MyOrder API Framework Developer Guide

MOFramework 0.0.1

1. Introduction

MyOrder Framework (code named as `MOFramework`) is a public iOS framework created by MyOrder to provide external developers with access to the vast amount of features and functionalities delivered by MyOrder. To name a few, it provides code for accessing the whole catalog of more than 11.000 merchants, managing parking tickets or ordering in external services like ThuisBezorgd.nl or ThuisAfgehaald.nl among many others.

The framework has been used in the development of the MyOrder app and other white label apps for external customers. Therefore, any functionality used in the official MyOrder app will be available for external developers to use, giving them the opportunity to develop full apps with their particular business requirements and UI.

It is important to mention that `MOFramework` depends on the MyOrder SDK used for payments, and that it does not contain any UI but a rich set of model objects and DAOs to access the MyOrder backends easily.

The present document is divided in several sections explaining installation and configuration instructions, architecture, all modules provided and some extra appendixes with useful information and code samples.

2. Installation

`MOFramework` can be installed with Cocoapods or manually in your project.

Cocoapods installation

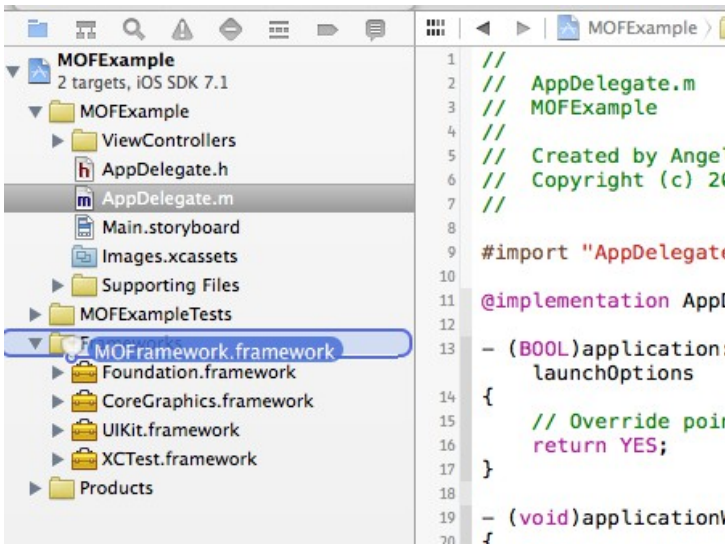
Available soon.

Manual installation

If you prefer a manual installation instead of Cocoapods, then do the following:

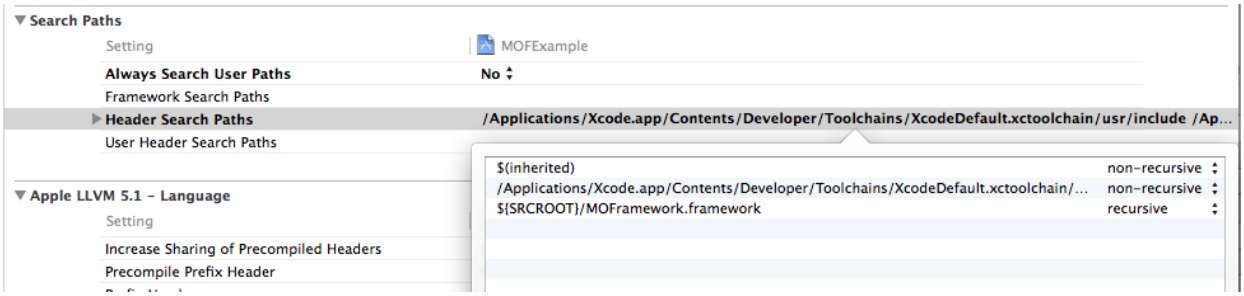
1. Download a copy of the `MOFramework`

2. Drag & Drop the `MOFramework.framework` from Finder into your project (make sure the copy option is selected)



3. Make sure that the following frameworks are added to your project:
    - CoreLocation
    - PassKit

4. Go to your project Build Settings, and in the **"Header Search Paths**" add "`$ {SRCROOT}/MOFramework.framework`" as recursive. Note that if you have changed the location of `MOFFramework.framework` you need to update the previous path accordingly.



5. Install `MyOrderSDK`: The framework uses and requires the `MyOrderSDK` payments library. Check out the official `MyOrderSDK` page for installation instructions on http://myorder.nl/sdk

6. Install the following external dependencies with instructions provided in their webpages:
    - LUKeyChainAccess
    - Reachability

A copy of all external dependencies can be found in the `MOFExample` project, although downloading the code from their original repository is preferred.

