



HALO SDK - iOS

Version 2.5

Last generated: March 28, 2018

Table of Contents

Overview

Getting Started	3
-----------------------	---

HALO Core SDK

Overview	7
SSL pinning.....	13
User authentication.....	14
User pocket	17

Content

Overview.....	20
Content search	24
Content sync	28
Content edition.....	30

Helpers

Cloudinary Helper.....	31
Translations Helper	32
Logging	33

HALO Notifications SDK

Overview	35
Rich notifications	42
Advanced Firebase Use Cases.....	48

HALO Social SDK

Overview	50
Facebook integration	55
Google integration	59

Release Notes

2.4.0 Release notes - Bora	63
2.3.1 Release notes - Altamira.....	64
2.3.0 Release notes - Altamira.....	65
2.2.5 Release notes	66
2.2.4 Release notes	67
2.2.3 Release notes	68
2.2.2 Release notes	70
2.2.1 Release notes	71
2.2.0 Release notes	72

2.1.1 Release notes	73
2.1.0 Release notes	74

Getting Started

In the sections below the process to integrate the Halo Framework into a new project will be described in depth. The Framework is developed in Swift 3 and compiled against the latest iOS SDK (11.0). In order to offer higher compatibility, the deployment target is set to iOS 8.0.

Carthage

One of the easiest ways to integrate the Halo Framework into your project and manage all the dependencies is by using Carthage (<https://github.com/Carthage/Carthage>).

In order to do that, we would need to add the following line to our Cartfile :

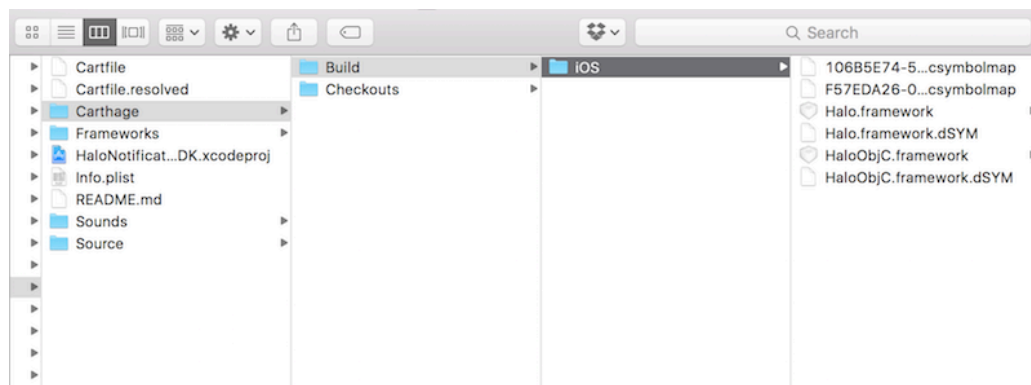
```
github "mobgen/halo-ios" "2.5.0"
```

After that, and by performing the following command in the terminal

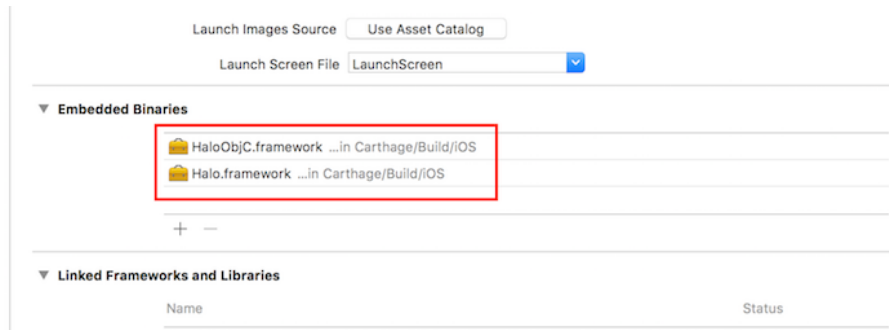
```
carthage update --platform iOS

*** Checking out halo-ios at "2.5.0"
*** xcodebuild output can be found in /var/folders/51/r2wsrnx1t9d99
1gs_bnwqsr0000gn/T/carthage-xcodebuild.syE1RN.log
*** Building scheme "Halo ObjC" in HAL0.xcworkspace
*** Building scheme "Halo iOS" in HAL0.xcworkspace
```

we will get the compiled frameworks we need in order to make our project work.



We would then need to add those frameworks as embedded frameworks (iOS 8+). Depending on the language we are using in our project (Swift/Objective-C) we will either need to just add the Halo.framework or also the HaloObjC.framework (as shown in the image).



CocoaPods

CocoaPods is also supported, so if this is your choice to manage your dependencies, you can add the Halo Framework to your project by adding the following line to your Podfile

```
pod 'HaloSDK', '2.5.0'
```

If using Objective-C, the corresponding SDK the line should be

```
pod 'HaloObjCSDK', '2.5.0'
```

Configuration

The configuration process is quite simple. Your app will only need to contain a `.plist` file within the main bundle where its credentials are set. By default, the Framework will look for a `HaLo.plist`, but a custom name can also be specified if needed. That will be covered later on in this wiki.

This `.plist` file should contain at least the `CLIENT_ID` and `CLIENT_SECRET` key-value pairs, with the right values for your app.

[illegible]

The full list of available keys for this configuration file is the following:

Key	Type	Description	Default value
CLIENT_ID	String	Client id for the authentication as application	-
CLIENT_SECRET	String	Client secret associated to the account used by the application	-
ENVIRONMENT	String	The desired Halo environment	https://halo.mobgen.com
DISABLE_SSL_PINNING	Boolean	Disable SSL pinning (usually for testing purposes).	NO
ENABLE_SYSTEM_TAGS	Boolean	Enable a set of default system tags added to the user (OS, device, etc).	NO

Troubleshooting

Q: I'm getting the following error when trying to run my app including the Halo SDK.

```
dyld: Library not loaded: @rpath/libswiftCore.dylib
  Referenced from: [...]Frameworks/Halo.framework/Halo
  Reason: Incompatible library version: Halo requires version 1.0.0
or later, but libswiftCore.dylib provides version 0.0.0
```

A: Regardless the method you choose to integrate the Halo SDK, you will need to make sure that the Always Embed Swift Standard Libraries under the Build Settings in your project is set to YES in order to avoid errors related to Swift.

GeneralCapabilitiesResource TagsInfoBuild Settings

BasicCustomizedAllCombinedLevels+

▼ Build Options

Setting

HaloSwiftDemo-ci

Always Embed Swift Standard LibrariesYes ↕

iOS Core SDK Overview

The Core Manager

Once all the configuration is done, the only remaining step is to call the `startup` method on the core manager to start all the process and initialise the SDK.

[Swift \(page 7\)](#) [Obj-C \(page 0\)](#)

```
Halo.Manager.core.startup { [weak self] (success) -> Void in
    // Do your stuff
}
```

It should also be mentioned here that apart from that `.plist` configuration file, the same configuration can be achieved through code, setting the corresponding properties within the manager (credentials and authentication mode) before calling the `startup` method.

This manager also has a delegate conforming to the `ManagerDelegate` protocol, that the developer could implement to customise the setup and launching process of the Framework.

The Core Manager also holds the configurable parameters of the Framework that can be set according to the needs of each project.

delegate

Setting the delegate of the core manager will allow to implement some hook methods that will be executed during the setup process of the Framework. This delegate must conform to the `Halo.ManagerDelegate` protocol.

[Swift \(page 7\)](#) [Obj-C \(page 0\)](#)

```
public protocol ManagerDelegate {
    func managerWillSetupDevice(device: Halo.Device) -> Void
}
```

`managerWillSetupDevice` will be called when the device is being set up, so that the developer could potentially add any details to the it. E.g.:

[Swift \(page 8\)](#) [Obj-C \(page 0\)](#)

```
// MARK: ManagerDelegate methods

func managerWillSetupDevice(device: Halo.Device) -> Void {
    device.addTag(name: "test", value: "testValue")
}
```

Later on, the device will be accessible through the Core Manager's `device` property.

environment

Apart from the configuration `.plist`, this is one of the properties that can also be set programatically. There is a set of predefined environments, but also an option to specify a custom one by providing the full url.

[Swift \(page 8\)](#)

[Obj-C \(page 0\)](#)

```
public enum HaloEnvironment {
    case int
    case qa
    case stage
    case prod
    case custom(String)
}
```

The environment can be changed then using the following function in the Core Manager, which provides also a completion handler to be executed once the environment has been successfully changed.

```
public func setEnvironment(_ env: HaloEnvironment, completionHandler: ((Bool) -> Void)? = nil)
```

defaultOfflinePolicy

A caching system is available out of the box for the network requests executed through the Framework. There are three modes in which the Framework can operate:

- **None** : No caching for the network requests is done
- **LoadAndStoreLocalData** : The data is fetched from the network and also stored in order to be used in other scenarios (e.g. temporary lack of internet connection)

- **ReturnLocalDataDontLoad** : Don't even try to fetch data and just return whatever information is cached locally

Policy	Swift	Obj-C
None	<code>.none</code>	<code>HaloOfflinePolicyNone</code>
<code>LoadAndStoreLocalData</code>	<code>.loadAndStoreLocalData</code>	<code>HaloOfflinePolicyLoadAndStoreLocalData</code>
<code>ReturnLocalDataDontLoad</code>	<code>.returnLocalDataDontLoad</code>	<code>HaloOfflinePolicyReturnLocalDataDontLoad</code>

Although this is the default mode, it can also be specified per request, overriding this default value if needed.

`numberOfRetries`

A retry system for failed requests is implemented within the Framework. If desired, a value can be set so that the requests are retried `n` times before finally failing. The default value is `0`, so no retries will be automatically done.

Just like the `defaultOfflinePolicy`, this is a default value that will be used only if a specific value is not set per request.

`authenticationMode`

The SDK provides and uses three authentication modes (app/app+/user), based on which it will use the credentials provided.

```
public enum AuthenticationMode {
    case app
    case appPlus
    case user
}
```

Those credentials can either be specified through the already mentioned `.plist` configuration file (using the keys `CLIENT_ID`, `CLIENT_SECRET`, `USERNAME` and `PASSWORD`) or through code, setting the properties `appCredentials` and/or `userCredentials` as follows:

```
Halo.Manager.core.appCredentials = Credentials(clientId: "clientId", clientSecret: "clientSecret")
Halo.Manager.core.userCredentials = Credentials(username: "username", password: "password")
```

Switching the authentication mode would be then as easy as:

```
HALO.Manager.core.authenticationMode = .user
```

And again, this authentication mode will be used throughout the framework when no specific mode is set per request.

