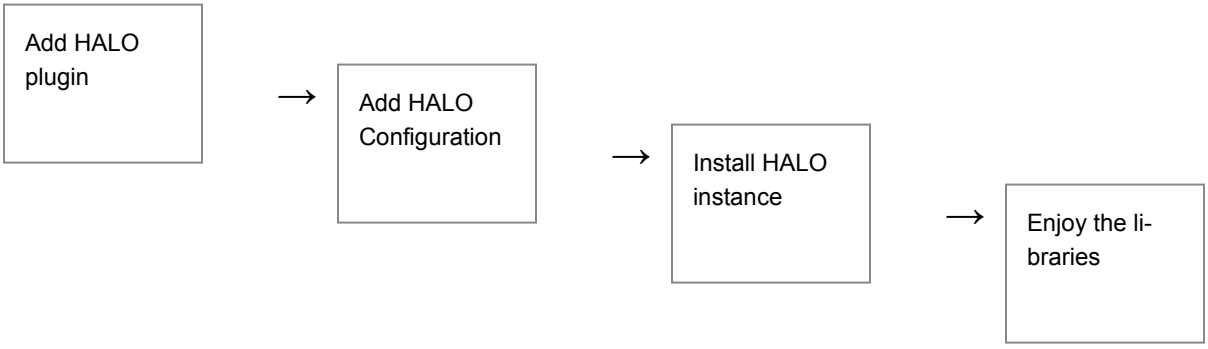
HALO SDK - Android

Version 2.0

Last generated: November 15, 2016

Table of Contents

Android SDK - Getting Started



This getting started guide will guide you on setting up HALO SDK for Android in a few minutes. We will provide a step by step guide to get everything working with the most basic setup, for more detailed information about specific calls or how a module works check the sidebar.

Step 1: Add the HALO plugin

Open the build.gradle of the project root and add the plugin to the classpath:

```
buildscript {
    dependencies {
          classpath 'com.mobgen.halo.android:halo-plugin:{versi
on}'
    }
}
```

Now apply the halo plugin to your HALO application after the android one:

```
apply plugin: 'halo'
```

Step 2: Add HALO configuration

Open the build.gradle of your app and apply the basic configuration based on your HALO project. Here you have the minimal configuration you will need. Put it after the Android node.

```
halo {
    clientId "YOUR_HALO_KEY"
    clientSecret "YOUR_HALO_SECRET"
}
```

Step 3: Init the Halo instance inside your app

Option 1: I don't have a custom Application class

If you **don't** have a custom application class, follow this instructions. Open your AndroidManifest.xml and apply the following configuration to your Application node:

```
 <application
        ...
        android:name="com.mobgen.halo.android.sdk.api.HaloAppli
cation">
        ...
</application>
```

Note: This will create the `Halo` instance for you. You can always access this instance by calling `HaloApplication.halo()`

Option 2: I do have a custom application class

In this case you can either install HALO by yourself or extend the `HaloApplication` class provided and override some of the methods that come with it.

Adding to your custom application class:

```
public class MyCustomApplication extends AnotherApplicationClas
s {
    @Override
    public void onCreate(){
        super.onCreate();
        Halo.installer(this).install();
    }
}
```

Extending the `HaloApplication` class:

```
public class MyCustomApplication extends HaloApplication {
    public Halo.Installer onCreateInstaller(){}
    public Halo.Installer beforeInstallHalo(@NonNull Halo.Insta
ller installer){}
    public Halo onHaloCreated(@NonNull Halo halo){}
}
```

Step 4: Start using HALO in your app

Use any of the HALO APIs or plugins without worrying when is it ready, since this is managed internally by the different libraries.

Android SDK - Installer Options

In the install step of HALO there are many parameters that can be configured. Here is the list of items and what are their purposes:

- **config**: provides the configuration for the HALO framework so you can add some aditional behaviour or configuration on it.
- **credentials**: sets the credentials client id and secret. They are added with gradle by default.
- **debug**: sets the debug flag to see what is going behind the scenes in HALO. See also the debug recipe (page 23).
- **endProcess**: adds a startup process that will be run during the installation process in HALO.
- **addTagCollector**: adds a tag collector that allows you to add some tags in the moment the app is initialized.
- **enableDefaultTags**: adds the halo default tags into the tag collectors. This was enabled by default some time ago but now it is optional.
- **environment**: sets the environment that will be used for the calls of halo.

Android SDK - Getting Started With Gradle Plugin

Here we provide the step by step guide to add the plugin to your project and start using it with the minimal configuration.

1. Apply the classpath to the buildscript

Open the build.gradle of the project root and add the plugin to the classpath:

```
buildscript {
    dependencies {
          classpath 'com.mobgen.halo.android:halo-plugin:{plugi
nVersion}'
    }
}
```

2. Apply the plugin

Now apply the configuration plugin to your HALO managed application project after applying the android application plugin.

```
apply plugin: 'halo'
```

Note: You can also use the name ‘com.mobgen.halo.android’ for the plugin if you want to.

Tip: The plugin can only be applied to android application projects, don’t apply it to any android library project.

3. Add the HALO configuration

Open the build.gradle of your app and apply the basic configuration based on your HALO project. This is the minimal configuration you have to add to the plugin. If you add less information than that the plugin will not allow to compile.

```
halo {
    clientId 'YOUR_HALO_KEY'
    clientSecret 'YOUR_HALO_SECRET'
}
```

Android SDK - Gradle Plugin Options

Gradle plugin options

What does the plugin do?