3. Configuration

After installing the framework in your project, you still need to do a few steps before starting to work with it. The following code is normally placed in your `appDelegate - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions`, although other locations are possible.

1. **Set the server baseUrl.** Note that different environments exist on a “per client” base. Check with MyOrder what is the proper environment for your application. Example:

```
[MOFNetworkConnection setBaseUrl:@"https://production.myorder.nl/api/v1/"];
```

2. Optionally, **set animation handlers**. An example used in MyOrder app is provided in Appendix B.

```
[MOFNetworkConnection setAnimationHandler:^(MOFNetworkConnection *connection,
MOFNetworkAnimationStatus status) {
   //Add custom UI feedback here for your app
}];
```

3. **Configure the MyOrderSDK** library. More info about how to do this in the official MyOrder documentation page: [http://myorder.nl/sdk.](http://myorder.nl/sdk)

When finished with the previous setup, please note that all DAOs require to be instantiated before their use. Example:

```
[MOFAuthDAO instantiate];
```

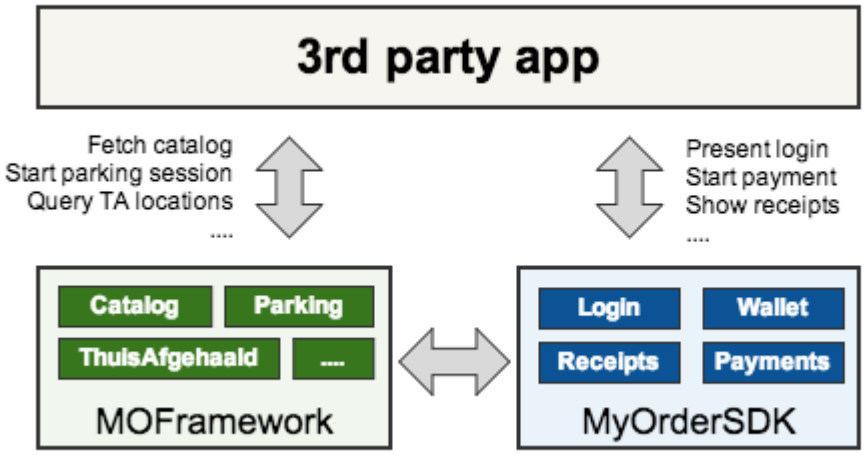
More info about the DAOs and instantiation can be found on next sections.

Check the `MOFExample` project’s `AppDelegate` or Appendix B for a complete setup example.

4. Architecture

Introduction

An app using the MyOrder Framework will have the following architecture:



There are 3 main components:

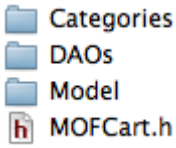
- **3rd party app**: This box refers to the code from the external developer. It contains UI and uses the other 2 components for communicating with the MyOrder environment.

- **MOFramework**: This component is the one explained in this document. It contains all code necessary to communicate with the MyOrder services. The framework is divided into modules (see list of them in following section). Each module provides functionality related to one specific feature of the Framework and they are composed of up to three basic components (**Model**, **DAOs** and **Categories**). All components can be imported with a header file called `"MOFXXX.h"` where `XXX` matches the module name. It is important to mention that the framework does not provide any UI, it only provides data. It is the developer's duty to create any custom UI to use the data depending on the app needs.

- **MyOrderSDK**: This component is an independent library also provided by MyOrder and used to perform payments and manage your wallet. It contains UI and logic for all operations, and it will be the component used for performing the login, starting a payment, checking your wallet or listing your previous receipts. The SDK can be used without the Framework, but not the other way around because the Framework depends on the SDK for the sessions and completing payments. Check out the official MyOrder page for more information at http://myorder.nl/sdk

Framework Components

All modules are composed by a combination of Categories, DAOs and classes. Here is an overview of each:



Model

Model

Model classes in the `MOFramework` are regular objects extending the `MOFModelObject` base class and optionally implementing `NSCoding` protocol. This class provides some basic persistence methods to keep objects in memory, cache or disk, and provides basic implementations for mappers. **You would normally not use any of the following methods**, but they are provided as a reference in case that an extension is needed:

```
+ (BOOL)existsObjectWithIdentifier:(id)identifier;
```
Returns whether an existing object of this type exists or not. Only valid in combination with persistence.

```
+ (instancetype)objectWithIdentifier:(id)identifier;
```
Returns an object with the existing identifier or creates a new one if none exist.

```
+ (NSArray *)objectsWithPredicate:(NSPredicate *)predicate sortedBy:
(NSArray *)sortDescriptors;
```
Returns an array of objects existing in the persistence store passing the predicate provided. Note that depending on the persistence store this method might return no results.

```
- (void)save;
```
Inserts the object into the persistence context if it is not there yet. Automatically called by all the `objectWithIdentifier:` alike methods.

```
- (void)delete;
```
Removes the object from the persistence context. A call to `persist`: is still required if you want to persist changes on disk.

```
+ (void)persist:(BOOL)wait;
```
Persists all unsaved data in the persistence store. This method only does something when used in combination with `MOFModelObjectPersistenceDisk`.