Other functions

Apart from the aforementioned properties, the Core Manager provides a set of functions:

```
public func startup(completionHandler handler: ((Bool) -> Void)?)  
-> Void
```

Will be called in order to start the setup and launching process of the Framework. The closure will be executed once the whole process has finished and all the Framework features are ready to be used.

```
public func saveDevice(completionHandler handler: ((NSHTTPURLResponse?, HALO.Result<HALO.Device?, NSError>) -> Void)? = nil) -> Void
```

It will save the device model that the HALO SDK is currently holding and using for the different operations.

Add-ons

The architecture of the Halo Framework is based on a system of add-ons intended to easily integrate and provide extra features and functionalities.

Currently there are three protocols that are being used by different existing addons in the Framework's ecosystem:

```

public protocol Addon {

    var addonName: String {get}

    func setup(core: Halo.CoreManager, completionHandler handler: ((Halo.Addon, Bool) -> Void)?) -> Void
    func startup(core: Halo.CoreManager, completionHandler handler: ((Halo.Addon, Bool) -> Void)?) -> Void

    func willRegisterAddon(core: Halo.CoreManager) -> Void
    func didRegisterAddon(core: Halo.CoreManager) -> Void

    func willRegisterDevice(core: Halo.CoreManager) -> Void
    func didRegisterDevice(core: Halo.CoreManager) -> Void

    func applicationDidFinishLaunching(application: UIApplication, core: Halo.CoreManager) -> Void
    func applicationDidEnterBackground(application: UIApplication, core: Halo.CoreManager) -> Void
    func applicationDidBecomeActive(application: UIApplication, core: Halo.CoreManager) -> Void
}

```

Every single add-on will conform to the protocol shown above, so that it has the chance to perform the desired operations in different steps of the setup and lifecycle of the app.

More specific protocols exist depending on the purpose of the add-on (notifications, network traffic).

```

public protocol NotificationsAddon: Addon {

    func application(application: UIApplication, didRegisterForRemoteNotificationsWithDeviceToken deviceToken: NSData, core: Halo.CoreManager) -> Void
    func application(application: UIApplication, didFailToRegisterForRemoteNotificationsWithError error: NSError, core: Halo.CoreManager) -> Void

    func application(application: UIApplication, didReceiveRemoteNotification userInfo: [NSObject : AnyObject], core: Halo.CoreManager, fetchCompletionHandler completionHandler: (UIBackgroundFetchResult) -> Void) -> Void

}

```

```
public protocol NetworkAddon: Addon {  
  
    func willPerformRequest(request: NSURLRequest) -> Void  
    func didPerformRequest(request: NSURLRequest, time: NSTimeInterval, response: NSURLResponse?) -> Void  
  
}
```

Adding an add-on to the core is as simple as registering it using the appropriate function (always before calling the `startup` function):

```
fileprivate func setup() -> Void {  
  
    [...]  
  
    let notificationsAddon = FirebaseNotificationsAddon()  
    Halo.Manager.core.registerAddon(notificationsAddon)  
  
    [...]  
}
```

SSL Pinning

Using SSL for network connections is the de facto method of ensuring secure data transmission in today's mobile apps.

The default way iOS SSL connections work is as follows: the client makes a connection to the server and the server responds with its SSL certificate. If that certificate was issued by a Certificate Authority that is trusted by the OS, then the connection is allowed. All data sent through this connection is then encrypted with the server's public key. For an attacker to perform a *man in the middle* attack, the mobile device would have to trust the attacker's certificate. It is very unlikely that the attacker possesses a trusted certificate and therefore this is normally not an issue. However SSL weaknesses have happened before and using SSL Pinning can mitigate this possibility.

SSL Pinning is making sure the client checks the server's certificate against a known copy of that certificate.

SSL pinning within the HALO iOS SDK

SSL pinning is enabled by default in the SDK, and the certificate for those domains under *.mobgen.com are already delivered with it. However, it can happen that the SDK is being used against any other environment/domain. In that case, the certificate for that environment will have to be added to the main bundle of the application.

By adding all the possible certificates in any of the supported formats (.cer, .crt, .der), the SDK will automatically try to match those against the server's certificate.

Disabling SSL pinning

If for any reason the SSL pinning needs to be disabled (testing, debugging... not recommended in final builds), it can be done by adding the boolean key `DISABLE_SSL_PINNING` in the configuration `.plist` and setting it to `YES`.

User authentication

The HALO SDK provides a way to register and store user accounts within the platform, allowing those users to later log in into the platform. All these operations will be performed through the `AuthManager` .

UserProfile and AuthProfile

The two basic structures involved in the authentication process within HALO are the `UserProfile` and the `AuthProfile` .

The first one will be needed when registering a user, and it will contain basic information about the user, such as email, name, surname, etc. On the other hand, the latter will store information about the credentials (username, password, token, etc) and will be needed both when registering a new user and also when performing the login action.

UserProfile

[Swift \(page 14\)](#)[Obj-C \(page 0\)](#)

```
public init(id: String?,
            email: String,
            name: String,
            surname: String,
            displayName: String?,
            profilePictureUrl: String?)
```

AuthProfile

An `AuthProfile` can be constructed in two ways, either by providing the email, password and deviceId (being the latter optional in Swift, it will default to the deviceId of the current device if any) or by providing a token, the identifier of a network and the deviceId (same as before).

This last option is intended to be used when integrating the login in HALO with other social networks (Google, Facebook).

[Swift \(page 15\)](#)[Obj-C \(page 0\)](#)


```
public init(email: String, password: String, deviceId: String?
= default)

public init(token: String, network: Halo.Network, deviceId: String? = default)
```

Registration

Registering a user in the platform is really simple after creating the two structures containing the required information. Once the call is performed, the result will be provided within the completion handler, either by returning the successfully registered user in HALO, or an error if any has occurred.

[Swift \(page 15\)](#)[Obj-C \(page 0\)](#)

```
public func register(authProfile: AuthProfile,
                    userProfile: UserProfile,
                    completionHandler handler: @escaping (UserProfile?,
HaloError?) -> Void) -> Void
```

Login

For an already registered user, the login action can be performed by providing the corresponding `AuthProfile` containing the credentials of that user. An extra parameter (`stayLoggedIn`) allows the developer to decide whether those credentials should be safely stored (in the keychain) to try and automatically log in again when the app is restarted.

[Swift \(page 15\)](#)[Obj-C \(page 0\)](#)

```
public func login(authProfile: AuthProfile,
                 stayLoggedIn: Bool,
                 completionHandler handler: @escaping (User?,
HaloError?) -> Void) -> Void
```

Sample code

[Swift \(page 16\)](#)[Obj-C \(page 0\)](#)

```
import Halo

[...]

// Set an AuthProfile to login
let authProfile = AuthProfile(email: "your@email.com", password: "yoursecretpassword")

// Request login with the AuthProfile
Manager.auth.login(authProfile: authProfile, stayLoggedIn: false) { (user, error) in
    if error != nil {
        // Something went wrong.
    } else {
        // User logged in successfully. Do something with "user".
    }
}
```

The **current user**, if any, can be accessed at any time through the `currentUser` property of the `AuthManager` instance (optional in Swift, nullable in Objective-C).

Logout

The logout action is fairly simple as well. Only by calling the `logout` function, the `currentUser` will be removed and the result of the operation will be provided in the completion handler.

[Swift \(page 16\)](#)[Obj-C \(page 0\)](#)

```
public func logout(completionHandler handler: @escaping (Bool)
-> Void) -> Void
```

User pocket

The HALO SDK provides a way to store custom data for a given identified user, and it is called Pocket .

The Pocket model

Each user may have his own pocket stored in the server containing custom information. The structure of the pocket is simple and it has two components: references and data .

references	Set of collections (with custom keys) that will allow, for example, to store references to different items (instances, modules, etc)
data	Custom JSON (limited to 2000 characters)

The raw representation of a pocket from the server point of view would be something like the following:

```
{
  references: {
    favorites: [ 'id1', 'id2', ... ],
    instances: [ 'id1', 'id2', ... ],
    modules: [ 'id1', 'id2', ... ]
  },
  data: {
    // custom json object limited to 2000 chars (length)
  }
}
```

In order to modify the pocket there are a set of helper functions that simplify the fact of making changes

[Swift \(page 18\)](#)[Obj-C \(page 0\)](#)

```
// Add a reference under the given key
public func addReference(key: String, value: String) -> Void

// Remove a reference under the given key. Will return true if s
// ucceeds, false otherwise
public func removeReference(key: String, value: String) -> Bool

// Set a collection of references directly under a key. It will
// overwrite any existing ones
public func setReferences(key: String, values: [String]?) -> Voi
d

// Remove all the references stored under a key (it will interna
// lly set it to nil)
public func removeReferences(key: String) -> Void

// Set the custom JSON data to be stored in the pocket
public func setData(_ data: [String: Any]) -> Void
```

Retrieving and storing a Pocket

Any operation over the pocket will require a previously logged in user. These requests will only work using credentials of identified user (*app+* role).

The first step will be creating an empty `Pocket`, populating it with the desired values and storing it in the server. If the operation succeeds, the saved `Pocket` will be returned through the completion handler.

[Swift \(page 18\)](#)

[Obj-C \(page 0\)](#)

```
public func savePocket(
    _ pocket: Pocket,
    completionHandler handler: @escaping (HTTPURLResponse?, Resul
t<Pocket?>) -> Void
) -> Void
```

Getting a `Pocket` for the logged in user is really straightforward. An extra feature will allow the developer to filter the references to be retrieved. The keys can be specified as an argument, and the resulting `Pocket` will only contain those references. This will help saving network traffic and processing unnecessary data. Setting it to `nil` will avoid any filtering.

