Classes

# Client

## Utility classes

Most of these classes are a collection of static functions used throughout that app.

* **BitmapUtils** – contains function(s) for dealing with bitmaps
* **GlideUtils** – contains function(s) for working with Glide (image processing library)
* **InputValidationUtils** – contains function(s) for validating user input using Regex patterns
* **ListUtils** – contains some custom utility functions and various high level list functions that aren’t available for the minimum SDK the app can be deployed on as such needed custom implementation.
* **UIUtils** - contains function(s) for dealing with UI components such as getting cleaned text from a text field and hiding elements.
* **RequestCodes** – this class contains codes for the activity result and permission result requests the app will make.

## Local Data Stores

These classes contain functionality of saving and retrieving local data. Some of them deal with persisted data and some only save to RAM.

* **DataSets** – a singleton class that deals with data kept only in RAM. It includes a reference to the currently logged in user (of class User) and a list of all categories products can be assigned to (list of class ProductCategory). Both these values are received via a login response (through 3 different endpoints). The references to these objects never change, only the internal members do.

This is used throughout the app to get user information and to display list of categories.

* **InitializedStore** – an abstract superclass containing the infrastructure for singleton classes that need initialization before use.
* **TokenStore**, a subclass of InitializedStore – this singleton class deals with persisting the jwt token received via server request to SharedPreferences.

This token is used in most server requests and received from most server responses (only request/responses where the token is unknown will leave it out such as registering)

As the token may refresh at any request (as a security measure), it is returned in server responses and persisted locally.

This token is used to identify the user for most server request and allows such things as automatic login for a user who has logged in in the past and hasn’t logged out.

* **CartTransactable** – an interface describing the basic functionality of the cart.
* **CartDB**, a subclass of SQLiteOpenHelper and implements CartTransactable – this class deals with persisting the cart to SQLite store locally. It saves pairs of product id’s and quantities for them. These pairs are described in the class CartProduct. This class contains various standard database transactions such as insertion, updating, deletion and selection.
* **CartStore**, a subclass of InitializedStore and implements CartTransactable. This singleton class acts as the public API for the cart transactions for the rest of the app. It holds a reference to an instance of CartDB and performs the various operations through it with some adjustments which increase ease of use.

## Generic Events

These classes allow for defining event with any type of fire method. These classes are used to listen to activity request and permission request results.

* **Event** – this class describes a generic event and contains a list of listeners to said event (of some implementation of EventListener). In addition, a generic fire method is provided which notifies all listeners.
* **EventListener** – an interface describing the minimum for being an even listener. It contains a fire method, which the observed event will call when it fires.
* **OnActivityResultListener, OnRequestPermissionResultListener** – two abstract classes that implement the EventListener interface. Each provides a method corresponding to the life cycle event methods onActivtyResult and onRequestPermissionResult belonging to an Activity class. The single Activity class that this app contains, ParentActivity contains Event instances for each and first each even accordingly. These are used in the app when requesting permission for storage and/or camera access and when receiving the result of choosing a picture from storage/taking a picture via the camera.

## Functional Interfaces

These interfaces expand on some utility interfaces that the Android SDK provides us. As the minimum SDK for the app’s deployment bars the use of some of the newer ones, these required manual implementation. These are used as utility interfaces for lambda expressions throughout the app in addition to those provided by the Android SDK.

* **BiFunction** – an interface for a generic function which accepts two input parameters and returns an output.
* **TriConsumer** – an interface for a generic function that accepts three input parameters and has no output.

## UI Classes

These deal with various aspects of the UI of the application and their logic. This category contains types such as Activities, Fragments, Popups (Dialogs), List Adapter for collection layouts, and custom Views, along with a helper enum.

### Overarching UI classes

* **ParentActivity**, a subclass of AppCompatActivity – the only activity class the app contains. It corresponds to the layout *activity\_parent*.

Visually it is comprised of two sections. At the top there is a tab bar for navigation through the different screens of the app. The rest of the screen is taken up by the fragment that is currently being displayed.

Functionally it has several important components. It houses the fragments that display the various screens of the app. It contains methods for transitioning through the different fragments which is done primarily though the tab bar (though not exclusively), methods for requesting permission and getting activity results (as stated above, used for getting images). It initializes all local data stores and holds an instance of the loader popup that can be displayed while making server requests. These as well as some other utility methods like making the local data changes when logging in or out a user and changing the state of the tab bar depending on if the user has entered admin mode or not (which will be discussed in a different section).