Relations in `MOFModelObjects` are done lazily by the use of categories with primary keys. This means that querying a relation normally results in an object search and should therefore be minimized when possible. Anyway, unlike CoreData based frameworks, most relations live in memory and therefore are very fast.

Finally, an important note to make is that the model objects are shared across threads, so **your code should handle thread safety properly**. **No context switch is required** to move objects between threads as you would normally do with CoreData based frameworks.

DAOs

`MOFModelObjects` should not be used or created directly from third party apps. Instead, the developer should make use of the DAO components. DAOs are very different depending on the modules, but they all extend `MOFBaseDAO` and provide the same basic methods:

`+ (void)instantiate;`
Instantiates the sharedInstance if not created already. Call this method before start using a DAO. If Unit test, use `setSharedInstance` to set a mock instead.

`+ (instancetype)sharedInstance;`
Returns the configured shared instance. Call `instantiate` or `setSharedInstance` before using this method.

DAOs should always be instantiated at least once before its use, normally by calling the `instantiate` method. This method will create a new instance of the DAO if none exists, resulting on the initialization of other properties, notifications or dependencies that the DAO uses. Do not instantiate a DAO if you do not plan to use it, as it could consume extra resources in your app. If you want to use the DAO within a Unit Test, you can make use of the `setShareInstance` method to set the shared instance to the object of your choice.

Apart from the shared methods, all DAOs follow the same naming convention when fetching data from server. An example method could look like:

```
- (MOFNetworkConnection *)loadProductDetail:(MOFProduct *)product
                                  animation:(MOFNetworkAnimation)animation
                                   onResult:(MOFBaseDAOResultBlock)resultBlock
                                    onError:(MOFBaseDAOErrorBlock)errorBlock;
```

The previous example method is part of the `MOFShopDAO` and it is used for fetching a product detail. The parameters it receives are the product that you want to fetch, an animation to use and a result and error block that will be called on case of success or error; it returns the connection made (if any). Lets take a closer look to them:

- `animation`: This parameter allows the developer to easily indicate the animation type to apply. Note that passing an animation type does not mean that any UI will be shown. Instead, you need to provide a callback block on the `+(void)setAnimationHandler:(MOFNetworkAnimationHandler)animationHandler` of the `MOFNetworkConnection` with the proper UI to show for each case. Passing the animation type at this point will be reflected on the `animationHandler` block. An example of the code used for the MyOrder app can be found in Appendix B.

- `onResult`: Contains a result block that will be executed when the data is fetched from the server. `MOFBaseDAO` declares two different result blocks containing either a `NSArray`

response (when multiple results are expected) or an "`id`" response (when a single object is expected). Other DAOs could potentially use different block formats if required, yet they are documented clearly on their header file. For example:

```
onResult:^(MOFProduct *product) {
     [self configureViewWithProduct:product];
}
```

- `onError`: Contains an error block that will be executed in case of any error. Only one of `onSuccess:` or `onError:` will be called, but never both. The error block contains an `NSError` with extended details. An example use of the error block is:

```
onError:^(NSError *error) {
     //Do your own error handling
}];
```

- Returned `(MOFNetworkConnection *)`: Most DAO operations will return an `MOFNetworkConnection` object performing a network operation. When returned, it allows the external developer to cancel it in case the app does not need the data anymore (before receiving the response). Check the `MOFNetworkConnection` documentation for more details about available methods.

All callback blocks are always called on the main thread independently of the original thread where the DAO call was made.

More information about the specific DAOs can be found later in this document and in the HTML code reference included with the framework.

Categories

Besides DAOs and Model objects, some modules can also include extra categories. This categories are normally used to extend the functionality of external objects existing out of the boundaries of the module, but dependent on it. For example, the `MOFCart` module includes a category that extends `MOFProduct` (a model from `MOFShop`) with extra properties to read the amount of products in the actual shopping cart. Check the modules below for more information about existing categories.

5. Modules

`MOFramework` includes a big list of modules to encapsulate all the different features of a MyOrder based app.