To avoid lines and lines of configuration we decided to create a gradle plugin to make your life as a developer easier. It handles the dependencies and the magic numbers you need to access HALO.

In your project you need to use Android Studio, or at least, a gradle build system. With that in mind we will describe how to put this plugin in your application project and the possible options to customize the SDK we are offering.

Options

In the HALO configuration inside the build.gradle there are many options you can configure to keep the SDK configured as you need. Here you have the reference for each property:

```
halo {
    clientId // Client id of the application created in HALO. S
tring:Required.
    clientSecret // Client secret of the application created i
n HALO. String:Required.
    clientIdDebug // Client id of the application created in HA
LO for testing purposes, this will be used when the debug flag
is set in the installer. String:Optional.
    clientSecretDebug: Client secret of the application create
d in HALO for testing purposes, this will be used when the debu
g flag is set in the installer. String:Optional.
    services { // Allows you to add more services from HALO. *C
losure:Optional*.
        content // You will enable the content library. This li
brary helps when retrieving content. Boolean:Optional.
        notifications // Enables Google FCM integration with HA
LO used to receive push notifications. Remember to add also th
e google-services.json to your project. Boolean:Optional.
        translations: Enables the translations library for HAL
O. Boolean:Optional.
        presenter: Enables the presenter UI library for HALO. I
t is a small helper to use the MVP pattern in the UI. Boolean:O
ptional.
    }
    androidVariants { // In the case you want different configu
rations for different variants, you can enable it with this con
figuration. Optional:Closure
        variantName { //You can put here the same configuratio
n with services and clientId/clientSecret as you do in the glob
al config. Both kind of configurations cannot be mixed.
            //Here it goes the same config with clientId, clien
tSecret and services for the current variant.
        }
    }
}
```

Full example copy/paste script

Here you can find a HALO plugin configuration you can put and fill in your project:

```
apply plugin: 'com.android.application'
apply plugin: 'halo'

android {
    ...
}

halo {
    clientId "halotestappclient"
    clientSecret "halotestapppass"
    clientIdDebug "halotestappclient"
    clientSecretDebug "halotestapppass"
    services {
        content true
        notifications true
    }
}
```

Here the same configuration, but with notifications enabled only for release builds.

```
apply plugin: 'com.android.application'
apply plugin: 'halo'

android {
    ...
}

halo {
    androidVariants {
        debug {
            clientId "TestDebug"
            clientSecret "TestDebug"
            services {
                content true
            }
        }

        release {
            clientId "TestRelease"
            clientSecret "TestRelease"
            services {
                content true
                notifications true
            }
        }
    }
}
```

Android SDK - HALO Core API

You will be able to access some internals of HALO by accessing to its `HaloCore` instance. This instance keeps the current device with its segmentation tags, the credentials, token and configuration provided in the plugin.

You can always access the core from the instance but be careful when modifying anything, since most of it is automatically assigned and you typically don't want to change the internals:

Method name	Functionality
`sessionManager()`	provides the session manager bucket where the HALO session is stored.
`debug()`	returns the current debug configuration.
`credentials()`	returns the current credentials stored in the core.
`credentials(credentials)`	sets the current credentials and reauthenticates the APIs.
`logout()`	if you are logged in with some credentials you will be logged out from those user credentials.
`version()`	provides the library version which is the version of all the items in the sdk.
`flushSession()`	removes the current session token, so another one will be requested when any request is done with Halo.
`framework()`	provides the instance of the HALO framework used internally to manage all the calls.
`manager()`	provides access to the `HaloManagerApi` which contains all the operations you can do to manage the internal state of the HALO instance.

Method name	Functionality
`device()`	provides the current stored device. If HALO is not fully installed the device will be an anonymous one not ready yet. This cannot be null.
`pushSenderId()`	provides the configured sender id for the push notifications. If it is null, the push notifications are not enabled.
`notificationsToken()`	provides the token from the Firebase cloud messaging service if the notifications library is enabled.
`notificationsToken(token)`	sets the FCM token. Just for internal use.
`segmentationTags()`	collects all the segmentation tags for the current device.
`serverVersionCheck()`	tells if the current sdk is outdated based on the server version. 1 means valid, 2 means outdated, 3 means not checked (ie, no internet).
`isVersionValid()`	checks the serverVersionCheck to ensure wether this version is valid or not. Valid means it has not been checked for some reason (typically network problems) or it is outdated.

Android SDK - HALO Manager API

API definition

The manager plugin is in charge to all the management actions that can be done in the core, such as changing the tags of the device or requesting the modules that belongs to the current application.

In this guide you will find an explanation of the api available methods. To create an instance for the manager API you should use: `java Halo.core().manager();`

Method name	Functionality
storage	Provides the current storage api
getModules	Provides the modules that belongs to the current application. You can decide if this request supports offline and also in which thread this request should be done
getServerVersion	Provides the server version for the current sdk. This tells if the sdk is outdated or not
requestToken	Requests a new authentication token
isAppAuthentication	Tells if the current authentication is based in the app credentials
isPasswordAuthentication	Tells if the current authentication is based in the password credentials
syncDevice	Syncs the current device with the device in HALO
subscribeForDeviceSync	Subscribes for the device update
syncDeviceWhenNetworkAvailable	Synchronizes the device stored with the one in the server once the network is available. In this case callback is not allowed

Method name	Functionality
sendDevice	Updates the device that is present in the core sending it to the server
getCurrentDevice	Provides the current cached device
addDeviceTag	Adds a new tag to the device to segment information and syncs it
addDeviceTags	Adds multiple tags to the device and syncs it
removeDeviceTag	Removes a tag from the device and syncs it
removeDeviceTags	Removes multiple tags