Most UI components of the app hold a reference in one form or another to the instance of ParentActivity so as to allow access to its various methods.

* **AppScreen** – an enum containing the different screens that the app can show. Each of these corresponds to a specific fragment which will be hosted in the single Activity that the app contains (ParentActivity). Each enum member contains an instance of its corresponding fragment. These fragment instances are used for navigation within the app. This is done within the methods of ParentActivity.

I make a distinction between two different uses of fragments: screen fragments, that is, fragments which have the job of being an entire screen of the app, and area fragment, that is, fragments that represent a logic and encapsulated section UI that will generally be only one part of a single screen.

### Screen Fragments

* **BaseFragment**, an abstract subclass of Fragment – this is the super class of all screen fragments. It includes a utility method getting an instance of ParentActivity (as it is the activity that creates them all), and a default setting for the tab bar visibility.
* **SplashFragment**, a subclass of BaseFragment – a fragment that is the first one shown when the app is opened. It is responsible for attempting to auto-login the user if a token is stored in the TokenStore and if not or if the auto-login fails, the login popup is displayed.

Corresponds to the layout *fragment\_splash*.

* **RegisterFragment**, a subclass of BaseFragment – a fragment for registering for a new user. In the fragment the user can enter the info and after successful validation, the information will be sent to the server for storing and the user will log in and received a token.

This fragment uses the area fragment UserInfoFragment for the user details form.

Corresponds to the layout *fragment\_register*.

* **ProfileFragment**, a subclass of BaseFragment – a fragment where the user and view and update their information. For a user who is an admin, the option to enter admin mode is also there. Upon entering admin mode, the top bar display will change and various other screens will change accordingly.

This fragment uses the area fragment UserInfoFragment for the user details form.

Corresponds to the layout *fragment\_profile*.

* **CartFragment**, a subclass of BaseFragment – a fragment where the user and view’s the contents of their cart, remove items, change quantities, clear the cart, or proceed to checkout using the popup class CheckoutPopup.

The fragment loads all products in the cart SQLite database using CartStore and retrieves, via server request, the products that correspond to the ids saved in the local database.

This fragment displays a collection of products using a RecyclerView coupled with the list adapter CartProductsAdapter. Further details about a product can be accessed by clicking on it and the ProductDetailsPopup popup class will be used to display that information. When in admin mode, the tab to access this fragment isn’t visible.

Corresponds to the layout *fragment\_cart*.

* **ProductsFragment**, a subclass of BaseFragment – a fragment where the user can browse the active products in the store and add them to their cart. The fragment supplies the ability to filter the products by making use of the ProductFilter model coupled with the popup class FilterProductsPopup. The products are displayed using a RecyclerView along with the list adapter class CatalogProductsAdapter. Further details about an individual product can be accessed by clicking the product. When in normal mode the popup class ProductDetailsPopup will be used.

When in admin mode, this fragment shows the products without quantities as they will not be added to a cart. Clicking on a product will use the popup class ProductDetailsAdminPopup to allow the user to edit or deactivate the product. In addition, in this mode, a button for creating a new product will be visible at the top of the screen, clicking on which will also use the ProductDetailsAdminPopup to allow the user to enter details and create a new product.

In both cases, the products displayed (filtered or not) are pages. The client will load more products from the server when the user scrolls to the bottom of the existing products.

Corresponds to the layout *fragment\_products*.

* **OrdersFragment**, a subclass of BaseFragment – a fragment where the user can browse their past orders. When in admin mode, a text field at the top is visible where the user can enter the id of any user and search for that user’s past orders. The orders displayed use a RecyclerView along with the list adapter class OrderSummariesAdapter (which uses the model class ExpandableOrder which is instantiated using the OrderSummary model class and adds a boolean for if the instance is expanded or not). Clicking an order opens the item up further for some more details and clicking the details button opens the popup class OrderDetailsPopup to show more details including the products that order consisted of.

The orders in the fragment are pages, more are loaded when the user scrolls to the bottom of the current displayed orders.

Corresponds to the layout *fragment\_orders*.