Auth

Auth module is one of the most important modules, as it is required for the rest to work properly. Auth provides methods to perform an anonymous login and transform the anonymous session into a user session when there is a login done in the MyOrder SDK. Anonymous sessions are similar in behaviour to HTTP sessions, they are created once at the start of the app and they are used to track the same user across multiple different requests, even if the user is not logged in.
Note that most of the DAOs require an existing session to be created (anonymous or non anonymous) before performing any operation so it is a good practice to always instantiate this DAO.

Model objects provided by the module are:

- `MOFSession`: Contains information about an existing session (anonymous or not), with properties like the creation date associated phone number (if any) or customer Id.

- `MOFUser`: Contains user information like name, email, profile image or addresses.

Operations provided by `MOFAuthDAO` are:

- Get current session
- Login anonymously (create session)
- Logout
- Load user information
- Update user data
- Update user addresses
- Update user profile image

No extra categories are provided by this module.

Shop

Shop is the module used for fetching the MyOrder catalog, with a few exceptions (ThuisAfgehaald, Cinema, Events or ThuisBezorgd). It provides quite a lot of model objects and DAOs and it is used as the base for other catalogs.

Model objects provided by the module are:

- `MOFMerchant`: Contains information about a particular location. A merchant can be understood as a restaurant, a shop or any other kind of location. It contains information

about the type, name, address, website, phone, categories, products, medias, fulfilment methods or schedules. It also add some extra information like the open status, a coupons count or pre-order and post-order information.

- `MOFCategory`: A menu card is composed of categories. Categories can contain other subcategories and/or products. Categories also have information about the merchant, name, details or medias among others.

- `MOFProduct`: Products are the main object in the catalog. They contain information about the parent category, merchant, name, details, medias and prices. Besides, a product can contain product options when it can be composite. For example, a “Burguer Menu” can be composed of 3 options for the burger, the side dish and the drink.

- `MOFProductOption`: As the name suggest, a product option contains information about a specific option for a composite product like the name, minimum/maximum amount of values to chose and a list of values. In the “Burger Menu” example, a product option would be “Kind of Drink”.

- `MOFProductOptionValue`: option values contain information about a specific value for a particular option like the name, price adjustment or medias. In the “Burguer Menu” example, a value for option “Drinks” would be “Coca-cola”.

- `MOFFulfillmentMethod`: Fulfillment method is the model associated to a particular delivery method of a merchant. It includes information about the type, label, price adjustment (if extra cost for a delivery), and method parameters. Note that fulfillment methods are mainly used in the shopping cart, but also included on merchant level to allow apps to filter or display information per merchant.

- `MOFFulfillmentMethodParameter`: Parameters are included in the `extraInfo` of a fulfillment method and are required before proceeding to checkout. They contain information like the name, type, available values, whether it is required or not and the selected value. Selected values must be filled in by the user with information like his name, company, pickup time,...

- `MOFCheckIn`: Temporal object used when performing a checkin action in a merchant.

- `MOFCheckInStatus`: Object containing the status of a checkin (checkin count and checkin on/off).

- `MOFCluster`: Model class used to define aggregations of data (normally merchants) based on location to be displayed in a map. It contains the coordinate and count.

- `MOFCoupon`: Not available yet.

Apart from model classes, the Shop module also includes a `MOFShopDAO` with the following operations:

- Load all merchants based on types, location, search term, post code, city or external id.
- Load merchant clusters (for use in a map) based on types and locations
- Load a merchant detail, including the menu categories and products.
- Load a merchant status (open/close)
- Load all categories based on types or merchants.
- Load a category detail
- Load all products based on types, merchants, term, categories or external ids.
- Load a product detail
- Load the checkin status for a user and merchant
- Perform a checkin from a particular location in a merchant from a user
- Perform a checkout from a particular merchant

No extra categories are provided by this module.

Cart

Cart module is used for managing a shopping basket. It is important to note that the cart is preserved on server side, so as long as the user is logged in with same credentials he will keep the cart synchronized across devices. Shopping cart is also preserved when an anonymous user makes a login.
An important note to make is that a cart can only contain items from one merchant, so adding a product from a different merchant will perform a reset first. Besides, some merchant types like cinemas only allow 1 single product in the cart (you can check it on the `MOFMerchantType` object)

Model objects provided by the module are:

- `MOFOrder`: Order object contains information about a cart. It contains a list of items, prices, associated merchant and fulfillment method among others.

- `MOFOrderItem`: Items contain information about a specific product configuration in the cart. It contains information about the name, quantity, prices and the product. Note that because a product can have options associated, the same product can be multiple times in a cart in different items, each one with different option combinations.

This module makes use of `MOFProduct`, `MOFFulfillmentMethod` and `MOFFulfillmentMethodParameter`. Check the Shop module for more information. It also uses the MyOrderSDK for Payments (more information below).