[Swift \(page 19\)](#)[Obj-C \(page 0\)](#)

```
public func getPocket(  
    filterReferences: [String]? = nil,  
    completionHandler handler: @escaping (HTTPURLResponse?, Result  
t<Pocket?>) -> Void  
) -> Void
```

HALO Core SDK - Content

Modules and instances

All the content manipulation from this HALO SDK is based on the existence of two main models: `Module` and `ContentInstance` (`HaloModule` and `HaloContentInstance` if using the Obj-C version of the SDK). These two models match perfectly with the concepts present in the CMS.

`Module` (`HaloModule`)

Property	Swift type	Obj-C type	Description
<code>customerId</code>	<code>Int?</code>	<code>NSInteger</code>	Id of the customer owning this module
<code>id</code>	<code>String?</code>	<code>NSString</code>	Id of the module
<code>name</code>	<code>String?</code>	<code>NSString</code>	Name of the module
<code>isSingle</code>	<code>Bool</code>	<code>BOOL</code>	Whether the module is a single one or not
<code>createdAt</code>	<code>Date?</code>	<code>NSDate</code>	Creation date of the module
<code>updatedAt</code>	<code>Date?</code>	<code>NSDate</code>	Date of the last update of the module
<code>deletedAt</code>	<code>Date?</code>	<code>NSDate</code>	Deletion date of this module
<code>createdBy</code>	<code>String?</code>	<code>NSString</code>	User responsible for the creation of the module

updatedBy	String?	NSString	User responsible for the last update of the module
deletedBy	String?	NSString	User responsible for the deletion of the module
tags	[String: Tag]	NSDictionary	Collection of tags associated to this module

ContentInstance (HaloContentInstance)

Property	Swift type	Obj-C type	Description
id	String?	NSString	Id of the instance within the platform
moduleId	String	NSString	Id of the module to which this instance belongs
name	String	NSString	Name of the instance
values	[String: Any]	NSDictionary	Dictionary containing the key-value pairs representing the custom data of the instance
createdAt	Date	NSDate	Creation date of the instance
publishedAt	Date?	NSDate	Publication date of the instance
updatedAt	Date?	NSDate	Date of the last update of the instance

deletedAt	Date?	NSDate	Date in which the instance has been deleted
removedAt	Date?	NSDate	Date in which the instance has been removed
archivedAt	Date?	NSDate	Date in which the instance has been archived
createdBy	String?	NSString	User responsible for the creation of the instance
updatedBy	String?	NSString	User responsible for the last update of the instance
deletedBy	String?	NSString	User responsible for the deletion of the instance
tags	[String: Tag]	NSDictionary	Collection of tags associated with the instance

Another structure widely used related to the content is the `PaginationInfo` . Although pagination can be skipped, the result will be returned in the shape of paginated modules or instances depending on the call (to maintain coherence).

`PaginationInfo` (`HaloPaginationInfo`)

Property	Swift type	Obj-C type	Description
page	Int	Int	Value of the current page
limit	Int	Int	Limit set for the current result
offset	Int	Int	Offset which the current result starts from

totalItems	Int	Int	Total number of items resulting from the query
totalPages	Int	Int	Total number of pages resulting from the query

Retrieving modules

The collection of existing modules can be retrieved through the Core Manager using a specific call. The response will come in the shape of paginated content.

PaginatedModules ([HaloPaginatedModules](#))

Property	Swift type	Obj-C type	Description
paginationInfo	PaginationInfo	HaloPaginationInfo	Structure containing the pagination info
modules	[Module]	[HaloModule]	Array of modules

[Swift \(page 23\)](#)

[Obj-C \(page 0\)](#)

```
public func getModules(completionHandler handler: (NSHTTPURLResponse?, Result<PaginatedModules?>) -> Void) -> Void
```

It will return a request set up to request the available modules. It can be customised as needed and then executed.

Content search

In order to retrieve specific instances, a whole system based on queries and filters is implemented which will allow the developer to perform really fine-grained search operations.

Depending on the desired format of the results, the search operation can return already parsed content instances or, if any extra processing needs to be made, the SDK also offers the chance to receive the response as *raw* data.

[Swift \(page 24\)](#)[Obj-C \(page 0\)](#)

```
public func search(query: SearchQuery, completionHandler handle
r: @escaping (HTTPURLResponse?, Result<PaginatedContentInstance
s?>) -> Void) -> Void

public func searchAsData(query: SearchQuery, completionHandler h
andler: @escaping (HTTPURLResponse?, Result<Data>) -> Void) -> V
oid
```

There is a set of parameters that can be specified in order to define the search operation and the desired results, and those are provided using the `SearchQuery` object.

The `SearchQuery` (`HALOSearchQuery`)

The `SearchQuery` provides a flexible mechanism to build an object containing all the parameters that will be forwarded to the search engine in HALO. Since the functions return a `SearchQuery`, the calls can be conveniently chained.

[Swift \(page 24\)](#)[Obj-C \(page 0\)](#)

```
let query = SearchQuery().moduleIds([moduleId1, moduleId2]).sear
chFilter(filter).skipPagination()
```

The different functions than can be used to customise the `SearchQuery` are the following (expand each section to get all the details):

▶ [addRelatedInstances \(page 0\)](#)

▶ [addAllRelatedInstances \(page 0\)](#)

▶ [addSortBy \(page 0\)](#)

▶ [fields \(page 0\)](#)

▶ [instanceIds \(page 0\)](#)

▶ [locale \(page 0\)](#)

▶ [metaFilter \(page 0\)](#)

▶ [moduleIds \(page 0\)](#)

▶ [moduleName \(page 0\)](#)

▶ [pagination \(page 0\)](#)

▶ [populateFields \(page 0\)](#)

▶ [populateAll \(page 0\)](#)

▶ [searchFilter \(page 0\)](#)

▶ [segmentMode \(page 0\)](#)

▶ [segmentWithDevice \(page 0\)](#)

▶ [serverCache \(page 0\)](#)

▶ [skipPagination \(page 0\)](#)

▶ [tags \(page 0\)](#)

The **SearchFilter** (**HaloSearchFilter**)

A `SearchFilter` can be created through the existing Swift *high level* functions (static methods in Objective-C), which will allow to define conditions over different fields and their values:

[Swift \(page 26\)](#)
[Obj-C \(page 0\)](#)

```
public func eq(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func neq(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func gt(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func lt(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func gte(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func lte(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func valueIn(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func valueNotIn(property: String, value: Any?, type: String? = nil) -> SearchFilter
public func like(property: String, value: String) -> SearchFilter
public func or(_ elements: SearchFilter...) -> SearchFilter
public func and(_ elements: SearchFilter...) -> SearchFilter
```

Since all these operations return a `SearchFilter`, they can be conveniently combined to compose more complex filtering criteria.

[Swift \(page 26\)](#)
[Obj-C \(page 0\)](#)

```
let searchFilter = and(
    or(
        and(
            gte(property: "age", value: 18),
            lt(property: "age", value: 100)
        ),
        eq(property: "registration", value: nil)
    ),
    neq(property: "name", value: "July Young"),
    valueIn(property: "name", value: ["Forever alone", "Fake name"])
)
```


Content sync

One big feature that the SDK offers related to content is the ability to synchronize it, doing incremental updates, saving network traffic and waiting time.

In order to do that, the request has to be performed via a `SyncQuery`.

The SyncQuery (HaloSyncQuery)

With an empty constructor, the object is implemented in the sense of a builder, so that the different methods can be chained to create the desired `SyncQuery`.

<code>moduleId</code>	<code>String</code>	Id of the module to be synchronised
<code>moduleName</code>	<code>String</code>	Name of the module to be synchronised
<code>locale</code>	<code>Halo.Locale</code>	Locale in which we want to retrieve the information. If none provided, the default one will be used
<code>fromSync</code>	<code>Date</code>	Initial date from which we want to sync the content
<code>toSync</code>	<code>Date</code>	Final date to which we want to sync the content

The sync operation

In order to start a sync, the operation can be triggered via the `ContentManager` by providing a `SyncQuery` that fits the requirements.

[Swift \(page 28\)](#)[Obj-C \(page 0\)](#)

```
public func sync(
    query: SyncQuery,
    completionHandler handler: @escaping (String, HaloError?) -> Void
) -> Void
```

Once the operation has finished and the content is stored locally, both the items and a log containing a summary of the operations performed (creations, updates and deletions) can be handled through a set of existing functions in the `ContentManager` .

[Swift \(page 29\)](#)[Obj-C \(page 0\)](#)

```
public func getSyncedInstances(moduleId: String) -> [ContentInstance]

public func removeSyncedInstances(moduleId: String) -> Void

public func getSyncLog(moduleId: String) -> [SyncLogEntry]

public func clearSyncLog(moduleId: String) -> Void
```

Content edition

The HALO SDK provides a set of methods to create, edit and delete content. These methods are available through the `ContentManager` .

Content instances can be created or updated by sending them to the server via the right method:

[Swift \(page 30\)](#)[Obj-C \(page 0\)](#)

```
open func save(
    _ instance: ContentInstance,
    completionHandler handler: @escaping (HTTPURLResponse?, Halo.Result<ContentInstance?>) -> Void
) -> Void
```

Content instances can also be deleted providing the id of the instance:

[Swift \(page 30\)](#)[Obj-C \(page 0\)](#)

```
open func delete(
    instanceId: String,
    completionHandler handler: @escaping (HTTPURLResponse?, Halo.Result<ContentInstance?>) -> Void
) -> Void
```

Moreover, there is an option to perform a *batch operation* so that multiple simple operations (creation, edition, deletion) are performed in the same request, saving time and network traffic.

[Swift \(page 30\)](#)[Obj-C \(page 0\)](#)

```
open func batch(
    operations: BatchOperations,
    completionHandler handler: @escaping (HTTPURLResponse?, Halo.Result<BatchResult?>) -> Void
) -> Void
```


HALO Cloudinary Helper

HALO Translations Helper

Logging

The HALO SDK provides a flexible and powerful logging system, based on a protocol (`Logger`) which allows a developer to even extend the already existing functionality.

Registering loggers

In order for a logger to be used, it must be instantiated and registered in the `CoreManager` . This should be done prior to calling the `startup` function, to make sure all the requests are taken into account when logging the results.