Example: request the modules

Here is a full example on how to request the modules for the current app:

```
halo.core().manager()
    .getModules(Data.NETWORK_AND_STORAGE)
    .asContent()
    .threadPolicy(Threading.POOL_QUEUE_POLICY)
    .execute(new CallbackV2<List<HaloModule>>() {
        @Override
        public void onFinish(@NonNull HaloResultV2<List<HaloMod
ule>> result) {
                        //Manage the result
                }
        });
```

Example: Add a device tag

Here is the full example to add a tag to the current device and sync this device with HALO:

```
HaloSegmentationTag tagToAdd = new HaloSegmentationTag("myNewTa
gName","myNewTagValue");
halo.core().manager()
    .addDeviceTag(tagToAdd)
    .threadPolicy(Threading.POOL_QUEUE_POLICY)
    .execute(new CallbackV2<Device>() {
                @Override
            public void onFinish(@NonNull HaloResultV2<Device>
result) {
                        //Do something with the new device
                }
        });
```

Android SDK - Cloudinary Media Helper

HALO uses the Cloudinary service to upload media files. We have added to our SDK a helper with the filtering options. Check the javadoc documentation to learn more about this helper.

Here we are providing an example to blur an image provided:

```
HaloCloudinary.builder()
                .addEffects(HaloCloudinary.Effects.blur(100))
                .build("http://cloudinary.com/..."); //Cloudina
ry image url
```

Android SDK - Custom Middleware Requests

It is likely that you want to create a custom module and add its functionalities to HALO. You can do it using HALO by adding his module endpoint into the administration console. Use the help on the CMS to make sure you configure it properly.

When a new module is added, it generates urls in the following way:

```
https://halo.mobgen.com/api/---proxy---/-/
```

Keep in mind that the endpoint depends on the environment configured when Halo is installed and the default environment is production.

Using the SDK you can make custom requests to your endpoint to get the data, already being authenticated using the HALO authentication system. The available API to do so includes using the `HaloMiddlewareRequest` class. If, for example, we would like to make a request to our custom module with:

1. The company name “mobgen”
2. The Module type “cake”
3. The final url being `cakes/{id}`
4. Is under a proxy

The final url result would be:
`https://halo.mobgen.com/api/---proxy---/mobgen-cake/cakes/{id}`

Where id is the id of the cake you want to get. In the following pieze of code you will get an example of how to bring this data from the web service. Remember you can only make this request if HALO is already installed.

```
Map<String, String> params = new HashMap<String,String>(1);
params.put("id", myId);
HaloMiddlewareRequest.builder(halo.framework().network().clien
t())
                .method(HaloRequestMethod.GET)
                .module("mobgen", "cake")
                .hasProxy(true)
                .url("cakes/{id}", params)
                .callback(callback)
                .build().execute();
```

The response that comes from this web service call is an okHttp3 `Response` object, so you can rely on their documentation to process it.

Android SDK - Offline support

The SDK supports an offline database that caches the responses for many requests and provides some cached data when the device is offline or cannot stablish a connection.

Most of the requests support a flag to tell the request how it should be synchronized with the server and which kind of that should provide. This flags belong to the `Data` class and can take the following values:

- **NETWORK_AND_STORAGE**: Will make the request and cache the response, providing this cached response. If the device has no internet connection will provide the previously cached response, or nothing if there is no data inside.
- **NETWORK_ONLY**: Will make the request to the network and provide the same data received bypassing the cache.
- **STORAGE_ONLY**: Will get the cached data no matter if there is internet connection or not in the device.

The result of a request using the sync engine comes in a `HaloResultV2` which contains two methods: * **status()**: Provides the status of the data. It allows you to know if there was any error while performing the operation and the status of the thata (wether it is fresh, local or inconsistent). * **data()**: Provides the data brought. It can be null if there was an error or no data is cached while in offline mode. You must always nullcheck it.

The callback for an operation comes only with one method which is `onFinish` and receives a `HaloResultV2` with the status and the data.

Example

```
halo.core().manager().getModules(Data.NETWORK_AND_STORAGE)
        .asContent()
        .threading(Threading.POOL_QUEUE_POLICY)
        .execute(new CallbackV2<List<HaloModule>>() {
          @Override
          public void onFinish(@NonNull HaloResultV2<List<HaloR
emoteModule>> result) {
              if(result.status().isSuccess()){
                  showData(result)
              }else{
                  showError(result)
              }
          }
        });
```

Android SDK - Change Server Environment

Maybe you don’t want to use HALO in the default production environment (https://halo.mobgen.com (https://halo.mobgen.com)). To change the environment in which HALO is working on, you have to customize your installation process. See the example above:

```
Halo.installer(context)
        .environment("https://halo-int.mobgen.com")
        .install();
```

Android SDK - Enable Debug Mode

HALO supports a debugging mode that allows you to see many information of what is going on behind the scenes. To enable this mode you have to add it in the installer instance.

```
Halo.installer(context)
        .debug(true)
        .install();
```

This change will:

1. Enable the logging for HALO.
2. Take the client id debug and the client secret debug configuration.
3. Take the debug id to enable the push notifications if it is available.

Android SDK - Content SDK Overview

Download 2.0.0 (https://bintray.com/halo-mobgen/maven/HALO-Content/_latestVersion)

Add dependency

In the HALO plugin add the following to enable the content sdk.

```
apply plugin: 'halo'

halo {
        ...
        services {
                content true
        }
        ...
}
```

Content API methods

The content API is the facade for the Content HALO SDK. Importing this library will need a valid HALO instance configured with some credentials. The HALO Content SDK allows the user to retrieve instances from the HALO Backend in two main ways:

- Search
- Sync

☑ **Tip:** Use the search method to get conent when you want to get some concrete elements, segmented data or certain information.