The module also contains a `MOFCartDAO` with the following operations:

- Reset cart
- Load cart

- Add a new product (with options and quantity)
- Remove a product (all existing items with the product)
- Remove an specific item
- Update an item with new options and/or quantity
- Load all possible values of a fulfillment method parameters
- Select a fulfillment method and values
- Start checkout of cart
- Cancel unfinished checkout

To perform a full checkout, a 3rd party app needs to do the following:

1. Make a login (anonymous checkout is not allowed)
2. If needed, edit cart (add at least 1 product)
3. Ask user for a fulfillment method and all required method parameters
4. Send selected fulfillment method and pass all required values.
5. Start checkout: The result of checkout is a `MOOrder` already configured to be sent to the MyOrder SDK for payment. An example of a checkout would be:

```
[[MOFCartDAO sharedInstance] startCheckoutWithAnimation:MOFNetworkAnimationHUD
 onResult:^(MOOrder *moOrder) {
     //Pays the MOOrder with the MyOrder SDK
     UIViewController *vc = [[MyOrder shared] paymentViewControllerForOrder:moOrder];
     [self.navigationController pushViewController:vc animated:YES];
 }
 onError:^(NSError *error) {
     //Handle error if custom handling wanted
 }];
```

After successfully starting a checkout the order is moved to a submitted status, waiting for payment, and do not allow any change. If the payment does not succeed, it is responsibility of the developer to call the cancel method to revert it back into a valid shopping cart order. More information on how to detect errors, cancellations, etc. during a payment can be found in the MyOrderSDK documentation.

Besides, the module provides the following categories:

- `MOFProduct+Cart`: Adds dynamic properties to easily fetch the items associated to a particular product.

Cinema

Cinema module is the one used for managing merchants of type “cinema” (`kMOFMerchantTypeCinema`).
Note that cinemas are like any other merchant, but its catalog is specialized and they provide a few extra operations and models. For example, in a cinema merchant the movies are represented by categories, dates are subcategories and times are the products. This whole hierarchy is abstracted by the `MOFMovie` model object. Therefore, even if you can use the Shop module for cinema, we

strongly recommend to use this module instead for making its use easier. Times, as any other `MOFProduct`, can be added to the cart.

Model objects provided by the module are:

- `MOFMovie`: This model agglutinates categories, subcategories and products related to a movie in an abstract model. It provides simple accessors for fetching the movie details, associated merchants, dates and times.

The module also contains a `MOFCinemaDAO` with the following operations:

- Load all movies based on merchants
- Load a movie detail

No extra categories are provided by this module.

Events

Events module is the one used for managing merchants of type “events” (`kMOFMerchantTypeEvents`).
Note that events are like any other merchant, but its catalog is specialized and they provide a few extra operations and models. For example, in an event merchant the events are represented by categories, and then the location, date and time is part of a the product name, while the specific ticket seat is a product option. This whole hierarchy is abstracted by the `MOFEvent` model object. Therefore, even if you can use the Shop module for events, we strongly recommend to use this module instead for making its use easier. As any other shop product, an event together with the specific seat (option) can be added to the cart.

Model objects provided by the module are:

- `MOFEvent`: This model agglutinates categories, products and options related to an event in an abstract model. It provides simple accessors for fetching the cities, dates, times and products associated to an event.

No extra DAOs are provided by the module.

The module provides the following extra categories:

- `MOFProduct+MOFEvents`: Provides accessors for fetching the name, city, date and time of an event in a product level. Note that this category requires the use of `MOFEvent` before its use on the product.

Favorites

Favorite module allows to save any `MOFModelObject` into a favorites list stored in the device's file system. Note that in the current version this module does not have any synchronization with server data, but it will provide it in future releases for `MOFMerchant` and `MOFProduct` objects under the same API available nowadays. Because of that, we strongly suggest to avoid saving any object other than the previous two, even if the current version supports it.

Model objects provided by the module are:

- `MOFFavorite`: This model object is used internally by the framework to store an associated object in disk. As a third party developer you should not use this model directly but rely on the provided DAO methods.

The module also contains a `MOFFavoritesDAO` with the following operations:

- Get all loaded favorites
- Check if an object is favorite
- Set an object as favorite
- Load all favorites from server (no op in current version)
- Save all favorites on server (no op in current version)

The module also provides the following extra categories:

- `MOFModelObject+Favorites`: Provides an easy way to fetch all favorited objects of a particular model type and check if an instance is favorited.

Parking

The parking module is used to perform parking operations like check an existing session, stop or extend it or create a new session. Note that all methods from this module require the user to be logged in as a valid MyOrder user, so make sure to be logged in before its use.