[Swift \(page 33\)](#)[Obj-C \(page 0\)](#)

```
HALO.Manager.core.addLogger(ConsoleLogger())
```

Logging levels

There are several logging levels which can be set to the `CoreManager` instance (property `LogLevel`). The different values and meanings of each of the values of the enumeration are:

Level	Description	Swift	Obj-C
NONE	Disable any logging output. No information will be shown	<code>.none</code>	<code>LogLevelNone</code>
ERROR	Only show critical messages (errors)	<code>.error</code>	<code>LogLevelError</code>
WARNING	Show also messages related to warnings and potential issues	<code>.warning</code>	<code>LogLevelWarning</code>

INFO	The most verbose option, showing all the info related to requests and responses. Useful for debugging purposes	.info	LogLevelInfo
------	--	-------	--------------

NOTE: Each level includes all the less verbose ones below it (i.e. setting the log level to INFO will provide the messages from ERROR and WARNING apart from the own ones of the INFO level).

A log message can easily be added through the core manager as follows:

[Swift \(page 34\)](#)

[Obj-C \(page 0\)](#)

```
HALO.Manager.core.logMessage(message: "message", level: .info)
```

Available loggers

There are some loggers already implemented, but as mentioned earlier, a new logger could be implemented and used as long as it conforms to the Logger protocol.

ConsoleLogger

This is the most simple logger. It will print all the messages to the debugging console.

FileLogger

In this case, the logger dumps all the messages of the execution to a file, which can be later accessed. It could be useful if a log file needs to be sent via email or something similar.

The path of the file being written can be accessed via the property `filePath` of the logger (a reference to the file in the file system as a URL).

HALO Notifications SDK

Configuration

Adding notifications to a HALO-powered application from the SDK point of view is just as simple as instantiating a notifications addon and registering it within the core.

⚠ Warning: HALO will handle the download of all necessary dependencies to use the Firebase push notifications SDK. Do not duplicate Firebase libraries on your project or you will not receive correctly push notifications.

But in order to do that, the Framework needs to be added as a dependency to the project:

[Carthage \(page 35\)](#)

[CocoaPods \(page 0\)](#)

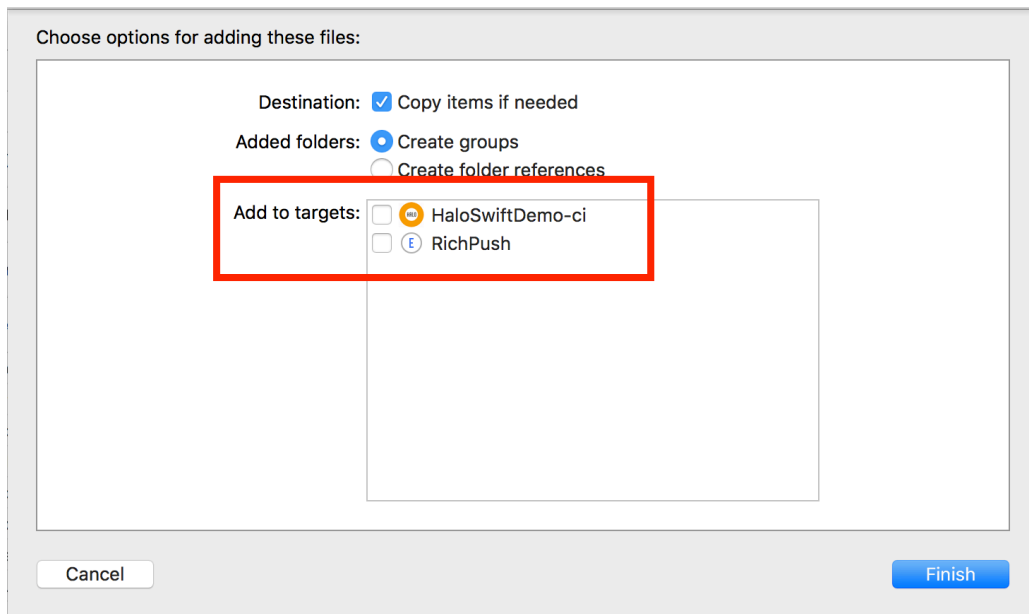
```
github 'mobgen/halo-notifications-ios' '2.5.0'
```

Custom sounds

The HALO platform offers the option to customise the push notifications by setting different sounds apart from the default one. Since that one is an optional feature, and in favour of a smaller SDK, the sound files are kept outside of the delivered SDK, leaving the responsibility of adding the needed ones to the developers.

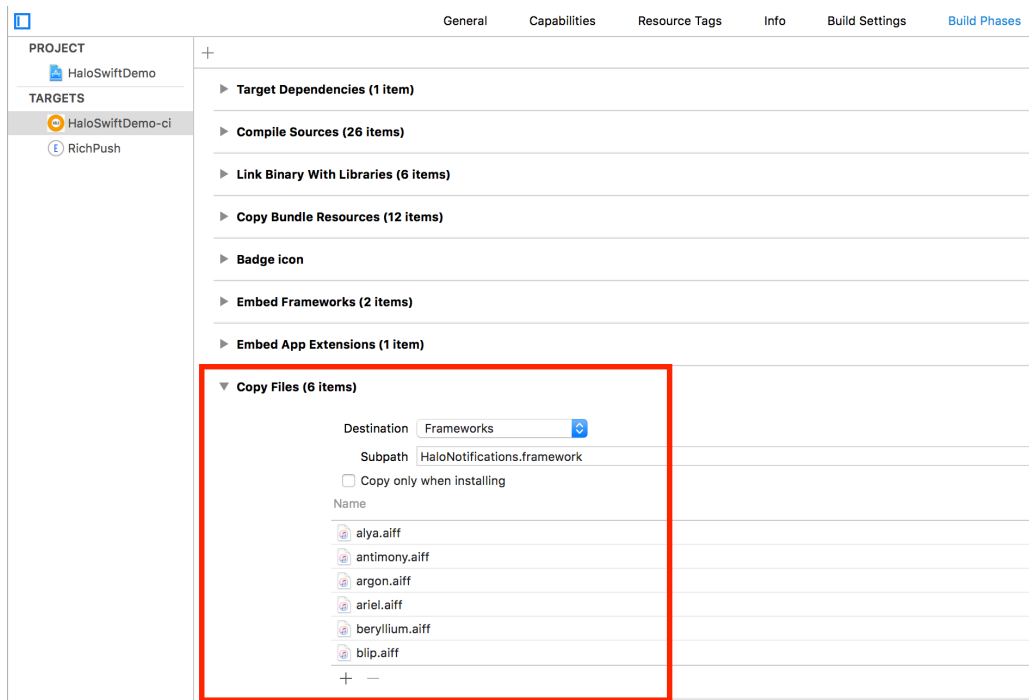
The full pack containing all of the custom sounds offered by the platform can be downloaded [here \(page 0\)](#).

After downloading the sounds, the desired ones will need to be copied to the project. Remember to uncheck everything from *Add to targets*, since we will be specifying a custom location for those files.



The next step will be adding a new *Build Phase*, where we will define the location where these sound files will be copied. The destination will be inside the notifications framework, and for that, we will add a *New Copy Files Phase*, where we will choose Frameworks as destination and add `HaloNotifications.framework` as subpath.

That way, the sound files will get copied to that location and will be used when receiving a push notification specifying one of those custom sounds.



The HaloNotification model

This Notifications SDK, contrary to what iOS does by default, offers a model which makes accessing some of the information contained in the push notifications slightly easier.

Field	Swift	Obj-C	Description
scheduleId	String?	NSString	Id which internally identifies the push notification within the HALO platform
title	String?	NSString	Title of the push notification
body	String?	NSString	Content of the message included in the push notification
icon	String?	NSString	Icon specified for the push notification (Android feature, but also accessible here in case it is needed)

sound	String?	NSString	Sound of the push notification. It will be automatically handled by the system, but it can be also accessed
type	HaloNotificationType	HaloNotificationType	Type of the push notification (normal, silent or two factor)
payload	[AnyHashable: Any]	NSDictionary	“Raw” payload of the push notification as received from the server

Usage

An already implemented add-on based on Firebase is provided by the SDK. Registering this add-on is then as simple as:

[Swift \(page 38\)](#)

[Obj-C \(page 0\)](#)

```
import HaloNotifications

let notificationsAddon = FirebaseNotificationsAddon()
notificationsAddon.delegate = self

Halo.Manager.core.registerAddon(notificationsAddon)
```

In order to handle the received notifications, a delegate should be set to the notifications add-on, conforming to the `NotificationsDelegate` protocol.


```
public protocol NotificationsDelegate {  
  
    func application(_ app: UIApplication,  
        didReceiveRemoteNotification notification: HaloNotificatio  
n,  
        userInteraction user: Bool,  
        fetchCompletionHandler completionHandler: ((UIBackgroundFe  
tchResult) -> Void)?) -> Void  
  
}
```

Also, an already implemented custom app delegate is provided by the core (`HaloAppDelegate`) so that the app delegate can inherit from it. If that's not possible, some methods need to be overwritten to redirect the flow to the Core Manager:

```

public func application(application: UIApplication,
    didRegisterForRemoteNotificationsWithDeviceToken deviceToken:
    NSData) {

    Manager.core.application(application,
        didRegisterForRemoteNotificationsWithDeviceToken: deviceTo
    ken)

}

public func application(application: UIApplication,
    didFailToRegisterForRemoteNotificationsWithError error: NSError) {

    Manager.core.application(application,
        didFailToRegisterForRemoteNotificationsWithError: error)

}

public func application(application: UIApplication,
    didReceiveRemoteNotification userInfo: [NSObject : AnyObject],
    fetchCompletionHandler completionHandler: (UIBackgroundFetchResult) -> Void) {

    Manager.core.application(application,
        didReceiveRemoteNotification: userInfo,
        fetchCompletionHandler: completionHandler)

}

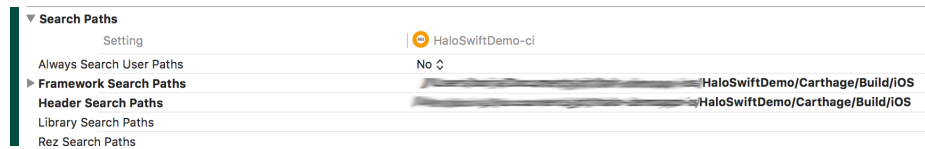
```

Configuring the project in Firebase

In order for the notifications to work, the project must be set up in Firebase, the system HALO uses to send the push notifications. A detailed guide provided by Google can be found [here \(https://firebase.google.com/docs/ios/setup\)](https://firebase.google.com/docs/ios/setup).

