          *    alert("Item has been set.");
176          * }
177          * var errorCallback = function(error) {
178          *    alert("An error occurred: " +
JSON.stringify(error));
179          * }
180        * store.setItem("somekey", "somevalue", successCallback,
errorCallback);
181          */
182         this.setItem = function (key, value, successCallback,
errorCallback) {
183             try{
184                 argscheck.checkArgs('SSFF',
'EncryptedStorage.setItem', arguments);
185             }catch(ex){
186                 errorCallback(this.ERROR_INVALID_PARAMETER);
187                 return;
188             }
189
190             cordova.exec(successCallback, errorCallback,
"EncryptedStorage",
191                 "setItem", [storageName, storagePassword, key,
value]);
192         }
193
```

```
194         /**
195          * This function removes the item corresponding to the
given key.  If there is no
196          * item with the given key in the first place, that is
still counted as a success.
197         * @param {String} key The key of the item to remove.  If
null or undefined is
198          * passed, "null" is used.
199          * @param {sap.EncryptedStorage~successCallback}
successCallback If successful,
200          * the successCallback is invoked with no parameters.
201          * @param {sap.EncryptedStorage~errorCallback}
errorCallback If there is an error,
202         * the errorCallback is invoked with an ErrorInfo object
as the parameter.
203          * @memberof sap.EncryptedStorage
204          * @function removeItem
205          * @instance
206          * @example
207          * var store = new sap.EncryptedStorage("storeName",
"storePassword");
208          * var successCallback = function() {
209          *    alert("Value removed");
210          * }
211          * var errorCallback = function(error) {
212          *    alert("An error occurred: " +
JSON.stringify(error));
213          * }
214          * store.removeItem("somekey", successCallback,
errorCallback);
215          */
216         this.removeItem = function (key, successCallback,
errorCallback) {
217             try{
218                 argscheck.checkArgs('SFF',
'EncryptedStorage.removeItem', arguments);
219             }catch(ex){
```

```
220                 errorCallback(this.ERROR_INVALID_PARAMETER);
221                 return;
222             }
223
224             cordova.exec(successCallback, errorCallback,
"EncryptedStorage",
225                 "removeItem", [storageName, storagePassword,
key]);
226         }
227
228         /**
229          * This function removes all items from the store.  If
there are no
230          * items in the store in the first place, that is still
counted as a success.
231          * @param {sap.EncryptedStorage~successCallback}
successCallback If successful,
232          * the successCallback is invoked with no parameters.
233          * @param {sap.EncryptedStorage~errorCallback}
errorCallback If there is an error,
234          * the errorCallback is invoked with an ErrorInfo object
as the parameter.
235          * @memberof sap.EncryptedStorage
236          * @function clear
237          * @instance
238          * @example
239          * var store = new sap.EncryptedStorage("storeName",
"storePassword");
240          * var successCallback = function() {
241          *    alert("Store cleared!");
242          * }
243          * var errorCallback = function(error) {
244          *    alert("An error occurred: " + JSON.stringify();
245          * }
246          * store.clear(successCallback, errorCallback);
```

```
247          */
248         this.clear = function (successCallback, errorCallback)
{
249             try{
250                 argscheck.checkArgs('FF',
'EncryptedStorage.clear', arguments);
251             }catch(ex){
252                 errorCallback(this.ERROR_INVALID_PARAMETER);
253                 return;
254             }
255
256             cordova.exec(successCallback, errorCallback,
"EncryptedStorage",
257                 "clear", [storageName, storagePassword]);
258         }
259
260         /**
261         * This function deletes a store that has been created with
EncryptedStorage.
262          * This allows for the creation of a new store with the
same name and a different password.
263          *
264          * @param {sap.EncryptedStorage~successCallback}
successCallback If successful,
265          * the successCallback is invoked with no parameters.
266          * @param {sap.EncryptedStorage~errorCallback}
errorCallback If there is an error,
267         * the errorCallback is invoked with an ErrorInfo object
as the parameter.
268          * @memberof sap.EncryptedStorage
269          * @function deleteStore
270          * @example
271          * var successCallback = function() {
272          *    alert("Store deleted!");
273          * }
```