Model objects provided by the module are:

- `MOFParkingSession`: This model class is responsible of providing all information for an existing (or new) parking session. It includes address, end time, license plate, price, point id or status among some others.

- `MOFParkingPoint`: This model object represents a particular parking point and it is composed of an id (parking point number shown on parkimeters) and an address.

The module also contains a `MOFParkingDAO` with the following operations:

- Load all parking points for a particular location
- Load existing parking session
- Create a new parking session based on location/parking id and date
- Extend an existing parking session to a new date

- Stop an existing parking session
- Checkout a parking session with a license plate

To perform a full checkout, the 3rd party app needs to do the following flow:

1. Make login
2. Check no session already exists (if so, then use the extend method instead of create in 3.)
3. Create a parking session by location or parking point
4. Start checkout: The result of checkout is a `MOOrder` already configured to be sent to the MyOrder SDK for payment. An example of a checkout would be:

```
[[MOFParkingDAO sharedInstance] checkoutParkingSession:session
 licensePlate:licensePlate
   animation:MOFNetworkAnimationHUD
    onResult:^(MOOrder *order) {
        //Pays the MOOrder with the MyOrder SDK
        UIViewController *vc = [[MyOrder shared] paymentViewControllerForOrder:order];
        [self.navigationController pushViewController:vc animated:YES];
    }
    onError:^(NSError *error) {
        //Handle error if custom handling wanted
    }];
```

After successfully starting a checkout the order is passed to the MyOrderSDK. More information on how to detect errors, cancellations, etc. during a payment can be found in the MyOrderSDK documentation.

No extra categories are provided by this module.

ThuisBezorgd

ThuisBezorg module is used to query the external service http://www.thuisbezorgd.nl/. MyOrder provides a module abstracting the particular details of this external provider into objects equivalent to the ones provided by the Shop module. In fact, no models are provided by the module itself because it uses the Shop ones.

DAOs provided by this module are a bit particular, as they extend the existing `MOFShopDAO` and `MOFCartDAO` adding additional methods and intercepting the calls appropriately when the merchant is of type “thuisbezorgd“ (`kMOFMerchantTypeThuisBezorgd`). This means that when you instantiate the `MOFTBShopDAO` or `MOFTBCartDAO` they will replace the shared instance of `MOFShopDAO` and `MOFCartDAO` by the TB version, and will pass all operations to the original one if it is not a TB object. All this is done transparently for the external developer when the TB DAOs are instantiated so you can use the regular Shop and Cart DAOs or keep using the TB counterpart with the exact same results.

Besides the commented peculiarity of TB DAOs, `MOFTBShopDAO` also provide extra operations not applicable to other Shop merchants:

- Load a merchant detail with a delivery location or postcode

Also, note that check-in and favorite operations are not supported by TB merchants and products.

No extra categories are provided by this module.

ThuisAfgehaald

ThuisAfgehaald module is used to query the external service http://www.thuisafgehaald.nl/. This module requires the user to be logged in as it tries to use the credentials previously provided to login on TA service. If the user can not be logged in in TA automatically, and error will be returned in all endpoints until it successfully logs in TA with the corresponding login method in the DAO. It is responsibility of the app developer to provide UI to perform the login on this service.

TA data is very different to the existing catalog of merchants and products, and because of that a whole new set of model classes are provided:

- `MOFTAMeal`: This model class represents a meal in TA. It provides information about the category, details, image, place, price, title or cook among others.

- `MOFTACook`: This model represents a cook in TA. It provides information like name, details, followers, location, meals,...

- `MOFTAThank`: This class represents a “thanks” given in TA to a cook (similar in practice to a review comment). It provides the comment, date, name of commentator and image.

- `MOFTAPickUp`: This class contains information of how many portions, date, etc can be pickup of a particular meal. It is mainly used during ordering.

The module also contains a `MOFThuisAfgehaaldDAO` with the following operations:

- Login in TA service
- Load all meals based on location
- Load a meal detail
- Load a cook detail
- Follow/unfollow cook
- Request a TA meal with a quantity, comment and date
- Pickup a TA meal with a quantity and comment

Note that checkout logic is different in TA than other modules. There is no payment involved and the user can either request a new meal or ask for pickup and existing portion (if available portions exist).

TA service will notify updates back to the app by the use of a push notification with the following information (it is your responsibility as developer to handle the TA notifications properly if desired):

```
{
        "S", "TA",
        "T", "CON",
        "I": "123452"
}
```

Where:

- "S" is the source, this is "TA" for Home picked
- "T" is the type, this is for home picked: "REM", "CON" or "NEW"
- "I" is the ID of a Meal or Order depending on the type push message.