Troubleshooting

- Since the Notifications SDK relies on Firebase, the location of those SDKs (downloaded automatically as dependencies) will have to be added to the Header Search Paths under the project build settings. Probably something like `$(SRCROOT)/Carthage/Build/iOS`.



Enable notifications usage

When a new notification is received on a device, the SDK will send a request to HALO reporting notification updates. There are three different events we can report: receipt, open and dismiss. To enable this feature you must enable directly on the notification addon. This feature is only available on iOS 10+ versions.

⚠ Warning: Remember to set a valid notification category on the push notification. For the following example we will use this configuration:
 "click_action" : "dismiss_halo"

Swift (page 41)

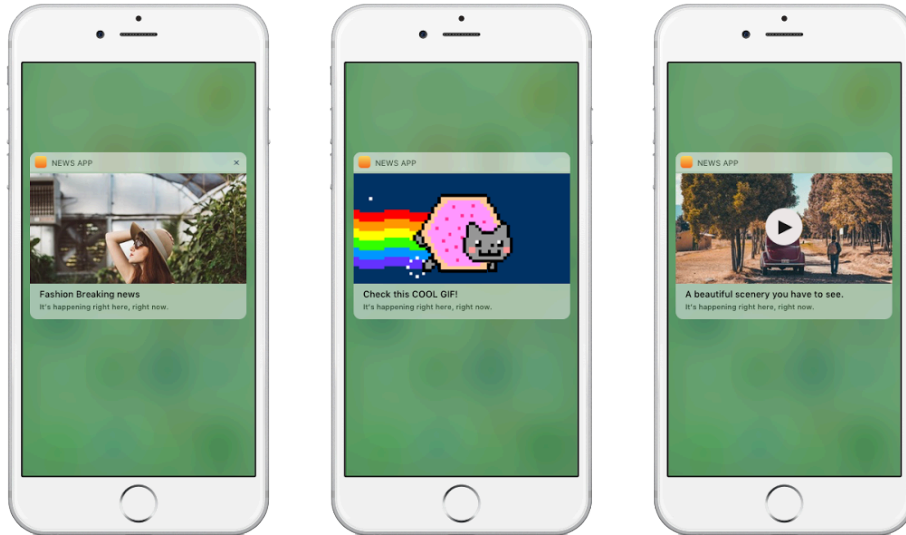
```
if #available(iOS 10.0, *) {
    UNUserNotificationCenter.current().delegate = self
    notificationsAddon.enableNotificationEvents(userNotificationCenter : UNUserNotificationCenter.current(), notificationCategory: "dismiss_halo")
}
```

You only have to override the following function to notify halo that a notification was dismissed or opened and HALO will manage everything for you.

```
@available(iOS 10.0, *)
func userNotificationCenter(_ center: UNUserNotificationCenter,
    didReceive response: UNNotificationResponse, withCompletionHandler completionHandler: @escaping () -> Void) {
    Manager.core.userNotificationCenter(center, didReceive: response, core: halo, fetchCompletionHandler: completionHandler)
}
```

Rich push notifications

Starting in iOS 10, push notifications can now be enhanced by adding multimedia content.



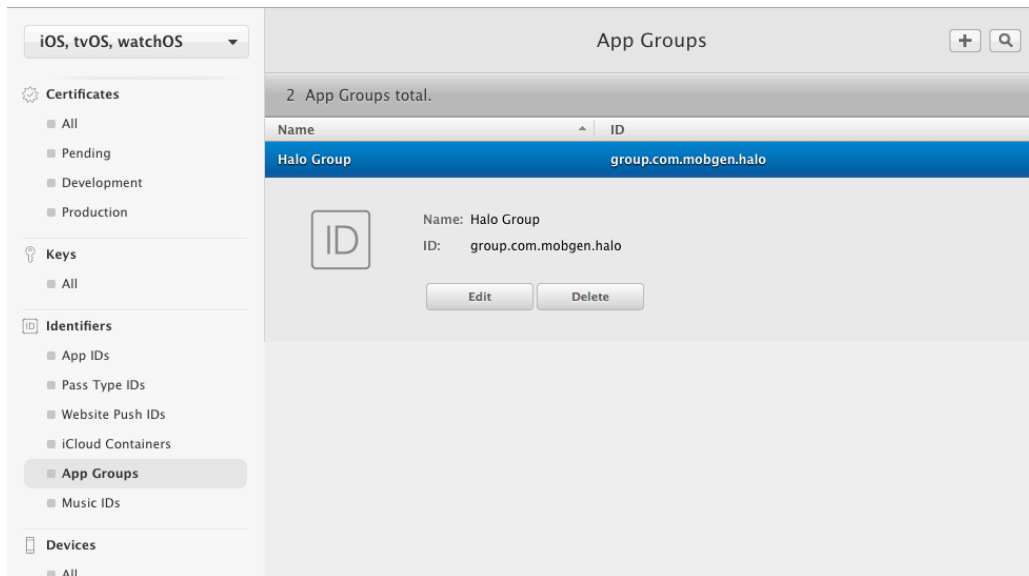
In order for it to work and make your app capable of receiving these rich push notifications, there are several configuration steps to be performed.

Configuration

Creating an App Group

The first step will be creating what Apple defines as an **App Group**, which is an identifier that will allow to create and identify shared containers so that apps and extensions can share content. The reason behind this is the need for the extension to download images in the background and store them in a place which can later be accessed.

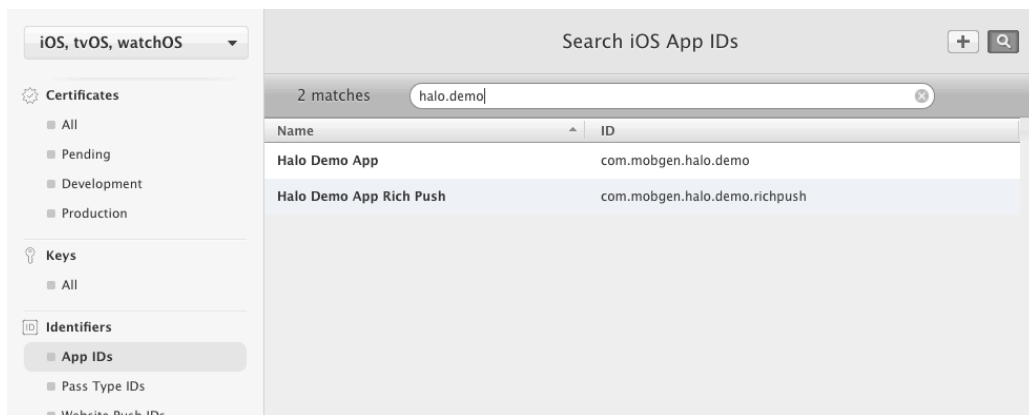
In order to do that, the app group **group.com.mobgen.halo** (that exact one, it is the one the SDK will look for) will need to be created within the section *App Groups* in the Developer Portal.



Creating a new App ID for the extension

Once the App Group is created, the next step will be to add a new **App ID** for the extension. Extensions define new bundle ids (which share the first portion with the hosting app) that will have to be set up also in the Apple Developer Program.

In this case, in the sample image, the identifier chosen for the extension is **richpush**, which will be embedded within a demo app, hence the full bundle id will be **com.mobgen.halo.demo.richpush**.



You will also need to make sure that both the extension app ID and also the hosting app ID, have the App Groups service enabled, as shown in the following images.

iOS App ID Settings

+

Q

Setup and configure services for this App ID.

ID


ID: com.mobgen.halo.demo.richpush

Name: Halo Demo App Rich Push

Enable


Service

☒


 **App Groups**
Enabled App Group IDs (1)

Edit

☐

 **Associated Domains**
Disabled

☐

 **Data Protection**

And remember to associate the appropriate App Group to the App ID.

App Group Assignment

+

Q

ID

App Group Assignment.

Select the App Groups you wish to assign to the bundle.