* **AdminFragment**, a subclass of BaseFragment – a fragment only accessible when admin mode is active. This screen is divided into two section, each governed by its own area fragment. The first section allows the user to register a new user, allowing that user to be be registered as an admin themselves. The section uses the AdminCreateUserFragment class. The second section allows the use to invalidate a token for a user whose id they enter (for security reasons). This section uses the AdminInvalidateTokenFragment.

Corresponds to the layout *fragment\_admin*.

### Area Fragments

* **UserInfoFragment**, a subclass of Fragment – a logical area fragment where the user information can be inputted and validated. It is used in RegisterFragment, AdminCreateUserFragment, ProfileFragment, and CheckoutPopup. It has the option of showing or hiding the passwords section and allowing the form to be editable or not.

Corresponds to the layout *layout\_user\_details*.

* **AdminCreateUserFragment**, a subclass of Fragment – a logical area fragment used in the AdminFragment class where the user can register another user. It is similar to the RegisterFragment but adds the option for the created user to be an admin. This fragment uses the UserInfoFragment class for the general user info form.

Corresponds to the layout *layout\_admin\_create\_user*.

* **AdminInvalidateTokenFragment**, a subclass of Fragment – a logical area fragment used in the AdminFragment class where the user can enter an id of any user and invalidate the token of that user.

Corresponds to the layout *layout\_admin\_invalidate\_token*.

### Popup Classes (AlertDialogs)

* **Loader**, a subclass of AlertDialog – a popup with a loading circle, used to indicate loading. This class is instantiated in ParentActivity and used in some server requests.

Corresponds to the layout *popup\_loader*.

* **SimpleMessagePopup**, a subclass of AlertDialog – it shows a message, it is used in various classes to show messages to the user.
* **ErrorPopup**, a subclass of AlertDialog – it shows error messages, it is used in various classes to show error messages to the user. Used for errors from server responses or validation fails.
* **LoginPopup**, a subclass of AlertDialog – Displayed in the SplashFragment. The user and enter an email and password which are then sent to the user to authenticate the user and allow entrance to the app. If the user chooses they can click the register button which will shift the current fragment to the RegisterFragment.

Corresponds to the layout *popup\_login*.

* **FilterProductsPopup**, a subclass of AlertDialog – Displayed in the ProductsFragment and uses the ProductFilter model for storing data. It is used to filter the products (Product class) loaded and displayed in the ProductsFragment class. It allows filtering by text in the title of the product or by category of the product. The categories are displayed in a Spinner using the list adapter class CategoriesFilterAdapter (using a list of the ProductCategory class). Corresponds to the layout *popup\_filter\_products*.
* **ProductDetailsPopup**, a subclass of AlertDialog – Displayed in the ProductsFragment and CartFragment to display data on a given product. Specifically, it is triggered from the list adapters classes CatalogProductsAdapter and CartProductsAdapter respectively (each used in the corresponding fragment).

Corresponds to the layout *popup\_product\_details*.

* **ProductDetailsAdminPopup**, a subclass of ProductDetailsPopup – Displayed in the ProductsFragment to display data on a given product and allow editing or disabling of that product or creating a new product (and can make the corresponding server requests). This popup is only available to admins. Specifically, it is triggered from the list adapters classes CatalogProductsAdapter when editing/disabling or the fragment itself when creating a new product.

Corresponds to the layout *popup\_product\_details*.

* **GetImagePopup**, a subclass of AlertDialog – Displayed in the ProductDetailsAdminPopup and used to allow the user to choose an image for the product they are creating/editing. This makes permission checks/requests and when given permission it can open the device’s camera to take a picture or open the device’s storage system to select an existing picture. If a picture is selected, it’s data is converted to base64 encoding and is stored in the Product model as such (and uploaded to the server in that format).

Corresponds to the layout *popup\_get\_image*.

* **CategoriesPicker**, a subclass of AlertDialog – Displayed in the ProductDetailsAdminPopup and used to allow the user to choose a selection of categories. It displays the categories using a RecyclerView and the list adapter class CategoriesPickerAdapter which uses the model class SelectableCategory (instances of this model are created using instances of the ProductCategory model and it adds a boolean for if it is selected or not). In addition, it allows for the addition of a new category (a title is provided and the relevant server request is made).

Corresponds to the layout *popup\_categories\_picker*.