`REM` = REMINDER, show voucher with order Id received
`CON` = CONFIRMATION, show voucher with order Id received
`NEW` = NEW meal, show meal detail screen with meal Id received

No extra categories are provided by this module.

Stories

The Stories module is meant to be used with user data like tickets, custom offers,... In current version, the only functionality provided is for Tickets, but more will come in future releases. This module requires a logged in user as it serves custom data per user.

Model objects provided by the module are:

- `MOFTicket`: Contains information about a ticket purchased by the user. When a user buys an item in MyOrder, some merchants like cinemas or events will prefer to provide a ticket besides the standard receipt. This class has properties for the date, merchant, summary, time, code and barcode.

The module also contains a `MOFTicketsDAO` with the following operations:

- Load all tickets
- Load a ticket detail
- Mark a ticket as used

No extra categories are provided by this module.

Generic

Generic is a special module that agglutinates other cross app models and DAOs that do not belong to any of the other defined modules and that have not enough entity to constitute a new module by themselves.

Model objects provided by the module are:

- `MOFMerchantType`: This class represents a section in the app. There are several merchant types provided, all of them corresponding a particular feature in MOFramework. List of current types are “shop”, “cinema”, “events”, “offers”, “parking”, “thuisafgehaald” and “thuisbezorgd”. New types might be added in future. Besides the type, this class also contains some extra useful information like the name, icon image, sorting order, associated disclaimer title and body, count of merchants around or unique tag among others.

- `MOFAddress`: This class is used by other entities in other modules and contains information about an address, like name, city, street, house number, postcode or location.

- `MOFMedia`: Model object containing information about a media element (photo or image), normally attached to merchants, categories, products and cart items.

- `MOFSchedule`: Model containing information about an opening schedule. This information is normally used with a merchant, but other future uses might be added. It contains a title, start time, end time, week day and open/close status.

- `MOFBanner`: Model class used to define a banner. A banner can contain some amounts, dates, credits, image and are normally associated to a merchant or products.

The module also contains a few DAOS:

`MOFCoreDAO`: This DAO contains core operations like:
- Load all merchant types based on a location.
- Load all available cities based on merchant types and term.
- Make a version check to force/suggest users to update or show them some message

`MOFNotificationsDAO`: This DAO contains operations for configuring APNS:
- Register for APNS with a APNS token
- Unregister from APNS

`MOFPassbookDAO`: This DAO contains operations for dealing with Passbook:
- Load a Passbook associated to a `MOFTicket`
- Load a Passbook associated to a `MOFOrder`

`MOFBannersDAO`: This DAO contains operations for loading banners.

No extra categories are provided by this module.

APPENDIX A: Endpoints used

All endpoints used for the Framework are documented in an online page at:

http://docs.myorderplaygroundrestapi.apiary.io/

APPENDIX B: Code samples

MOFNetworkConnection animation handlers

The following code is used on MyOrder app for setting the animation handler of the `MOFNetworkConnection`. The code assumes the existence of a “toast” UI component called `MOAToastView` and makes use of `MOProgressHUD`, provided by the MyOrder SDK, for the HUD indicator. You could of course make use of different UI components depending on your app requirements.

```
[MOFNetworkConnection setAnimationHandler:^(MOFNetworkConnection *connection,
MOFNetworkAnimationStatus status) {
        if (connection.animation == MOFNetworkAnimationToast) {
            switch (status) {
                case MOFNetworkAnimationStatusStart:
                    [MOAToastView toastMessage:NSLocalizedString(@"downloading-
information",nil) duration:MOAToastViewDurationInfinite style:MOAToastViewStyleLoading
tag:-20];
                    break;
                case MOFNetworkAnimationStatusError:
                    [MOAToastView toastMessage:[connection.mapperError
localizedDescription] duration:MOAToastViewDurationLong style:MOAToastViewStyleError
tag:-21];
                case MOFNetworkAnimationStatusSuccess:
                    [MOAToastView hideToastWithTag:-20];
                    break;
            }
        }
        else {
            switch (status) {
                case MOFNetworkAnimationStatusStart:
                    if (connection.animation == MOFNetworkAnimationHUD ||
connection.animation == MOFNetworkAnimationHUDWithConfirmation) {
                        [MOProgressHUD show];
                    }
                    break;
```

```
                case MOFNetworkAnimationStatusError:
                    [MOProgressHUD showError:connection.mapperError];
                    break;

                case MOFNetworkAnimationStatusSuccess:
                    if (connection.animation == MOFNetworkAnimationHUDWithConfirmation
{
                        [MOProgressHUD
showSuccessWithStatus:connection.successMessage];
                    }
                    else {
                        [MOProgressHUD dismiss];
                    }
                    break;
            }
        }
    }];
```

MOFCartDAO checkout process

The following code shows how to make a checkout and pay for an order. It assumes that the cart is already filled in with some products and that the user has been requested to select a fulfillment method and fill in all its extra info values.