```
274          * var errorCallback = function(error) {
275          *    alert("An error occurred: " + JSON.stringify();
276          * }
277          * ks = new sap.EncryptedStorage("storename",
"password");
278          * ks.deleteStore(successCallback, errorCallback);
279          */
280         this.deleteStore = function (successCallback,
errorCallback) {
281             try{
282                 argscheck.checkArgs('FF',
'EncryptedStorage.deleteStore', arguments);
283             }catch(ex){
284                 errorCallback(this.ERROR_INVALID_PARAMETER);
285                 return;
286             }
287
288             cordova.exec(successCallback, errorCallback,
"EncryptedStorage",
289                 "deleteStore", [storageName, storagePassword]);
290         }
291     };
292
293     // Error codes
294     /**
295      * This error code indicates an unknown error occurred.
296      * @memberof sap.EncryptedStorage
297      * @name sap.EncryptedStorage#ERROR_UNKNOWN
298      * @constant
299      */
300     EncryptedStorage.prototype.ERROR_UNKNOWN = 0;
301     /**
302      * This error code indicates an invalid parameter was
provided.
```

```
303      * (eg: a string given where a number was required).
304      * @memberof sap.EncryptedStorage
305      * @name sap.EncryptedStorage#ERROR_INVALID_PARAMETER
306      * @constant
307      */
308     EncryptedStorage.prototype.ERROR_INVALID_PARAMETER = 1;
309     /**
310      * This error code indicates that the operation failed due to
an incorrect password.  The password is
311      * set by the constructor of {@link EncryptedStorage}.
312      * @memberof sap.EncryptedStorage
313      * @name sap.EncryptedStorage#ERROR_BAD_PASSWORD
314      * @constant
315      */
316     EncryptedStorage.prototype.ERROR_BAD_PASSWORD = 2;
317     /**
318      * This error indicates that the string was too large to
store. Only applies to Android.
319      * For iOS, no hard limit is imposed, but be aware of device
memory constraints.
320      * @memberof sap.EncryptedStorage
321      * @name
sap.EncryptedStorage#ERROR_GREATER_THAN_MAXIMUM_SIZE
322      * @constant
323      */
324    EncryptedStorage.prototype.ERROR_GREATER_THAN_MAXIMUM_SIZE =
3;
325     /**
326      * This constant is the length of the largest string that can
successfully be stored on Android.  Only if all the
327      * characters in the string are encoded in 1 byte in UTF-8 can
a string actually be this big.  Since
328      * characters in UTF-8 can take up to 4 bytes, if you do not
know the contents of a string, the maximum
329      * length that is guaranteed to be successful is {@link
EncryptedStorage.COMPLEX_STRING_MAXIMUM_LENGTH}, which is
```

```
330      * {@link EncryptedStorage.SIMPLE_STRING_MAXIMUM_LENGTH}/4.
Note that this size restriction is present only on
331       * Android and not iOS.
332       * @memberof sap.EncryptedStorage
333       * @name sap.EncryptedStorage#SIMPLE_STRING_MAXIMUM_LENGTH
334       * @constant
335       */
336     EncryptedStorage.prototype.SIMPLE_STRING_MAXIMUM_LENGTH =
1048527;
337     /**
338      * This constant is the length of the largest string that is
guaranteed to be successfully stored on Android.  The
339      * limit depends on how many bytes the string takes up when
encoded with UTF-8 (under which encoding
340      * characters can take up to 4 bytes).  This is the maximum
length of a string for which every character
341      * takes all 4 bytes.  Note that this size restriction is
present only on Android and not iOS.
342       * @memberof sap.EncryptedStorage
343       * @name sap.EncryptedStorage#COMPLEX_STRING_MAXIMUM_LENGTH
344       * @constant
345       */
346     EncryptedStorage.prototype.COMPLEX_STRING_MAXIMUM_LENGTH =
262131;
347
348     module.exports = EncryptedStorage;
349
350
351     /**
352     * Callback function that is invoked on a successful call to a
function that does
353       * not need to return anything.
354       *
355       * @callback sap.EncryptedStorage~successCallback
356       */
```