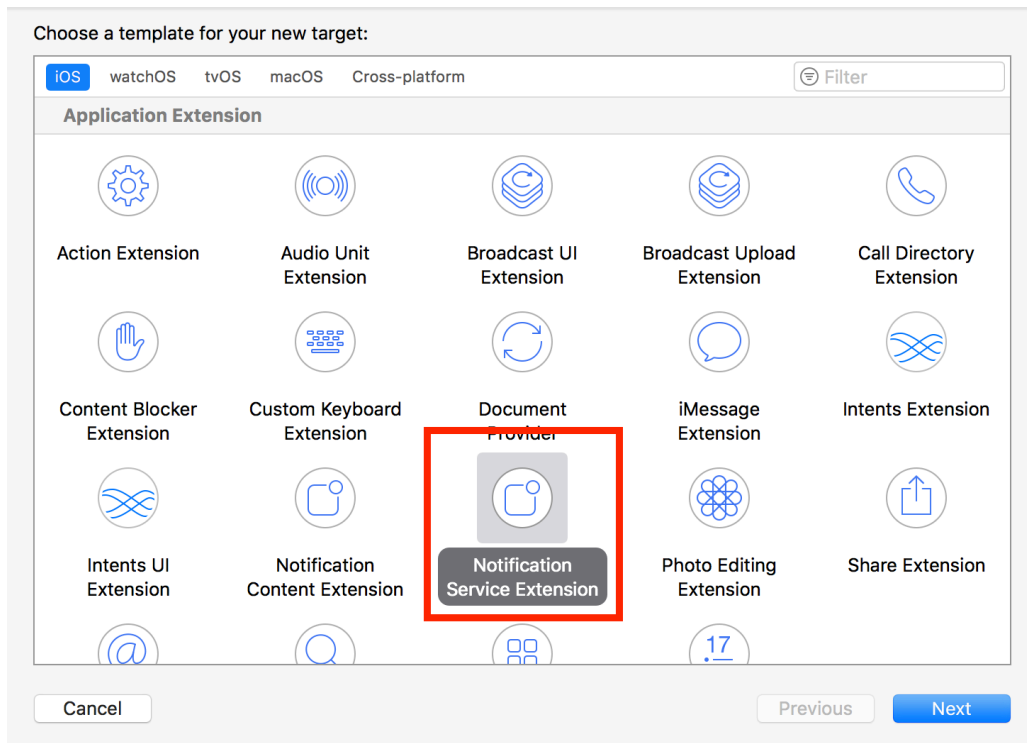
☐ Select All

☒ Halo Group

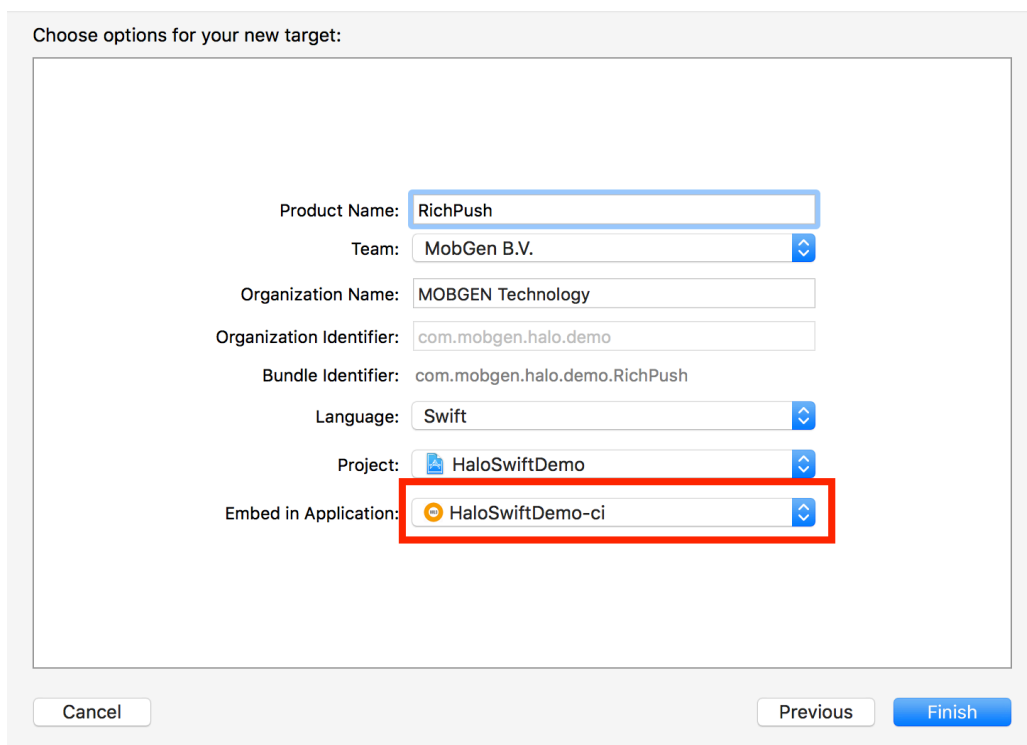
group.com.mobgen.halo

Creating a Notification Service Extension

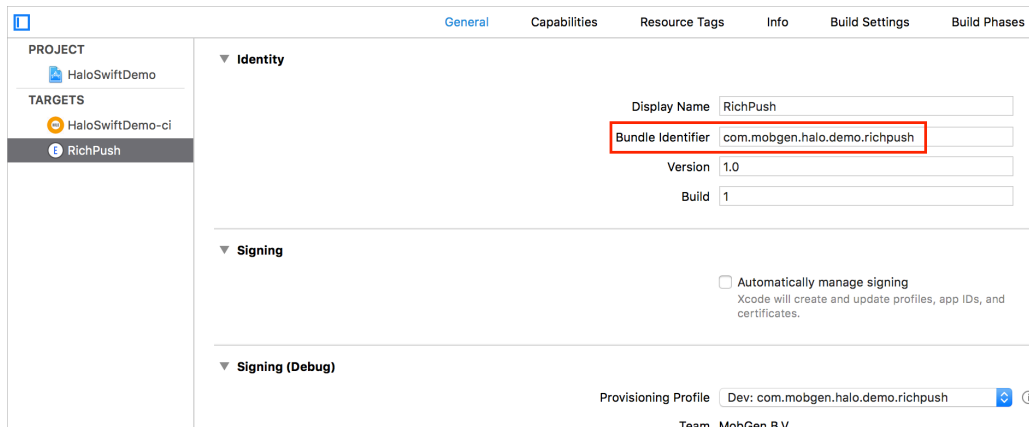
This new feature is achieved through a *Notification Service Extension*, which allows the modification of a push notification prior to being shown. This service extension is added to the app as a new target, so the first step will be adding a new target and selecting the *Notification Service Extension* template when asked.



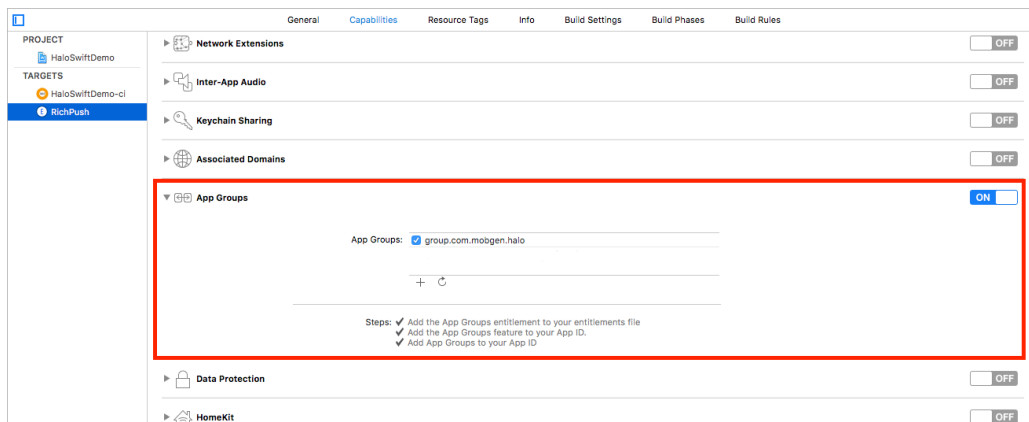
After that, a few settings need to be specified for the new target. Make sure that the extension is set to be embedded in your application.



Once the extension has been created, double check that the bundle identifier is the one it has been created in previous steps.



Another thing to check is that both the hosting app and the extension have the *App Groups* capability enabled and with the right group(s) selected.



Search paths

Remember to check that both the hosting app and the extension will need to have the Framework Search Paths and Header Search Paths correctly set (including the path to the Carthage folder, where the Firebase dependencies are downloaded).

Implementation

Dependencies

As of iOS 10.0, a new framework - `UserNotification.framework` - has been added in order to handle everything related to push notifications. Hence, that dependency needs to be added when targeting iOS 10+ devices.

Coding

When creating the service extension, a sample file with some code in it will be automatically created by Xcode. The HALO Notifications SDK already provides this feature implemented, so that no extra processing needs to be done by the developer.

All the code in that sample file can be replaced by the following one:

[Swift \(page 47\)](#) [Obj-C \(page 0\)](#)

```
import HaloNotifications

public class NotificationService: HaloNotificationService {

}
```

Advanced Firebase Use Cases

Advanced Firebase Use Cases

The Halo Push add-on module make usage of the Firebase libraries, including setting up their configuration and downloading the complete set of Firebase libs in order to be available for the developer by just by importing / referencing the required file. That helps on easing the setup process of the push notifications by taking care internally of the Firebase dependencies and setup.

External Configuration

The amount of flexibility of the Notifications add-on allows the developer to configure Firebase directly, the add-on will reuse that configuration by default. But TAKE CARE, the responsibility of having a correct and functional push notifications config relays on your end on those cases.

```
var firebaseConfigFile: String = "path/of/your/firebase/config/file"

if let firOptions = FirebaseOptions(contentsOfFile: firebaseConfigFile) {
    FirebaseApp.configure(options: firOptions)
} else {
    FirebaseApp.configure()
}
```

Usage of additional Firebase libs

As an example of another Firebase libs usage, the developer can add the Analytics lib (located at: \$(SRCROOT)/Carthage/Build/iOS/) to their project configuration (under Embedded Binaries) and later after the Halo Firebase configure [case 1] (occurring on the Push add-on startup) or their own configure [case 2] (explained on the above example) start making using of the initialized instance of Firebase Analytics.

```
// [case 1] - After the Halo startup
import FirebaseAnalytics

halo.startup(application) { (success) in
    // from this moment the Firebase configuration is assured then is safe to make use of any other Firebase lib
    Analytics.logEvent("event", parameters: nil)
}
```

```
// [case 2] - After an external Firebase configuration
import FirebaseAnalytics

FirebaseApp.configure() // external configure not handled by the Halo
Notifications SDK

Analytics.logEvent("event", parameters: nil)
```

Custom name support for the Firebase config .plist

Additionally you can set a different file name distinct than the Firebase default "GoogleService-Info.plist" by using an additional property keyed "FIREBASE_PLIST_NAME" on the Halo config .plist

```
<key>FIREBASE_PLIST_NAME</key>
<string>GoogleService-Info-Production</string>
```

HALO Social SDK

Adding the dependency

In order to use all the features provided by the HALO Social Framework, its dependency will have to be added to the project. Depending on the tool that it's being used for dependency management, the steps may change.

Carthage

Adding the dependency on the HALO Social Framework is as easy as adding the following line to the `Cartfile` :

```
github "mobgen/halo-social-ios" "2.5.0"
```

After that, performing a `carthage update` all the required resources to use this Framework should be downloaded.

CocoaPods

Similarly, configuring the dependency using CocoaPods is fairly simple:

```
pod 'HaloSocialSDK', '2.5.0'
```

Social API

The social API is the way to sign in with social providers on HALO SDK. Importing this framework will need a valid HALO instance configured with some credentials and the credentials of the network providers you want to import. The HALO Social SDK provides at the moment two integrations with existing social networks:

- Facebook integration. If you want to learn more, please refer to [the detailed documentation \(page 55\)](#)
- Google integration. If you want to learn more, please refer to [the detailed documentation \(page 59\)](#)

Facebook integration notes

You must follow the next instructions to enable the halo social facebook addon.