☑ **Tip:** Use the sync method if you prefer to download the whole module to use if offline. It is better for performance than search.

Creating an instance of the Content API is really simple once you have your HALO running. Just write the following line:

```
HaloContentApi contentApi = HaloContentApi.with(halo);
```

In addition you can provide a default locale to include it in all the queries even if you didn't provide it in the search/sync.

Search

If you want to bring certain data based on some criteria or search some of them in the HALO Backed, you have to use the search operation in the API. To do that we provide a class called `SearchQuery` with a simple fluent API that allows you to specify the criterias for the search. Once selected, you can perform the search and cache the result in local for offline use. See the example above:

```
SearchQuery query = SearchQuery.builder()
        .moduleIds("myModuleId")
        .beginSearch()
                .eq("name", "Sample")
        .end()
        .build();
contentApi.search(Data.NETWORK_ONLY, query)
        .asContent()
        .execute(callback);
```

This search will request all the instances for the module id "myModuleId" and which body contains a name with the value "Sample". Check out the rest of the available options in the detailed documentation (page 27).

Sync

The sync operation is thought for performance critical tasks. It allows to synchronize a full module consuming the less amount of data possible.

Syncing involves three steps:

- Listening for sync updates.
- Requesting a sync for a module.
- Requesting synced local instances.

Check out the following example for a full sync lifecycle:

Listen for sync updates

```
//Start listening for updates
ISubscription subscription = contentApi.subscribeToSync("my mod
ule name", listener);
//When you are done or to free memory
subscription.unsubscribe();
```

Request a sync for a module

```
//Create the sync query
SyncQuery query = SyncQuery.create("my module name", Threadin
g.POOL_QUEUE_POLICY);
contentApi.sync(query);
```

Request the synced instances

```
//Request the synced content
contentApi.getSyncInstances("my module name")
        .asContent(MyCustomClass.class)
        .execute(callback);
```

Note: Make sure your MyCustomClass.class is properly annotated with LoganSquare `@JsonObject` annotation to make it work properly, otherwise the result will not be parsed. You can check it in content parsing section (page 34).

If you want to go in deep into this module, please refer to the detailed documentation (page 27).

Android SDK - Content SDK Detailed APIs

Here you can find fine grained explanations for every public param of the content SDK. The rest of the library is obfuscated over proguard and only intended methods are public and properly named although the code is (and will be) Open Source.

Search

With the search query you can request some instances from the HALO Backend based on some query parameters. See the Search query section for all the available params.

```
HaloContentApi api = HaloContentApi.with(halo);
api.search(Data.NETWORK_AND_STORAGE, query)
        .asContent()
        .execute(callback);
```

Search Query

The `SearchQuery` object supports many params to help in the search task. To create a new instance of the `SearchQuery` you have to use the `Builder` pattern by calling `SearchQuery.builder()` . The build object is parcelable and so you can send it across activities if needed.

Here you can find the full list of options you can chain into the `SearchQuery` :

Search param	Explanation
moduleName	requests a module by its name. You must owe it and it must be available for client applications.
moduleIds	the list of module ids to request.
instanceIds	the list of instance ids to request.

Search param	Explanation
pickFields	filters the values returned from the api, so it will not send more information than the needed by the application.
segmentMode	specifies how the segmentation should work against the tags. It can take two values: `PARTIAL_MATCH` or `TOTAL_MATCH`. Partial match checks if there is at least one tag in the content provided while total match ensures the content retrieved contains all the tags provided.
tags	segmentation tags to apply to this content. This param is related to the segmentMode one since it selects and segments the content based on those tags.
setDevice	applies all the tags from the current device and matches the segment mode to `PARTIAL_MATCH` if no mode was set. Refer to the segmentMode param.
segmentWithDevice	applies automatically the current device in the core to the search specified.
populateAll	if there are fields with relations, they are populated.
populate	a list of the fields that should be populated.
beginSearch/end	syntax declaration to make custom queries inside the content info. For example, if we have instances in halo with two fields, name and amount, we would be able to filter the searched instances based in both values.
beginMetaSearch/end	it has the same concept as the search but allows the user to filter based on metadata of the instance, such as the updated time or the instance name.
searchTag	tag name that will be used as an id for offline caching. This allows to override previous search data tagging them. It is useful for searches that include timestamps, since those searches would generate much more content than the needed for offline.

Search param	Explanation
locale	the locale specified for localized fields. If no locale is provided an object with all the locales will be provided for the given field.
ttl	time that the content should remain available offline.
pagination	indicates which page and which limit should be requested to get the instances.
onePage	allows to make a request with a single page. It is equivalent to make the request without pagination but provides the information as if you did it in a single page. The priority of this param is higher than the pagination one.

In the beginSearch/end and beginMetaSearch/end parameters there are many query parameters supported. Here you have an index on how to use them and an example.

Search condition	Explanation
and	Adds a condition to make both expression work together with an and condition.
or	Adds a contition to make both expressions work together with an or condition.
in	Check if the field has the values provided inside it.
nin	Ensures the field has not the values inside.
eq	Checks fields that are equals to the given value.
neq	Checks the fields are not equals to the given value.
lt	Checks the value is less than the provided value.
lte	Checks the value is less than or equals the provided value.
gt	Checks the value is greater than the provided value.

