MyOrder API Framework Developer Guide

MOApiFramework 1.0.0

1. Introduction

MyApiOrder Framework (code named as `MOApiFramework`) is a public Android 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 `MOApiFramework` 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

`MOApiFramework` Jar file can be to the lib folder of the project. This framework has dependency with jackson and apache http lib which is compiled with different package name to support the http caching.

You can find all dependencies files in “dependencies” folder.

3. Configuration

After coping jar file into your project, you still need to do a few steps before starting to work with it. The following code is normally placed in your `Main Activity` -

```
ApiManager.init(SERVER_URL, getApplicationContext(),
FOURSQUARE_CLIENT_ID,FOURSQUARE_CLIENT_SECRET);
```
, although other locations are possible.

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

```
SERVER_URL = "https://production.myorder.nl/api/v1/"];
```

2. **`FOURSQUARE_CLIENT_ID`** `and` **`FOURSQUARE_CLIENT_SECRET`**. are used to get the merchant images from foursquare api.

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

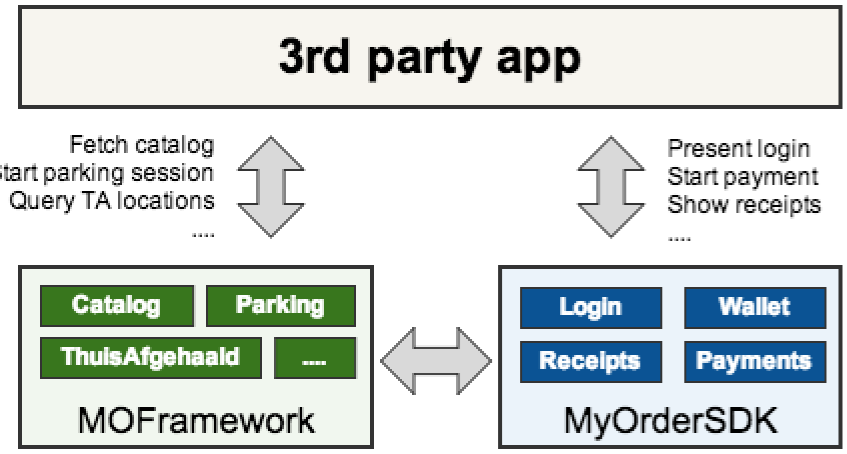
When finished with the previous setup then we have all the DAO initialized.

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

4. Architecture

Introduction

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



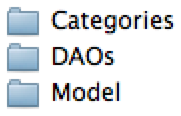
There are 3 main components:

- **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.

- **MOApiFramework**: 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**). 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


Categories
DAOs
Model

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

Model

Model classes in the `MOApiFramework` are regular objects extending the `BaseEntity` base class. **You would normally not use any of the following methods**, but they are provided as a reference in case that an extension is needed:

`ContentValues getDbContentValues();`
Returns ContentValues for existing object which can be used to store object data into database.

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:

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

```
ApiManager.getInstance().getMerchantDAO().getMerchantDetail(merchantId, true, true,
dataLoadListener)
```

The previous example method is part of the `MerchantDAO` and it is used for fetching a merchant detail. The parameters it receives are the merchantId that you want to fetch merchant for, boolean param includeCategories to include merchant categroies into details, boolean param includeProducts to include merchant product into details, and a DataLoadListener that will be called on case of success or error; it returns the connection made (if any). Lets take a closer look to them:

- `DataLoadListener.`: This is used to make a asynchronous call to the server, this listener include two methods. `onDataload` this method will be called once successful response has be received from the server or all data processing is done successfully. And method parameter will contains a result Object that will be executed when the data is fetched from the server. `OnError`: this method will called once an error is occured during the data processing or server has returned the error response. Only one of `onDataload:` or `OnError:` will be called but never both.

- Returned `(DataLoader *)`: Most DAO operations will return an `DataLoader(AsyncTask)` 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).

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 `ShoppingCartDAO` includes a category `Product` (a model from `CatalogDAO`) 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

`MOApiFramework` 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 perform login.

Model objects provided by the module are:

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

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

Operations provided by `AuthDAO` are:

- Login
- 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:

- `Merchant`: 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.

- `Category`: 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.

- `Product`: 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.

- `ProductOption`: 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".

- `ProductOptionValue`: 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".

- `FulfillmentMethod`: 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.

- `FulfillmentExtraInfo`: 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,...

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

- `Cluster`: 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.

- `Coupon`: Not available yet.

Apart from model classes, the Shop module also includes some DAO's `CatalogDAO`, `CinemaDAO`,`FavouriteDAO`, `MerchantDAO` etc.
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

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 `MerchantType` object)

Model objects provided by the module are:

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

- `OrderItem`: 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 `Product`, `FulfillmentMethod` and `FulfillmentExtraInfo`. Check the Shop module for more information.(more information below).

The module also contains a `ShoppingCartDAO` 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 `Order` already has the values which are required for payment SDK. An example of a checkout would be:

```
ApiManager.getInstance().getShoppingCartDAO().checkoutShoppingCart(prepareFulfillmentExtr
aInfo(), new DataLoadListener() {

                      @Override
                      public void onError(Object errorObj) {

                      }

                      @Override
                      public void onDataLoad(Object result) {
                        if (result instanceof Order) {
                             Order orderObj = (Order) result;
                             MOOrder moOrder = new MOOrder();
                             moOrder.setOrderId(orderObj.getOrderNumber());
                            moOrder.setExternalOrderId(orderObj.getId());
                           moOrder.setOrderItems(new ArrayList<OrderItem>());
                           moOrder.setPrice(totalAmount);
                          //Send this Object to MyOrderPaymentSDK for starting the
payment
                        }
                      }
               });
```

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:

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

Cinema

Currently not implemented

Favorites

Favorite module allows to save any `Merchant and Product` model 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 `Merchant` and `Product` objects under the same API available nowadays.

Model objects provided by the module are:

- `Favorite`: 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 `FavoritesDAO` 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)

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:

- `ParkingSession`: 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.

The module also contains a `ParkingSessionDAO` 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

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: Create `MOOrder` object and send to the MyOrder SDK for payment. An example of a checkout would be:

```
MOOrder order = new MOOrder();
Double totalPrice = parking.getPrice().getAmount() +
parking.getTransactionPrice().getAmount();
order.setPrice(totalPrice);
order.setExternalOrderId(parking.getId());
order.setOrderId(parking.getId());
order.setOrderItems(null);
```

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 `TBRestaurantsDAO`.When merchant is of type “thuisbezorgd“ (`MerchantTypeEnum.TB`). This means that when you need the merchant details or catalog for thuisbezorgd you need to call this `TBRestaurantsDAO` . For external developer when the TB DAOs are instantiated so you can use the regular Cart DAOs or keep using the TB counterpart with the exact same results.

- Load a merchant detail with a delivery location or postcode
- Load merchants

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:

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

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

- `TAThank`: 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.

- `PickUp`: 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 term, location and postCode
- Load a meal detail
- Load a cook detail
- Follow/unfollow cook
- Request a TA meal with a quantity, commends and date
- Pickup a TA meal with a quantity and comment

Note ThuisAfgehaald **checkout is currently not implemented**

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:

- `Ticket`: 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 `TicketsDAO` 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:

- `MerchantType`: 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.

- `Address`: 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.

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

- `Schedule`: 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.

The module also contains a few DAOS:

`NotificationsDAO`: This DAO contains operations for configuring GCM:

- Register for GCM with a GCM token
- Unregister from GCM

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

For sample code please checkout `MOExample` added to framework.