⚠ Warning: Remember to set `Always Embed Swift Standard Libraries` to Yes

⚠ Warning: You should add a build phase run script to code sign all the embedded frameworks.

```
pushd ${TARGET_BUILD_DIR}/${PRODUCT_NAME}.app/Frameworks/HaloSocialFacebook.framework/Frameworks

for EACH in *.framework; do

echo "-- signing ${EACH}"

/usr/bin/codesign --force --deep --sign "${EXPANDED_CODE_SIGN_IDENTITY}" --entitlements "${TARGET_TEMP_DIR}/${PRODUCT_NAME}.app.xcent"

--timestamp=none $EACH

done

popd
```

Simple usage

Login with a social provider (Facebook or Google)

Once the HALO SDK is started and you add the Social SDK, you can use a social provider to log in.

✓ **Tip:** If you want to learn more about the Facebook integration, please refer to [the detailed documentation \(page 55\)](#).

✓ **Tip:** If you want to learn more about the Google integration, please refer to [the detailed documentation \(page 59\)](#)

If you choose Facebook:

[Swift \(page 52\)](#) [Obj-C \(page 0\)](#)

```
Halo.Manager.auth.loginWithFacebook(viewController: self, stayLoggedIn: false) { (user, error) in
    if error != nil {
        // Something went wrong.
    } else {
        // User logged in successfully. Do something with "user".
    }
}
```

If you choose Google:

[Swift \(page 52\)](#)

[Obj-C \(page 0\)](#)

```
// Your ViewController should implement GIDSignInUIDelegate protocol.
Halo.Manager.auth.loginWithGoogle(uiDelegate: self, stayLoggedIn: false) { (user, error) in
    if error != nil {
        // Something went wrong.
    } else {
        // User logged in successfully. Do something with "user".
    }
}
```

Register

Once the HALO SDK is started, you can use the HaloAuthManager and try to register a new user.

✔ **Tip:** This process only register the user against HALO so you must call to login after registration process finishes correctly.

[Swift \(page 53\)](#)

[Obj-C \(page 0\)](#)

```
import Halo

[...]

// Device should be set after calling startup.
guard let deviceAlias = Halo.Manager.core.device?.alias else {
    // Without a device, you cannot login.
    return
}

let email = "your@email.com"

// set an AuthProfile to register.
let authProfile = AuthProfile(email: email, password: "yoursecre
tpassword", deviceId: deviceAlias)

// set a UserProfile to register.
let userProfile = UserProfile(id: nil, email: email, name: "Your
Name", surname: "YourSurname", displayName: nil, profilePictureU
rl: nil)

Manager.auth.register(authProfile: authProfile, userProfile: use
rProfile) { (userProfile, error) in
    if error != nil {
        // Something went wrong.
    } else {
        // User registered succesfully. Do something with "userProfi
le".
    }
}
```

Logout

Once the HALO SDK is started and user is logged in, you can use theHaloAuthManager and try to logout.

[Swift \(page 54\)](#)[Obj-C \(page 0\)](#)

```
import Halo

[...]

Halo.Manager.auth.logout { success in
    // If success is true, logout is succesful.
}
```


Facebook Integration

This getting started guide will guide you on setting up Facebook SDK for iOS in a few minutes. We will provide a step by step guide to get everything working with the most basic setup.

⚠ Warning: Remember to set Always Embed Swift Standard Libraries to Yes

⚠ Warning: You should add a build phase run script to code sign all the embedded frameworks.

```
pushd ${TARGET_BUILD_DIR}/${PRODUCT_NAME}.app/Frameworks/HaloSocialFacebook.framework/Frameworks

for EACH in *.framework; do

echo "-- signing ${EACH}"

/usr/bin/codesign --force --deep --sign "${EXPANDED_CODE_SIGN_IDENTITY}" --entitlements "${TARGET_TEMP_DIR}/${PRODUCT_NAME}.app.xcent"

--timestamp=none $EACH

done

popd
```

Step 1: Create the app

Register in the facebook console and create a new app. You must have a properly configured developer account.

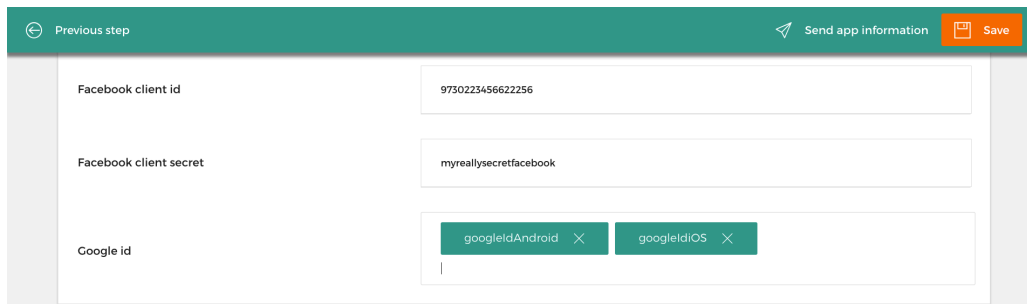
Optional Step to add security

ⓘ Note: You can add extra security if you add the client secret and client id into the Halo CMS.



This step is optional and if you add this field to the HALO CMS it would verify that the tokens you provide belongs to the Facebook application. In another case the HALO system only verifies if it is a valid token against Facebook.

You can add this information in app section on HALO CMS.



Step 2: Add your package

Add the package name and your potential deeplink activity on the facebook console.

Step 3: Generate the hashes

To generate a hash of your release key, run the following command substituting your release key alias and the path to your keystore.

```
keytool -exportcert -alias <RELEASE_KEY_ALIAS> -keystore <RELEASE_KEY_PATH> | openssl sha1 -binary | openssl base64
```

This command should generate a string. Copy and paste this Release Key Hash into your facebook console.

Step 4: Configure the project

Add **Bolts.framework**, **FBSDKCoreKit.framework**, **FBSDKLoginKit.framework**, **FBSDKShareKit.framework**, **FacebookCore.framework**, **FacebookLogin.framework** and **FacebookShare.framework** to your project.

Open your project's *.plist* file as Source Code and insert the following XML snippet into the body of your file just before the final `</dict>` element.

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>fb{your-app-id}</string>
    </array>
  </dict>
</array>
<key>FacebookAppID</key>
<string>{your-app-id}</string>
<key>FacebookDisplayName</key>
<string>{your-app-name}</string>
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>fbapi</string>
  <string>fb-messenger-api</string>
  <string>fbauth2</string>
  <string>fbshareextension</string>
</array>
<key>NSPhotoLibraryUsageDescription</key>
<string>{human-readable reason for photo access}</string>
```

Replace *fb{your-app-id}* with your Facebook app ID, prefixed with *fb*.

Replace *{your-app-id}* with your app ID.

Replace *{your-app-name}* with the **display name** you specified in the App Dashboard.

Replace *{human-readable reason for photo access}* with the reason your app needs photo access.

Step 5: Enable single sign-on

Open the app in the console, open settings and enable “Single sign-on” by setting it to YES. Make sure you save the changes.

Step 6: Register the Facebook Addon

In your AppDelegate, register the Facebook Addon before you call the **startup** method of Halo.

[Swift \(page 58\)](#)[Obj-C \(page 0\)](#)

```
import Halo
import HaloSocialFacebook

[...]

let facebookAddon = FacebookSocialAddon()
Halo.Manager.core.registerAddon(addon: facebookAddon)

[...]

Halo.Manager.core.startup()
```

Step 7: Login with Facebook

Use the **loginWithFacebook** method to login with Facebook.

[Swift \(page 58\)](#)

[Obj-C \(page 0\)](#)

```
import Halo
import HaloSocialFacebook

[...]

Halo.Manager.auth.loginWithFacebook(viewController: self, stayLo
ggedIn: false) { (user, error) in
    if error != nil {
        // Something went wrong.
    } else {
        // User logged in succesfully. Do something with "user".
    }
}
```

Note: For further information about Facebook SDK visit the [official Facebook documentation page \(https://developers.facebook.com/docs/facebook-login/ios\)](https://developers.facebook.com/docs/facebook-login/ios)

Google Integration

This getting started guide will guide you on setting up Google Sign-In SDK for Android in a few minutes. We will provide a step by step guide to get everything working with the most basic setup.

Step 1: Create the app

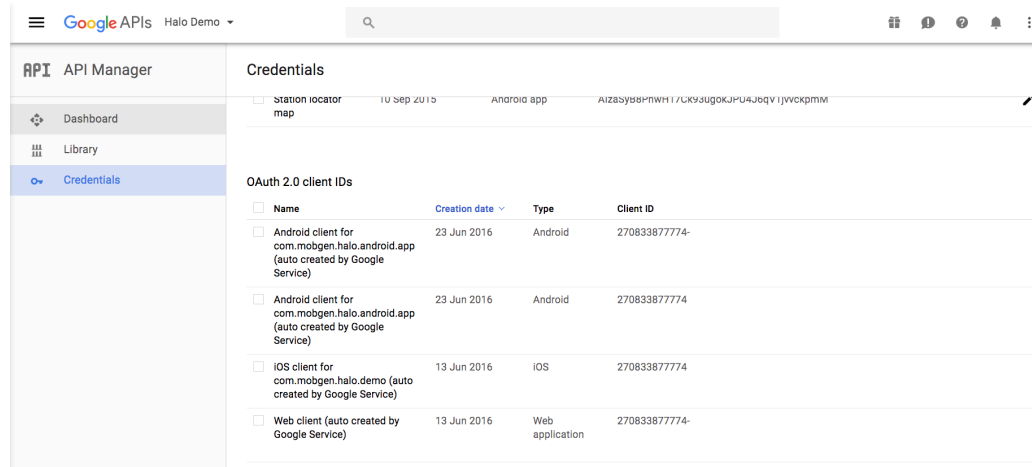
Register in the firebase console and create a new app, if you don't already have one. If you already have an existing Google project associated with your mobile app, click Import Google Project. Otherwise, click Create New Project.

Step 2: Generate an OAuth web key

Add the package name and follow the setup steps.

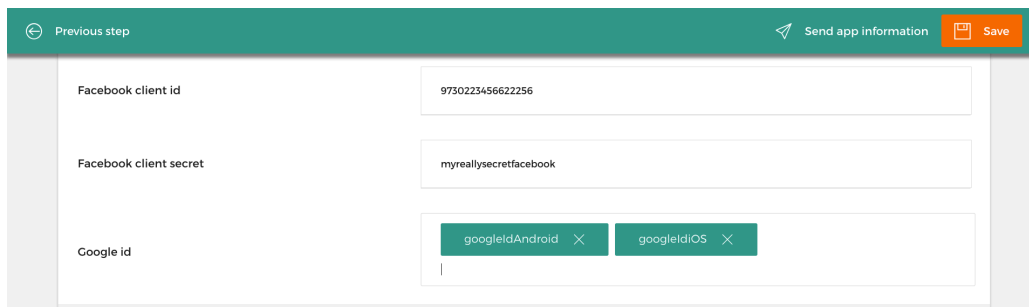
Optional Step to add security

Note: You can add extra security if you add the google id into the Halo CMS.



This step is optional and if you add this field to the HALO CMS it would verify that the tokens you provide belongs to the Google application. You can provide as many ids as you need for different platforms. In another case the HALO system only verifies if it is a valid token against Google.

You can add this information in app section on HALO CMS.



Step 3: Generate the hashes (Optional)

To generate a hash of your release key, run the following command substituting your release key alias and the path to your keystore.