Search condition	Explanation
gte	Checks the value is greater than or equals the provided value.
beginGroup	Begins a parenthesis group.
endGroup	Ends a parenthesis group.

Custom search sample

```
SearchQuery.builder()
    .beginSearch()
        .in("name", listOfNames)
        .and()
        .beginGroup()
            .nin("name", bannedNames)
            .and()
            .eq("active", "Activated")
        .endGroup()
    .end()
    .build();
```

> If the expression built for the search is not correct it will throw a Runtime exception.

Data provider

The search supports 3 modes to select the source where the content comes from:

- **Data.NETWORK_ONLY**: this would be the simplest one. Just does the request and provides you the result.
- **Data.NETWORK_AND_STORAGE**: in this case the data is brought from network, stored in the local storage and retrieved to the user.
- **Data.STORAGE_ONLY**: this option provides only the cached data for the given request.

Data parsing

When you call the `api.search` method you are not actually doing the request, it provides you an object that can be further configured to bring the data in the format you expect. In this case you have 3 different options:

- `asRaw()` : provides a cursor you can parse by your own. Usually this will not be used unless you need some sort of performance critical task in a list.
- `asContent()` : provides a `HaloContentInstance` list. This can be useful if you need to check also the metadata. To parse it to a custom class you can use the following operation on each instance:

```
HaloContentHelper.from(instance, MyCustomClass.class, halo.fram
ework().parser());
```

- `asContent(Class<T> clazz)` : this is the typical configuration you will need. Just pass your custom class to the content parameter and it will parse the list for you directly from the json received. Also the class must be annotated with `@JsonObject` and the fields with `@JsonField` . Refer to LoganSquare documentation (https://github.com/bluelinelabs/LoganSquare) for more details.

> ⚠ **Important:** Remember that to make the class available for parsing you need to use the correct annotations. See the content parsing section (page 34).

Sync

When a given module has so many items that handling them takes too much time or you want to have some content available in the background for the user, synchronization can make that work for you.

The process of the synchronization is the following:

- Subscribe to the synchronization hub to ensure when the synchronization process is done.

```
HaloContentApi contentApi = HaloContentApi.with(halo);
ISubscription subscription = contentApi.subscribeToSync(moduleN
ame, new HaloSyncListener() {
    @Override
    public void onSyncFinished(@NonNull HaloStatus status, @Nul
lable HaloSyncLog log) {
            if(log != null){
                Log.i("Sync", log.toString());
        } else {
                Log.e("Sync", "Error " + status);
        }
    }
});
```

- Then you can request an internal module name to be synced:

```
SyncQuery query = SyncQuery.create("my module name", Threadin
g.POOL_QUEUE_POLICY);
contentApi.sync(query);
```

- You can then get the instances of the synced module even if you are offline:

```
contentApi.getSyncInstances("my module name")
        .asContent(MySampleClass.class)
        .execute(callback);
```

- Remember to unsubscribe from the `Subscription` created when you are done, this will avoid possible memory leaks:

```
subscription.unsubscribe();
```

- For debugging purposes, we keep a log for all the synchronizations that are done by an application. You can take them by module or all using the following call:

```
contentApi.getSyncLog("my module name")
        .asContent()
        .execute(callback);
```

- Finally you can remove your stored data for a given module using:

```
contentApi.clearSyncInstances("my module name")
        .asContent()
        .execute(callback);
```

The result provided by all the callbacks is a HaloResult as many other calls, so you can always take advantage of the state of the data and the possible errors that can appear, whether they are from the database or network.

Execution options

This api supports some execution options based on the sdk. Checkout them to see how productive you can be.

Threading

Almost every request supports the threading more. This mode allows you to select which is the threading context in which the request will be executed. Lets say you are already in another thread and you don't want to spawn another one, with this param you can specify this behavior. There are 3 modes supported:

- **Threading.POOL_QUEUE_POLICY**: spawns a new thread into a thread pool that will be executed as soon as possible.
- **Threading.SINGLE_QUEUE_POLICY**: spawns a new thread into a thread queue that will execute it once the other threads enqueued free the queue.
- **Threading.SAME_THREAD_POLICY**: does not spawn a thread, it uses the same context as it was called so everything will be executed synchronously.

Content parsing

We use LoganSquare for performance reasons inside our sdk. It allows us to parse really fast and without too much methods (avoiding the method count limit). Here we are providing a sample of how to annotate some content to parse it properly. Refer to LoganSquare documentation (https://github.com/bluelinelabs/LoganSquare) for more details.

Mapping sample

```
@JsonObject
public class Article {

    @JsonField(name = "Title")
    public String mTitle;

    @JsonField(name = "Date")
    public Date mDate;

    @JsonField(name = "ContentHtml")
    public String mArticle;

    @JsonField(name = "Summary")
    public String mSummary;

    @JsonField(name = "Thumbnail")
    public String mThumbnail;

    @JsonField(name = "Image")
    public String mImage;
}
```

Android SDK - Notifications SDK Overview

Download 2.0.0 (https://bintray.com/halo-mobgen/maven/HALO-Notifications/_latestVersion)

In the HALO plugin add the following to enable the notifications sdk.

```
apply plugin: 'halo'

halo {
        ...
        services {
                notifications true
        }
        ...
}
```

You also need to add the google play services json you download from the firebase console from google (`google-services.json`). Add it in a proper place to make this work.