* **OrderDetailsPopup**, a subclass of AlertDialog – Displayed in the OrdersFragment. Specifically, it is shown when clicking the details button of an order summary element displayed using the list adapter class OrderSummariesAdapter (using the ExpandableOrder model class as explained above). It displays details about the order corresponding to the OrderDetails model class. Most importantly the products purchased in that order. This is done using a RecyclerView along with the list adapter class OrderDetailsProductsAdapter which uses the OrderDetailsProduct model class as items.

Corresponds to the layout *popup\_order\_details*.

* **CheckoutPopup**, a subclass of AlertDialog – Displayed in the CartFragment when the checkout button is clicked. It is used for the user to fill out the checkout form. This included the user information for the order (which does not have to be the same as the user’s account information – though that is automatically filled in for ease of use) and credit card information. This popup uses the area fragment UserInfoFragment for the user information. Once validated a server request is made to create a new order. Once successful the cart is emptied and in the orders tab, the user will see the new order.

Corresponds to the layout *popup\_checkout*.

* **AdjustableSpinnerDatePickerDialog**, a subclass of DatePickerDialog. This is functionally the same as DatePickerDialog except that different sections of the date (days, months, or years) may be hidden. This popup is used in the CheckoutPopup class to allows the user to choose the expiration date of the credit card. In this use, the days section of the date is hidden.

### List Adapter Classes

Below are the classes used to present lists of model classes in the UI. In the case of this app, they are all used for RecyclerViews aside from one used for a Spinner.

Note that I use the term “list adapter class” for these but the Android SDK does provide a class called ListAdapter. That is not what I’m referring to, I am referring to the concept of adapting a list of data to a visible list on the UI. Though ArrayAdapter used for the Spinner’s adapter does inherit from ListAdapter, RecyclerView.Adapter which is used for the RecyclerView’s adapters does not.

* **BaseProductsAdapter**, a subclass of RecyclerView.Adapter. It is an abstract superclass for adapting lists of the Product model into UI lists. It contains functionality for displaying a single product, loading the quantity of it in the cart (using the CartStore), in the case of non-admin mode, displaying and adjusting that quantity using the custom view class, Stepper, and displaying a popup containing for information (providing the overridable default of ProductDetailsPopup as the popup containing that information).
* **CartProductsAdapter**, a subclass of BaseProductsAdapter. This class is used for adapting a list of the Product model into the corresponding UI of the used in the CartFragment.

Corresponds to the layout *cell\_product\_cart*.

* **CatalogProductsAdapter**, a subclass of BaseProductsAdapter. This class is used for adapting a list of the Product model class into the corresponding UI for the ProductsFragment. It also overrides the additional information popup to either ProductDetailsPopup or ProductDetailsAdminPopup depending on if the user is in admin mode or not.

Corresponds to the layout *cell\_product\_catalog*.

* **CategoriesPickerAdapter**, a subclass of RecyclerView.Adapter. This class is used for adapting a list of the SelectableCategory UI model (each created from an instance of the ProductCategory model as described above) into the corresponding UI of CategoriesPicker popup. Aside from displaying information, it allows the user to select any amount of the items using checkboxes.
* **OrderSummariesAdapter**, a subclass of RecyclerView.Adapter. This class is used for adapting a list of the ExpandableOrder UI model (each created from an instance of the OrderSummary model as described above) into the corresponding UI of the OrdersFragment. Aside from displaying information about the OrderSummary, it allows the user to click on each item which will then expand that item showing more information and a details button which will open the OrderDetailsPopup for that order when clicked.

Corresponds to the layout *cell\_order\_summary*.

* **OrderDetailsProductsAdapter**, a subclass of RecyclerView.Adapter. This class is used for adapting a list of the OrderDetailsProduct model into the corresponding UI of the OrderDetailsPopup, displaying the products purchased for the order in question.

Corresponds to the layout *cell\_order\_details\_product*.

* **CategoriesFilterAdapter**, a subclass of ArrayAdapter. This class adapts a list of the ProductCategory model class into the corresponding UI for the FilterProductsPopup.

Corresponds to the Android provided layout *simple\_spinner\_item* (chosen item) and *simple\_spinner\_dropdown\_item* (dropdown item).

### Custom View Classes

One custom view was developed for this app. A Stepper view, as it appeared in a few places.