1. Select fulfillment method

```
//Set the selected fulfillment options
[[MOFCartDAO sharedInstance] selectFulfillmentMethod:fulfillmentMethod
 animation:MOFNetworkAnimationHUD
  onResult:^(MOFOrder *order) {
      //Continue to checkout
      [self checkoutWithPaymentMethod:@"MiniTix"];
} onError:^(NSError *error) {
      //Custom error handling if desired
}];
```

The previous code selects the fulfilment method and sends all values to the server. All fulfillment values should be already filled in in the `fulfillmentMethod.extraInfo` options, by the use of the `selectedValue` property. Note that some `MOFFulfillmentMethodParameter` might be optional, and types `kMOFFulfillmentMethodParameterTypeTimeBlock` and `kMOFFulfillmentMethodParameterTypeBlockcan` can only contain a preset list of values (available in the `values` property after call to `loadFulfillmentMethodValues:`). Dates are converted into strings of format `HH:MM` using the `dateToValue:` method of the `MOFFulfillmentMethodParameter`.

When fulfillment has been selected, the order might be updated with a new price under very unusual circumstances (for example in delivery if postalcode is very far away). On success, you can proceed to checkout.

2. Start checkout

```
[[MOFCartDAO sharedInstance] startCheckoutWithAnimation:MOFNetworkAnimationHUD
 onResult:^(MOOrder *moOrder) {
    //MOOrder is a MyOrderSDK order ready for payment
    UIViewController *vc = [[MyOrder shared] paymentViewControllerForOrder:moOrder];
    [self.navigationController pushViewController:vc animated:YES];
} onError:^(NSError *error) {
   //Custom error handling if desired
}];
```

The previous code shows how to start a checkout. Note that the result of the checkout call is a `MOOrder`, not a `MOFOrder`. The difference is that `MOOrder` is a MyOrder payment SDK version ready to be sent to the payment SDK for checkout. In the previous example the payment is using the simple method `paymentViewControllerForOrder:`, but a most advanced one with the use of `transactionViewControllerForProvider:` could be used (providing callbacks for error, cancellation, success,...). Check the MyOrder Payment SDK documentation for more information.

When the user completes the payment, the fulfillment will automatically start, so no extra communication is required from the app. Also note that starting the checkout moves the cart to a submitted state, thus not allowing any further change. If you want to change it back to a cart, you need to call the `cancelCheckoutOfOrder:` method, but only if the order has not been paid yet.

MOFExample's AppDelegate

The following code shows an example configuration of `MOFramework` and MyOrder SDK used in the `MOFExample` project. It configures the SDK and `MOFramework` to use the Playground environment. You might use it as template but please check the Configuration section for more details.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions
{
    // MyOrder SDK setup
    MyOrder *myOrder = [MyOrder shared];
    myOrder.apiKey = @"36bd8913-bf56-4aa0-9492-49a3240597ea";
    myOrder.apiSecret = @"12H@c9kT$At";
    myOrder.URLScheme = @"mof-example";
    myOrder.environment = MyOrderEnvironmentPlayground;

    //MOFramework setup
    [MOFNetworkConnection setBaseUrl:@"http://playground-java.myorder.nl/api/v1/"];
```

```
    [MOFNetworkConnection setAnimationHandler:^(MOFNetworkConnection *connection,
MOFNetworkAnimationStatus status) {
        switch (status) {
            case MOFNetworkAnimationStatusStart:
                if (connection.animation == MOFNetworkAnimationHUD ||
connection.animation == MOFNetworkAnimationHUDWithConfirmation) {
                    [MOProgressHUD show];
                }
                break;

            case MOFNetworkAnimationStatusError:
                [MOProgressHUD showError:connection.mapperError];
                break;

            case MOFNetworkAnimationStatusSuccess:
                if (connection.animation == MOFNetworkAnimationHUDWithConfirmation) {
                    [MOProgressHUD showSuccessWithStatus:connection.successMessage];
                }
                else {
                    [MOProgressHUD dismiss];
                }
                break;
        }
    }];

    //Instantiate desired modules. For this demo only the following:
    [MOFAuthDAO instantiate];
    [MOFShopDAO instantiate];

    return YES;
}

// Handle the iDeal redirects
- (BOOL)application:(UIApplication *)application openURL:(NSURL *)url
sourceApplication:(NSString *)sourceApplication annotation:(id)annotation {
    return [[MyOrder shared] handleURL:url];
}
```