Notifications API

To start using the notifications, in the same way you do with other sdks you must create a new instance using the already configured instance of HALO.

```
HaloNotificationsApi notificationsApi = HaloNotificationsApi.wi
th(halo);
```

Also you have to release the memory in the `onTerminate` method of your application:

```
notificationsApi.release();
```

Simple use

Listen all the notifications that arrive to the device by attaching a listener to the api.

```
ISubscription subscription = notificationsApi.listenAllNotifica
tions(listener);
```

Then you will receive a call on this callback everytime a notification is received. This will contain the bundle with data provided by the notifications service and a bundle with extra data if it is provided.

Once you are done with this listener you can cancel the subscription by calling `unsubscribe()`.

```
subscription.unsubscribe();
```

You can also customize your notifications by adding a notification decorator. You will receive as parameter the `NotificationCompat.Builder` already configured by HALO, so you can override its behavior. If a decorator returns `null` instead the `NotificationCompat.Builder` configured, the notification will not be displayed. Refer to the detailed documentation (page 0) to learn how to write your custom decorator.

```
notificationsApi.setNotificationDecorator(notificationDecorato
r);
```

Android SDK - Notifications SDK Detailed APIs

Enable notifications

The notification system is built in in the SDK based on the FCM framework (Firebase Cloud Messages), which is the default system to receive notifications in the Android OS.

> ❗ **Warning:** Since FCM uses the Google Play Services framework, HALO cannot receive notifications in those mobile phones that does not support the Google Play Store with Play services.

Based on the HALO backend you can send many data in the notifications and segment depending on the segmentation of the users present in the system. In this guide we will show you how to enable the notifications on the SDK and how to handle some custom actions on those notifications.

Step 1. Configure a FCM project

First of all we need to create a FCM project in the Firebase Console (https://console.developers.google.com). If you already have an FCM project configured you can go directly to 3. If you already have an app with the given package configured go to 6.

1. Click on ‘Create New Project’ button.
2. Fill in the project name and the country/region.
3. Enter the project and click on ‘Add app’.
4. Click on ‘Add Firebase to your Android App’.
5. Fill in your package name of the app and the debug certificate SHA-1.
6. Click on the app menu you want receive the push notifications and select ‘Manage’.
7. There select cloud messaging.
8. Write down the server key. It looks like ‘AIzaSyAtN64Y0******-*********’

Configure FCM

Step 2. Add the Server key to HALO

Take the Server API Key obtained in the previous step and put it in the administration console of HALO.:

1. Do login in HALO as an admin.
2. Go to your apps.
3. Select the application you are enabling the push notifications to.
4. Update the app filling the **Android key** field with your server key.

Step 3. Enable the notifications in the SDK

To enable the notifications inside your app you have to add the HALO plugin (page 0). Once done, in your **HALO properties closure** enable the notifications service. Take the following code of a build.gradle as an example.

```
halo {
    ...
    services {
        notifications true
    }
    ...
}
```

Step 4. Add Google Play Configuration File

Take the file obtained in the Step 1 of this tutorial and place it in the root of your application module or in your flavor folder named as “*google-services.json*”.

React to a notification

In HALO we support two different notification types. **Normal notifications** and **silent notifications**. Normal notifications includes the UI that shows the user a notification was received, while silent notifications only notify you in a callback to perform some background work.

With this silent notifications the SDK will not display any UI but also will not perform any action. As a developer you have to put an entry point so we can send you the information received. In this guide we tell you how to do it. Follow the instructions below to add a listener.

1. Create the listener

This instance will receive the notifications for those notifications that will be listened with the `HaloNotificationListener`. Here you can check an example:

```
public class NotificationReceiver implements HaloNotificationLi
stener {

    @Override
    public void onNotificationReceived(@NonNull Context contex
t, @NonNull String from, @NonNull Bundle data, @Nullable Bundl
e extra){
        //Do something with this data
    }
}
```

2. Attach this listener to HALO

You can attach the listener to listen **not silent**, **silent** or **all** notifications:

```
notificationsApi.listenAllNotifications(new NotificationReceive
r()); // All
notificationsApi.listenNotSilentNotifications(new NotificationR
eceiver()); //Not silent
notificationsApi.listenSilentNotifications(new NotificationRece
iver()); //Silent
```

Get UI notification id

When a notification is displayed in the UI, this notifications has an unique id assigned so you can perform some actions on it, like cancelling. You can access to this id in the data bundle provided in the notification callback. There is a helper method in the api to do so:

```
Integer notificationId = notificationsApi.getNotificationId(bun
dle);
```

Keep in mind this method returns null if there is no notification id attcached to the notification. This means there is no UI displayed linked to the received notification.

Customize notification displayed

Behind the scenes, when the server sends the notification to the Google Firebase Services, it can send information in two ways:

- **Providing a notification payload**: This generates a notification automatically and you cannot handle this behaviour.
- **Providing a data payload**: The data payload allows you to handle whatever action you need.

In Android, HALO always handles the notifications using the `NotificationService` and the data payload.

Since we create the notifications manually, we have designed every single functional addition to the notifications as a decorator pattern (https://en.wikipedia.org/wiki/Decorator_pattern). The base decorator class is `HaloNotificationDecorator` which is extended by many classes in the SDK that finally are chained to add the behaviour to the Android `NotificationCompat.Builder`.

To modify something in the notifications, we provide a method so you can handle or add behaviour to this notification just by extending the `HaloNotificationDecorator` class. Here you have the example guide to put an icon on the notification.

If you return `null` in your custom decorator the notification will not be displayed.