```
357
358     /**
359      * Callback function that is invoked on a successful call to
{@link EncryptedStorage.length}.
360      *
361      * @callback sap.EncryptedStorage~lengthSuccessCallback
362      *
363     * @param {number} length The number of key/value pairs in the
store.
364      */
365
366     /**
367      * Callback function that is invoked on a successful call to
{@link EncryptedStorage.key}.
368      * If the key returned is null that means the index passed to
{@link EncryptedStorage.key} was out of bounds.
369      *
370      * @callback sap.EncryptedStorage~keySuccessCallback
371      *
372      * @param {String} key The key corresponding to the given
index.  Will be null if the index passed to
373      * {@link EncryptedStorage.key} was out of bounds.
374      */
375
376     /**
377      * Callback function that is invoked on a successful call to
{@link EncryptedStorage.getItem}.
378     * If the returned value is null, that means the key passed to
{@link EncryptedStorage.getItem} did not exist.
379      *
380      * @callback sap.EncryptedStorage~getItemSuccessCallback
381      *
382     * @param {String} value The value of the item with the given
key.  Will be null if the key passed to
383      * {@link EncryptedStorage.getItem} did not exist.
```

```
384      */
385
386     /**
387      * Callback function that is invoked in case of an error.
388      *
389      * @callback sap.EncryptedStorage~errorCallback
390      *
391      * @param {number} errorCode An error code indicating what
went wrong.  Will be one of {@link
sap.EncryptedStorage#ERROR_UNKNOWN},
392      * {@link sap.EncryptedStorage#ERROR_INVALID_PARAMETER},
{@link sap.EncryptedStorage#ERROR_BAD_PASSWORD}, or
393      * {@link
sap.EncryptedStorage#ERROR_GREATER_THAN_MAXIMUM_SIZE}.
394      *
395      * @example
396      * function errorCallback(errCode) {
397     *    //Set the default error message. Used if an invalid code
is passed to the
398      *    //function (just in case) but also to cover the
399      *    //sap.EncryptedStorage.ERROR_UNKNOWN case as well.
400      *    var msg = "Unkown Error";
401      *    switch (errCode) {
402      *       case sap.EncryptedStorage.ERROR_INVALID_PARAMETER:
403      *          msg = "Invalid parameter passed to method";
404      *          break;
405      *       case sap.EncryptedStorage.ERROR_BAD_PASSWORD :
406      *          msg = "Incorrect password";
407      *          break;
408      *       case
sap.EncryptedStorage.ERROR_GREATER_THAN_MAXIMUM_SIZE:
409      *         msg = "Item (string) value too large to write to
store";
410      *          break;
411      *    };
```

```
412       *    //Write the error to the log
413       *    console.error(msg);
414       *    //Let the user know what happened
415       *    navigator.notification.alert(msg, null,
"EncryptedStorage Error", "OK");
416       * };
417       */
418
```

Using the 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.

Domain Whitelisting

Kapsel plugins support Apache Cordova's domain whitelisting model. Whitelisting allows you to control access to external network resources. Apache Cordova whitelisting allows you to whitelist individual network resources (URLs), for example, http://www.google.com.

For information about the whitelist rules, see *http://docs.phonegap.com/en/3.3.0/guide_appdev_whitelist_index.md.html*.

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 308	Starts the settings exchange.

Source

settings.js, line 12 on page 309.

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 95 on page 312.

Source code

settings.js

```
1       // 3.0.2-SNAPSHOT
2       var exec = require("cordova/exec");
3
4
5       /**
6        * Provides settings exchange functionality
7        *
8        * @namespace
9        * @alias Settings
10       * @memberof sap
11       */
12      var SettingsExchange = function () {};
13
14      SettingsExchange.prototype.connectionData   = null;
15      SettingsExchange.prototype.store            = null;
16      SettingsExchange.prototype.settingsSuccess  = null;
17      SettingsExchange.prototype.SettingsError    = null;
18      SettingsExchange.prototype.isInitialized    = false;
19
20
21      /**
22      * Starts the settings exchange process upon onSapLogonSuccess
event.
23       * @private
```