```
keytool -exportcert -alias <RELEASE_KEY_ALIAS> -keystore <RELEASE_KEY_PATH> | openssl sha1 -binary | openssl base64
```

This command should generate a string. Copy and paste this Release Key Hash into your firebase console.

Step 4: Download and setup configuration file

At the end, you'll download a GoogleService-Info.plist. You can download this file again at any time. Copy this file into your project's files.

⚠ Warning: Ensure you have a GoogleService-Info.plist with that exact name on your project, otherwise the Google Social Framework will not work.

Step 5: Configure the project

If you are not using Carthage you will need to add some libraries manually:

- GoogleSignInDependencies.framework
- GoogleSignIn.framework
- GoogleSignIn.bundle

⚠ Warning: You can download the **GoogleSignIn.framework** [official GoogleSignIn documentation page \(https://developers.google.com/identity/sign-in/ios/start-integrating\)](https://developers.google.com/identity/sign-in/ios/start-integrating)

Remove all the libraries from Linked Frameworks and Libraries and Embedded Binaries .

Add the `-ObjC` flag to `Other Linker Settings`. This can be found inside `Build Settings` tab in the `Linking` section.

Open app in **TARGETS**, click on `Info` tab and expand `URL Types` section.

Click on the `+` button and add a new `Url Scheme`. Find the key `REVERSED_CLIENT_ID` inside your `GoogleService-Info.plist` file. Copy the value of this key and paste it into the field `URL Schemes`. Let the other fields empty.

Click again on the `+` button and add a second scheme. Copy the ID of your package `com.example.app` and paste it into `URL Schemes` field. You can find the ID of your package in the **General** tab below **Identity > Bundle Identifier**.

Step 6: Register the Google Addon

In your `AppDelegate`, register the `GoogleAddon` before you call the **startup** method of `Halo`.

[Swift \(page 61\)](#)

[Obj-C \(page 0\)](#)

```
import Halo
import HaloSocialGoogle

[...]

let googleAddon = GoogleSocialAddon()
Halo.Manager.core.registerAddon(addon: googleAddon)

[...]

Halo.Manager.core.startup()
```

Step 7: Log in with Google

Use the `loginWithGoogle` function to log in with Google.

[Swift \(page 62\)](#)

[Obj-C \(page 0\)](#)

```
import Halo
import HaloSocialGoogle

[...]

// Your ViewController should implement GIDSignInUIDelegate protocol.
Halo.Manager.auth.loginWithGoogle(uiDelegate: self, stayLoggedIn: false) { (user, error) in
    if error != nil {
        // Something went wrong.
    } else {
        // User logged in successfully. Do something with "user".
    }
}
```

Note: For further information about Firebase SDK visit the [official Firebase documentation page \(https://firebase.google.com/docs/ios/setup\)](https://firebase.google.com/docs/ios/setup)

iOS Framework - Changelog for 2.4.0

Changelog

Core SDK

- Added support for the sort operation of content instances via the search query
- Implemented the pocket feature to store custom data associated to an identified user
- Added support for rich notifications (notifications with image)
- Other minor bugfixes
- Allow to sync by name
- Allow to change user and app credentials to change environment dynamically
- Fix several bugs.

Notifications SDK

- Added last firebase dependencies
- Fixed an issue where the device wouldn't get a valid push token in the startup process
- Fixed an issue where silent notifications were received as regular ones

Social SDK

- Removed unused Firebase dependencies
- Added new Google and Facebook dependencies
- Fix minor issues

iOS Framework - Changelog for 2.3.1

Changelog

Core SDK

- Make sure the `managerWillSetupDevice` gets called every time the app starts, so that any customisation can be performed by the developer.

Notifications SDK

- Reduced the size of the SDK by improving the size of the sound files for the custom notifications sounds.

iOS Framework - Changelog for 2.3.0

Changelog

Core SDK

- Added a new function to the `CoreManager` to retrieve registered addons by type:

```
public func getAddons<T: Addon>(type: T.Type) -> [T]
```

- Improved logging verbosity
- Other minor fixes

Notifications SDK

- Add a way to request permissions to receive remote notifications manually (intended to delay that step if desired). The constructor of the `NotificationsAddon` accepts now a new parameter (`autoRegister`, `true` by default), which allows to disable the auto-registration for these notifications. After that, one of the following functions would have to be called to request permission to get remote notifications depending on the iOS version:

```
/// iOS 10+
public func registerApplicationForNotificationsWithOptions(
    _ app: UIApplication = UIApplication.shared,
    authOptions options: UNAuthorizationOptions = [.alert, .badge, .sound]) -> Void
```

```
/// Older versions of iOS
public func registerApplicationForNotificationsWithSettings(
    _ app: UIApplication = UIApplication.shared,
    notificationSettings settings: UIUserNotificationSettings =
        UIUserNotificationSettings(types: [.alert, .badge, .sound], categories: nil)) -> Void
```

iOS Framework - Changelog for 2.2.5

Changelog

Core SDK

- Implemented the mechanism to perform batch operations over content instances (create, update and delete). See [Content Edition \(page 30\)](#) section for more details.

iOS Framework - Changelog for 2.2.4

Changelog

Core SDK

- Changed the visibility of two functions within the `HaloAppDelegate`

```
open func application(_ app: UIApplication, open url: URL, o
ptions: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Boo
l
open func application(_ application: UIApplication, open ur
l: URL, sourceApplication: String?, annotation: Any) -> Bool
```

iOS Framework - Changelog for 2.2.3

Changelog

Core SDK

- Implemented the observer pattern in the `AuthManager` (`AuthManagerObserver`) to keep track of the automatic log in

```
@objc
public protocol AuthManagerObserver {
    func userDidLogIn(_ user: User)
    func userDidLogOut()
}
```

Observers can be managed using the new functions in the `AuthManager` :

```
public func addObserver(_ observer: AuthManagerObserver) -> Void
public func removeObserver(_ observer: AuthManagerObserver) -> Void
```

- Added new function to the `ContentManager` to get the search instances result as raw data:

```
func searchAsData(
    query: Halo.SearchQuery,
    completionHandler handler: @escaping (HTTPURLResponse?, Halo.Result<Data>) -> Void
) -> Void
```

- Make publicly available the function in the `ContentManager` to remove the syn logs for a given module

```
public func clearSyncLog(moduleId: String) -> Void
```

- Added a new helper function to the `ContentInstance` to directly set a value for a given key

```
open func setValue(key: String, value: Any?) -> Void
```

- Make sure parsing and other data processing takes place in background

Breaking changes

Core SDK

- Timeouts for the network requests are now configurable via the `NetworkManager` as properties

```
public var timeoutIntervalForRequest: TimeInterval?  
public var timeoutIntervalForResource: TimeInterval?
```

iOS Framework - Changelog for 2.2.2

Changelog

Core SDK

- **New feature:** Content. Added the possibility of using relationships to retrieve related instances (new functions in `SearchQuery`).
- Timeouts for network requests are now configurable from outside the SDK (`timeoutIntervalForRequest` and `timeoutIntervalForResource`).
- Improved logging.
- Minor bugfixes.

Notifications SDK

- Added a new delegate protocol (`TwoFactorAuthenticationDelegate`) to specifically handle two factor authentication push notifications.

Breaking changes

Notifications SDK

- Replaced the three existing methods with only one. Created the `HaloNotification` model, which also contains a `type` helpful when trying to differentiate what kind of push notification the device is receiving (`normal` , `silent` , `twoFactor`).

iOS Framework - Changelog for 2.2.1

Changelog

Core SDK

- Fix issues related to the keychain
- Make sure the user is re-created when not correctly retrieved from the server
- Add a hash to check that the current app credentials match the ones of the stored device
- Give the developer the ability to remove the currently stored device from the keychain

iOS Framework - Changelog for 2.2.0

Changelog

Core SDK

- **New feature:** content creation/modification/deletion through the new functions in the `ContentManager` .
- Added a new `like` operator intended to be used when searching content.
- Added more verbose logging showing also the network responses (when in `info` level).
- Improvements and fixes to the user authentication process

Social SDK

- Release of the HALO Social SDK, providing the implementations of addons to log in using Facebook or Google credentials.

Breaking changes

Core SDK

- Removed some redundant named parameters in some functions.
- New way of logging messages. More flexible and powerful way, by implementing different loggers and registering them within the core.

iOS Framework - Changelog for 2.1.1

Changelog

Core SDK

- Improved error handling.
- Fix a potential crash related to locales and the `TranslationsHelper`.

Breaking changes

No breaking changes

iOS SDK - Changelog for 2.1.0

Changelog

Core SDK

- **New feature:** Created the `AuthManager` to handle all the authentication-related operations. Provide user authentication (HALO users) as part of the core.
- Fix some issues related to `NSCoding` and Swift 3.
- Information related to the app/user is now safely stored in the keychain (added new dependency to `SwiftKeychainWrapper`).
- Extended protocols for add-ons to handle urls.
- Update SSL certificate and enable SSL pinning by default (having also an option to disable it).
- Solve a bug when retrying requests.
- Added more tests and improved code coverage.

Breaking changes

Core SDK

- Changed the configuration and launching process of the core. All the setup should now be done inside the `application:willFinishLaunchingWithOptions:` function.
- Removed some redundant parameter names from some functions.