1. Create custom decorator

We are providing a custom implementation of the icon decorator:

```
public class CustomIconDecorator extends HaloNotificationDecora
tor {

    public NotificationIconDecorator(){
        super();
    }

    @Override
    public NotificationCompat.Builder decorate (@NonNull Notifi
cationCompat.Builder builder, @NonNull Bundle bundle) {
        builder.setSmallIcon(R.drawable.myNotificationIcon);
        //You can return also null if you want this notificatio
n not to appear
        return builder;
    }
}
```

2. Add decorator to HALO

If you want to add this decorator to the notification service you can do it while in the install process of HALO:

```
notificationsApi.setNotificationDecorator(new CustomIconDecorat
or());
```

As it is a Decorator, you can always chain as many decorators as you want by calling the method `NotificationDecorator#chain(builder, bundle)` .

This also can be used for example to execute an intent when the user clicks on the notification, and you are allowed to chain multiple decorators in the following way:

```
new PendingIntentDecorator(context, new CustomIconDecorator());
```

Example of a decorator to add notification behavior

```
public class PendingIntentDecorator extends HaloNotificationDec
orator {

    private Context mContext;

    public CustomNotificationDecorator(Context context, HaloNot
ificationDecorator decorator) {
        super(decorator);
        mContext = context;
    }

    @Override
    public NotificationCompat.Builder decorate(NotificationComp
at.Builder builder, Bundle bundle) {
        Intent intent = new Intent(ctx, MainActivity.class);
        PendingIntent pendingIntent = PendingIntent.getActivit
y(ctx, 0, intent, 0);
        builder.setContentIntent(pendingIntent);
        return chain(builder, bundle);
    }
}
```

Android SDK - Translations SDK Overview

Download 2.0.0 (https://bintray.com/halo-mobgen/maven/HALO-Translations/_latestVersion)

What is it for?

The HALO Translations SDK is a helper library that works as a plugin for HALO that, given a module name and the fields for the key and value it is able to bring the texts to your UI. The typical use case for this library goes over showing in the UI the translated texts that are present in the HALO backend. It makes it easy to make it work offline and asynchronous.

This library only supports one text for the same module at the same time so, once you change the language it will be cached removing the previous one.

Texts are synchronized based on the changes of the CMS, so if one text is removed, added or updated only those changes will be downloaded when the plugin is used. You can use push notifications also to redownload the texts if they have changed and you want them to be updated on the opened apps, but this use case is just a possible implementation and does not come directly implemented in the library.

Setup the lib

You can setup the translations using the HALO plugin:

```
halo {
    services {
        translations true
    }
}
```

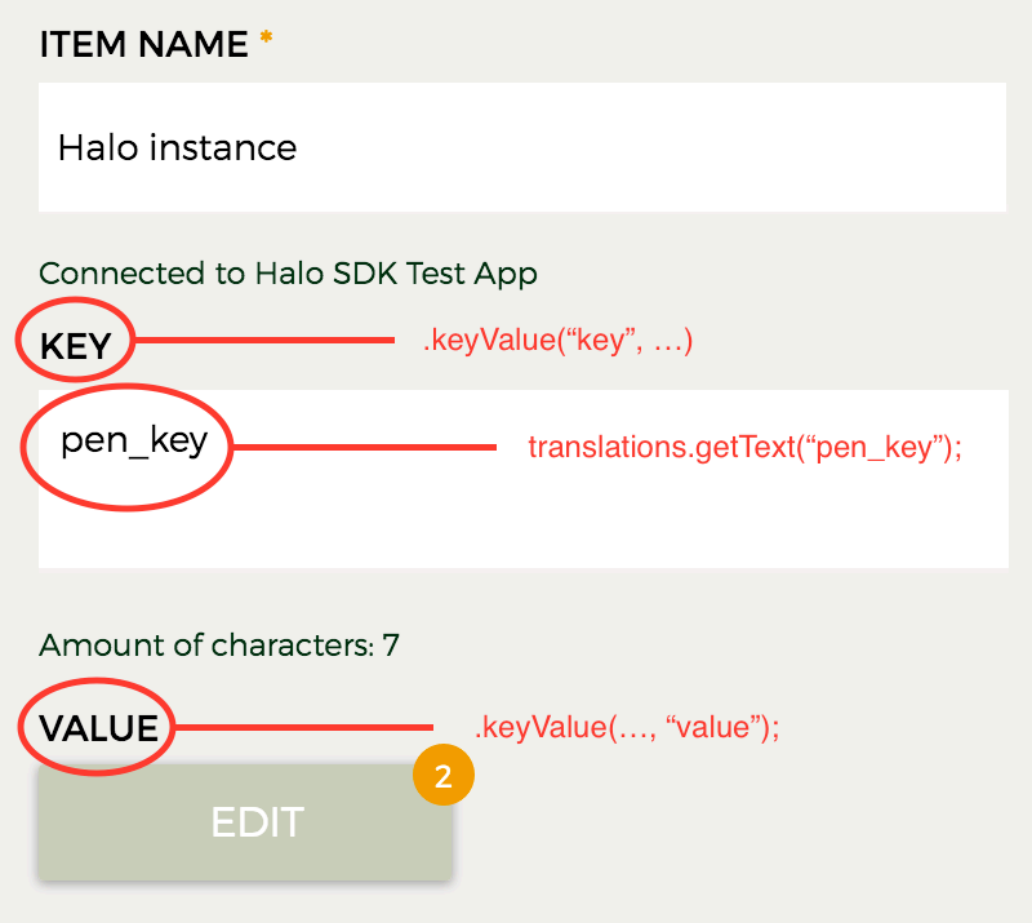
How can I use it?