```
24       */
25      var doSettingExchange = function () {
26
27
28          sap.Settings.isInitialized = true;
29          var pd ="";
30          sap.Logon.unlock(function (connectionInfo) {
31                              var userName =
connectionInfo["registrationContext"]["user"];
32                              var password  =
connectionInfo["registrationContext"]["password"];
33                              var applicationConnectionId =
connectionInfo["applicationConnectionId"];
34                              var securityConfig =
connectionInfo["registrationContext"]["securityConfig"];
35                              var endpoint =
connectionInfo["applicationEndpointURL"];
36                              var keySSLEnabled = "false";
37                              var splitendpoint =
endpoint.split("/");
38                              if (splitendpoint[0] == "https:")
39                              {
40                                  keySSLEnabled="true";
41                              }
42                              if (securityConfig == null) {
43                                  securityConfig = "";
44                              }
45                              var burl = splitendpoint[2];
46                              var appId = splitendpoint[3];
47                              pd = appId+userName+password;
48                              connectionData = {
49
"keyMAFLogonOperationContextConnectionData": {
50
"keyMAFLogonConnectionDataApplicationSettings":
```

```
51                                      {
52
"DeviceType":device.platform,
53
"DeviceModel":device.model,
54
"ApplicationConnectionId":applicationConnectionId
55                                      },
56
"keyMAFLogonConnectionDataBaseURL":burl
57                                  },
58
"keyMAFLogonOperationContextApplicationId":appId,
59
"keyMAFLogonOperationContextBackendUserName":userName,
60
"keyMAFLogonOperationContextBackendPassword":password,
61
"keyMAFLogonOperationContextSecurityConfig":securityConfig,
62                                  "keySSLEnabled":keySSLEnabled
63                              };
64                              sap.Settings.start(connectionData,
65                                                      function(mesg) {
66
sap.Settings.isInitialized = true;
67
sap.Logger.debug("Setting Exchange is successful
"+mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
68                                                      },
69                                                      function(mesg){
70
sap.Logger.debug("Setting Exchange failed" +
mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
71
sap.Settings.isInitialized = false;
72                                                      });
73                              }
74                              , function () {
```

```
75                              sap.Logger.debug("unlock failed
","SMP_SETTINGS_JS",function(m){},function(m){});

76                              }

77                              );

78

79

80      };

81

82      document.addEventListener("onSapLogonSuccess",
doSettingExchange, false);

83      document.addEventListener("onSapResumeSuccess",
doSettingExchange, false);

84

85      SettingsExchange.prototype.reset = function(key, sucessCB,
errorCB)

86      {

87          if ((typeof(sap.Settings.store) != undefined) &&
(sap.Settings.store != null)) {

88              sap.Settings.store.removeItem(key, sucessCB,
errorCB);

89          } else {

90              errorCB("Cannot access setting store");

91          }

92      }

93

94

95      /**

96       * Starts the settings exchange.

97       * @public

98       * @memberof sap.Settings

99       * @method start

100     * @param {String} connectionData This example below shows the
structure of the connection data.

101     * @param {function} successCallback Function to invoke if the
exchange is successful.
```

```
102      * @param {function} errorCallback Function to invoke if the
exchange failed.
103      * @example
104      * connectionData = {
105      *      "keyMAFLogonOperationContextConnectionData": {
106      *      "keyMAFLogonConnectionDataApplicationSettings":
107      *      {
108      *      "DeviceType":device.platform,
109      *      "DeviceModel":device.model,
110      *      "ApplicationConnectionId":"yourappconnectionid"
111      *      },
112      *
"keyMAFLogonConnectionDataBaseURL":"servername:port"
113      *  },
114      *
"keyMAFLogonOperationContextApplicationId":"yourapplicationid",
115      *
"keyMAFLogonOperationContextBackendUserName":"yourusername",
116      *
"keyMAFLogonOperationContextBackendPassword":"password",
117      *
"keyMAFLogonOperationContextSecurityConfig":"securityConfigName",
118      *  "keySSLEnabled":keySSLEnabled
119      *  };
120      * sap.Settings.start(connectionData, function(mesg) {
121      *
122      *
sap.Logger.debug("Setting Exchange is successful
"+mesg,"SMP_SETTINGS_JS",function(m){},function(m){});
123      *                                          },
124      *                                        function(mesg){
125      *                                    sap.Logger.debug("Setting
Exchange failed" + mesg,"SMP_SETTINGS_JS",function(m){},function(m)
{});
126      *                                        });
127      */
```