* **Stepper**, a subclass of FrameLayout – a View class which has two buttons for increasing and decreasing an amount, a textfield for input an amount manually and text for a title that what the amount represents. This component supports setting an initial value, minimum and maximum values, a step value (that is, how much is added/subtracted when the buttons are clicked), precision of decimal places, and title for the stepper and the value itself (along with various UI settings such as coloring and sizing)

In addition, it provides an event that fires when the value changes. A listener for list even is an instance of Stepper.OnValueChangedListener, an inner static class of Stepper.

This class is used in the ProductDetailsAdminPopup class and BaseProductsAdapter class (separately in both layouts that are loaded by it, *cell\_product\_catalog* and *cell\_product\_cart*).

Corresponds to the layout *layout\_stepper*.

## Models

Below are the data models used in the app. These are divided into different categories of models.

### Entity Models

These models each represent a single logical data group that are persisted to local or remote storage.

* **User** – a class representing a user of the app. It roughly corresponds to the schema of the users table in the remote MySQL database.
* **Product** – a class representing a product. It roughly corresponds to a join of the schemas of the products table, the stock table, the products\_categories table, and the categories table in the remote MySQL database. It contains categories as a list of ProductCategory model class.
* **ProductCategory** – a class representing a category a product can belong to. It roughly corresponds to the schema of the categories table in the remote MySQL database.
* **OrderSummary** – a class representing some of the data of an order, namely the information aside from the products that were purchased in the order. It roughly corresponds to a join of the schemas of the orders table and the orders\_products in the remote MySQL database.
* **OrderDetails** – a class representing some of the data of an order, namely the products that were purchased in the order. It roughly corresponds to a join of the schemas of the orders table, the orders\_products, and the products table in the remote MySQL database.
* **OrderDetailsProduct** – a class representing some of the data of a product, that is data shown for products that were purchased in an order. It roughly corresponds to a join of the schemas of the orders\_products and the products table in the remote MySQL database.
* **CartProduct** – a class representing a products added to the user’s cart. It corresponds to the schema of the cart table in the local SQLite database.

### Pure Response Models

The single model in this category represents a model that exists only to match a response from the server and has not use as a unit otherwise.

* **LoginResponse** – a class representing the response data received from the server for a login/register request. It contains an instance of the User model class representing the user that has just logged in and a list of instances of the ProductCategory model class representing all the categories products may belong to in the app.

### UI Models

These models exist as models that are needed for UI purposes. In this project, they expand on existing models to adapt them to UI needs.

* **ExpandableOrder** – a class adapting the OrderSummary model class, used in displaying a list of orders in the OrdersFragment using the OrderSummariesAdapter list adapter class. The UI allows for the items to be clicked and expanded, showing more details. This class contains an instance of the OrderSummary class and adds the data of whether the item is expanded or not.
* **SelectableCategory** – a class adapting the ProductCategory model class, used in displaying a list of categories in the CategoriesPicker popup using the CategoriesPickerAdapter list adapter class. The UI allows for the items to be selected with a checkbox, allowing several categories to be selected. This class contains an instance of the ProductCategory class and adds the data of whether the item is selected or not.

### Misc. Models

The single model in this category didn’t fit into any of the above categories and is somewhat a hybrid of a UI model and data model.

* **ProductFilter** – a class representing a filter for the products. This class is used in the ProductsFragment as the current active filter and in the FilterProductsPopup where a new filter is created to be used in the fragment.

## Networking

These classes deal with the the clients connection to the remote server. This project uses the Retrofit library for networking, as such many of the classes deal with Retrofit related topics.

### Infrastructure

* **RetrofitClient** – a class with the sole purpose of retrieving a sole instance of the Retrofit class (and instantiating if it hasn’t been). The Retrofit class is used for creating remote services. As such, it is used in the RemoteServiceManager class when instantiating each of the three services this app uses. This class is similar to a singleton except that the instance it provides is of a different class (the Retrofit class as opposed to the RetrofitClient class)
* **RemoteServiceManager** – a singleton class that instantiates and provides sole instances of each of the service classes (actually the façade of each service, that is UsersServiceFacade, ProductsServiceFacade, and OrdersServiceFacade) to be accessed throughout the app. In addition, it provides a sole instance of the Gson class that will be used for manual JSON parsing in the ServerCallback class.
* **ServerResponse**, a class representing the standard format that all server responses will fit (whether they were successful or not). This contains four fields, a String token, and error of class ServerResponseError, result of some generic class (whatever type we are expecting from the request), and a boolean hasError. Aside from hasError, any field can be null (or missing from the response and so will be received as null). If the server returned an error, the hasError flag will be true, error will be non-null, and result will be null. If the server returned a success, the hasError flag will be false, the result will be non-null (except in the case were the generic type of result is Void), and error will be null. In both cases, if there is a token to be returned it will be (this may not be the case for instance in a response for the register or authenticated login endpoints as the user is unknown and so there is no token to return).
* **ServerResponseError** – a class representing the standard error format this project uses. It is comprised of an error code and error message (note that the code is not the HTTP status code, but a code returned by the sever indicating project-specific reason for failure).