Create a the plugin instance

It is fairly recommendable to create the instance as a singleton in your application class or using Dagger after installing HALO.

To make this work with HALO you will need to configure the module in the following way:

- A module created in HALO and its *module name*.
- Create as a manager two fields for this module, *one as a text* and a *second as a localized text*.
- Create some instances for this module with some localized texts.



To create the Android Application translations instance you will have to do the following:

```
Halo halo = Halo.installer(context).install();
HaloTranslationsApi translations = HaloTranslationsApi.with(hal
o)
                .locale(HaloLocale.ENGLISH_UNITED_STATES)
                .keyValue("key", "value")
                .moduleName(moduleName)
                .defaultText("No translation found")
                .defaultLoadingText("Translating...")
                .provideDefaultOnAsync(true)
                .build();
translations.load();
```

As you can see there are some parameters that you can provide to configure your translations plugin. In the following list you will find an explanation for each of those:

- **Locale** (Mandatory): This field specifies which is the locale that will be cached. You can change it later in the same instance using the `changeLocale()` method.
- **KeyValue** (Mandatory): To let the plugin know which of the values for the instance should be cached, you have to provide the name of the field that serves as a key and the name of the value that serves as a localized text. Afterwards you will be able to load a text by the key name in a TextView view.
- **ModuleName** (Mandatory): The name of the module. This name will be used to select which module should be synchronized. You can have as many modules as you want for localized texts as long as each module has its own instance of `HaloTranslationsApi`.
- **DefaultText** (Optional): You can provide either a default text or a strategy callback to select which text will be used as default in the case this text is not present in the module provided for any reason (no connection, someone removed it…). This parameter is optional.
- **DefaultLoadingText** (Optional): You can also optionally select a text that will be displayed when the translations are being loaded.
- **ProvideDefaultOnAsync** (Optional): The callback that provides the texts allows also to get the default value when loading. While loading the values it will call the text ready listener with the default value meanwhile the real one is being loaded.

Once the instance is created you can load the text as Strings or using a direct TextView. You don't have to worry about memory leaks, rotations or so, just use your key, the plugin handles that for you. In the following snippet you can have a look how to do it:

```
translationsInstance.textOn(textView, keyName);
```

or

```
mTranslationApi.getTextAsync(keyName, new TextReadyListener() {
    @Override
    public void onTextReady(@Nullable String key, @Nullable Str
ing text) {

    }
});
```

You can also use one of the following methods to interact with the translations api.

Method name	Explanation
`cancel`	Cancels the query done to sync the process.
`changeLocale`	Changes the locale in the current instance with another removing the previous translations and loading the new ones again.
`clearCallbacks`	Clears the pending callbacks.
`clearTranslations`	Clears the in memory translations. The ones in the database will remain.
`clearCachedTranslations`	Clears the in database and in memory translations.
`clearAll`	Clears callbacks and in mememory translations.
`getAllTranslations`	Provides a list of all the texts in the in memory map.
`getInMemoryTranslations`	Provides a map copy for the items in memory.

Method name	Explanation
`getDefaultText`	Get the default text for the given key.
`getText`	Gets the in memory text for the given key.
`getTextAsync`	Gets the text as an async process to make sure it is loaded when it is provided.
`textOn`	Assigns a text on a text view when it is ready. It internally used the `setTextAsync` .
`load`	Loads asynchronously and sync the pending texts.
`isLoading`	Tells if the plugin is still syncing the data.
`locale`	Provides the current locale configured.
`moduleName`	Provides the module name for this translations instance.
`setErrorListener`	Listen for possibly errors.
`removeLoadCallback`	Removes a loading callback in case you are in a context dependent environment.

Android SDK - Presenter SDK Overview

Download 2.0.0 (https://bintray.com/halo-mobgen/maven/HALO-Presenter/_latestVersion)

This library is intended to help an application to implement and attach the presenter lifecycle with the Activity or Fragment one. It provides a default implementation of the presenter called `EmptyPresenter` if you want to have an activity without a real presenter.

The MVP architecture is widely used when creating interfaces and having a helper in Android is really useful. Follow the next sections to learn how to use it and which are the functionalities provided.

In MVP there are three roles:

- The model: represents the content that will be displayed.
- The view: contains the UI elements and all the logic related to views and they way they are displayed.
- The presenter: brings the data and calls the needed methods from the view to allow the bridge between real business logic and the presentation logic.

Setup the lib

You can use the HALO plugin to configure it in an easy way:

```
halo {
        ...
        services {
                presenter true
        }
        ...
}
```

Since it is really well integrated with HALO, it provides in the presenter a method called `onInitialized` called once halo is ready. This can help us to show something while HALO is being prepared and also having a notification to perform some actions with it.

We provide also an interface called `HaloViewTranslator` that has some typical methods like `startLoading`, `stopLoading` or `showError`. This view translator should be implemented by your activity or other interfaces that inherits from it to provide functionality to the presenter.

To persist data also in the bundle the presenter provides two methods: `onSaveInstanceState` and `onInitStarted`, both managing the bundles on which the data is stored and retrieved respectively.

You can also inherit from the `AbstractHaloPresenter` to avoid reimplementing everyting and because it has already implemented the support for network conection drops and presenter lifecycle management.

How can I use it?

1. Inherit from one of `HaloActivity`, `HaloActivityV4`, `HaloFragment` or `HaloFragmentV4`.
2. Create a presenter that inherits from `AbstractHaloPresenter`.
3. Create an interface for your view that inherits from `HaloViewTranslator`.
4. Make your activity implement that interface.
5. Start coding your MVP logic in both, presenter and view.