```
128    SettingsExchange.prototype.start = function (connectionData,
successCallback, errorCallback) {

129         sap.Settings.settingsSuccess = successCallback;

130         sap.Settings.SettingsError = errorCallback;

131         sap.Settings.connectionData = connectionData;

132         sap.Logger.debug("Accessing the data from
vault","SMP_SETTINGS_JS",function(m){},function(m){});

133
sap.logon.Core.getSecureStoreObject( sap.Settings.getStoreDataSucce
ss, sap.Settings.getStoreDataError, "settingsdata");

134

135

136     };

137

138

139

140

141

142

143     /**

144      *  This is a private function. End user will not use this
plugin directly.

145     *  This function gets called after the start function is able
to read the current settings from the secured storage.

146      *  @private

147      *  @param {String} value This is the value of the current
setting exchange stored in the secured store.

148      **/

149

150     SettingsExchange.prototype.getStoreDataSuccess  =
function(value){

151         storedSettings = value;

152         sap.Logger.debug("Exchanging the
data","SMP_SETTINGS_JS",function(m){},function(m){});

153         exec(sap.Settings.SettingsExchangeDone,

154              sap.Settings.SettingsExchangeError,
```

```
155              "SMPSettingsExchangePlugin",
156              "start",
[JSON.stringify(connectionData),storedSettings]);
157     }
158
159     /**
160      *  This is a private function. End user will not use this
plugin directly.
161     *  This function is called after the start function is unable
to read the current settings from the secured storage.
162      *  @private
163     *  @param {String} message This is the error message produced
by the encrypted storage.
164      **/
165     SettingsExchange.prototype.getStoreDataError  =
function(mesage){
166         sap.Logger.debug("Setting exchange failed to read data
store: Proceeding without data",function(m){},function(m){});
167     }
168
169
170     /**
171      *  This is a private function. End user will not use this
plugin directly.
172      *  This function is called after the settings exchange
completes succefully.
173      *  @private
174     *  @param {String} message This is the  message produced when
the settings plugin completes successfully.
175      **/
176
177     SettingsExchange.prototype.SettingsExchangeDone =
function(message) {
178         sap.Logger.debug("Setting Exchange
Success","SMP_SETTINGS_JS",function(m){},function(m){});
179         var jsondata =  JSON.parse(message);
180         settingsString = JSON.stringify(jsondata["data"]);
```

```
181 sap.logon.Core.setSecureStoreObject(sap.Settings.SettingsWriteDone,
sap.Settings.SettingsWriteError,"settingsdata",settingsString);
182         if (sap.Settings.settingsSuccess != null) {
183             sap.Logger.debug("Setting exchange
successful","SMP_SETTINGS_JS",function(m){},function(m){});
184             sap.Settings.settingsSuccess(jsondata["msg"]);
185         }
186     }
187
188     /**
189      *  This is a private function. End user will not use this
plugin directly.
190      *  This function is called after the settings exchange
completes succefully
191      *  @private
192     *  @param {String} message This is the error message produced
when the settings plugin has an error.
193      **/
194     SettingsExchange.prototype.SettingsExchangeError =
function(message) {
195         sap.Logger.error("Setting Exchange failed calling the
error callback funciton","SMP_SETTINGS_JS",function(m){},function(m)
{});
196         if (sap.Settings.SettingsError != null) {
197             sap.Settings.SettingsError(message);
198         }
199     }
200
201     /**
202      *  This is a private function. End user will not use this
plugin directly.
203      *  This function is called after the setting data is stored
successfully.
204      *  @private
205     *  @param {String} message This is the message produced upon
successful storing of settings to the encrypted store.
```

```
206      **/
207     SettingsExchange.prototype.SettingsWriteDone  =
function(message) {
208          sap.Logger.debug("Setting
stored","SMP_SETTINGS_JS",function(m){},function(m){});
209
210     }
211
212     /**
213      *  This is a private function. End user will not use this
plugin directly.
214      *  This function is called after the storing of the setting
data fails.
215      *  @private
216      *  @param {String} message This is the message produced upon
failure to store the settings to the encrypted store.
217      **/
218     SettingsExchange.prototype.SettingsWriteError  =
function(message) {
219          sap.Logger.error("Setting store
failed","SMP_SETTINGS_JS",function(m){},function(m){});
220     }
221
222     /**
223      *  This is a private function. End user will not use this
plugin directly.
224      *  This function is called after the deviceready. This
uploads the logs to the server.
225      *  @private
226      *  @param {boolean} uploadLog This indicates whether the
upload log is currently enabled or disbled.
227      **/
228     SettingsExchange.prototype.logLevelUpdated  =
function(logLevel)
229     {
230          sap.Logger.setLogLevel(logLevel,
sap.Settings.LogLevelSetSuccess, sap.Settings.LogLevelSetFailed);
```

```
231         sap.Logger.upload(sap.Settings.logUploadedSuccess,
sap.Settings.logUploadFailed);
232     }
233
234     /**
235      *  This is a private function. End user will not use this
plugin directly.
236      *  This function is called when the log upload succeeds.
237      *  @private
238      *  @param {mesg} logupload message
239      **/
240     SettingsExchange.prototype.LogLevelSetSuccess =
function(mesg){
241         sap.Logger.debug("Log level set
successful","SMP_SETTINGS_JS",function(m){},function(m){});
242     }
243     /**
244      *  This is a private function. End user will not use this
plugin directly.
245      *  This function is called when the log upload succeeds.
246      *  @private
247      *  @param {mesg} logupload message
248      **/
249     SettingsExchange.prototype.LogLevelSetFailed =
function(mesg){
250         sap.Logger.error("Log level set
failed","SMP_SETTINGS_JS",function(m){},function(m){});
251     }
252
253     /**
254      *  This is a private function. End user will not use this
plugin directly.
255      *  This function is called when the log upload succeeds.
256      *  @private
257      *  @param {mesg} logupload message
```

```
258      **/
259     SettingsExchange.prototype.logUploadedSuccess =
function(mesg){
260          sap.Logger.debug("Log upload
successful","SMP_SETTINGS_JS",function(m){},function(m){});
261     }
262     /**
263      *  This is a private function. End user will not use this
plugin directly.
264      *  This function is called when the log upload fails.
265      *  @private
266      *  @param {mesg} logupload failure message
267      **/
268     SettingsExchange.prototype.logUploadFailed = function(mesg)
{
269          sap.Logger.error("upload log
failed","SMP_SETTINGS_JS",function(m){},function(m){});
270
271     }
272
273
274
275     module.exports = new SettingsExchange();
276
277
278
279
280
```

Developing a Kapsel Application With OData Online

Create an OData application, register the connection, and retrieve the application connection settings.

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. This is the application or bundled identifier that the application developer assigns or generates during application development. The administrator uses the Application ID to register the application to SAP Mobile Platform Server, and the client application code uses the Application ID while sending requests to the server.
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 desktop 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

If the selected hybrid app uses the **AppUpdate** plugin, activate the new version from this screen. If the hybrid app does not use the **AppUpdate** plugin, the application-specific settings are not applicable.

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. (Optional) If you imported a new application, skip this step. If you are updating the version, click **Browse** to upload another version of the application.
   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.

Property	Description
Description	Describes the Kapsel app.
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 has made changes, **AppUpdate** detects them using contents in the `www` folder (that is, the HTML based content), and not with native plugins or changes made outside of that folder. For changes made outside the `www` folder, the developer needs to post a new copy of the app to the application download site, or use Afaria to push the new app to all users.
4. Remove application version that have been imported, but not yet deployed:
   a. Click **Remove** and confirm to remove an application from one mobile platform.
   b. Click **Remove All** and confirm to remove 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 uses 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

A

C

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

W