This class is a static inner class of ServerResponse.

* **ResponseSuccessCallback** – an interface with with a method to be called when a server request of type ServerResponse is successful.
* **ResponseErrorCallback**– an interface with with a method to be called when a server request of type ServerResponse is unsuccessful.
* **StandardResponseErrorCallback**, a class implementing the ResponseErrorCallback interface. It is used for standard error handling for server errors. This includes allowing a retry for unknown errors, generic error message for unhandled error, standard error message for access denied responses and logging out the user if the sent token was found invalid.

This class is used in all server request throughout the project.

* **ServerCallback**, a subclass of Callback (of the Reftrofit library) and implements the ResponseSuccessCallback and ResponseErrorCallback interfaces – an abstract class representing a server callback, that is a response of a server request. It expands base Callback class, fitting it to the project. The base Callback class has two options, Response and Failure. These options do not fit this project well because Response simply means that a response was returned from the server even if that response was indeed a failure. Failure generally refers to a request that didn’t reach the server at all for various reasons.

To adapt the Callback class to fit the needs of the project, the ServerCallback class reclassifies server responses into two new options described in the interfaces it implementes. That is a response where the the server returned and error or another failure occurred, or if the server returned success. This is done by examining the response and when needed, manually parsing, into an instance ServerResponse class (on success this is done automatically as it is the base type of all server calls), which will create the correct instance of ServerResponseError. In the case of Failure, an instance of ServerResponseError is also created.

In addition, it stores the token returned as part of the ServerResponse that all server requests that reach the server respond with.

* **ErrorCodes** – a utility class for networking, it provides the various error codes, saved as constants, that the app reacts to. These are a subset of the error codes the server uses.

### Services

This section deals with the three services the server provides the client. That is, the User, Products, and Orders services. The client side implementation of each service is divide into two. The first is the service interface, enumerating all the enpoints along with their HTTP methods, header, parameters and expected response type, for which the Retrofit framework generates a concrete class that can be accessed. The second is a façade class which wraps it respective service, allowing greater ease of use and collects common code.

Each façade class has two overloaded versions of every method, one with a Loader instance as a parameter and one without. If a Loader instance is passed, then the server call will show the app loader while waiting for a response.

* **BaseRemoveService** – an interface which all service interfaces extend. It does not add any methods, it is used only for logically grouping the various services together.
* **BaseRemoteServiceFacade** – an abstract class serving as the superclass for all the service façade classes. It provides basic common functionality like queuing a given server call.
* **UsersService** – a service interface providing all the user endpoint: login (with token), authenticated login (with email and password), register, remote register, update information, invalidate token.
* **UsersServiceFacade** – a service facade class wrapping the server calls in the UserService interface.
* **ProductsService** – a service interface providing all the products endpoint: getting paged (and filtered) products, creating a product, update a product, set product enabled (only used to disable a product), get products by given ids, and create a product category.
* **ProductsServiceFacade** – a service facade class wrapping the server calls in the ProductsService interface.
* **OrdersService** – a service interface providing all the user endpoint: get paged order summaries by user id, get order details by id, create order.
* **OrdersServiceFacade** – a service facade class wrapping the server calls in the OrdersService interface.

# Server

## Utility classes

Most of these classes are a collection of static functions used throughout that app.

* **BcryptUtils** – contains function(s) dealing with the BCrypt library for securing passwords.
* **JwtUtils** – contains function(s) dealing with the JsonWebToken library for creating a working with json web tokens (JWT tokens). These are widely used throughout the server and client as the main form of authentication. Tokens for a user in the app are created using a secret key that that user has stored in the database. This is so that it can be refreshed/invalidated. If the secret key is refreshed (changed) then the existing token is no longer valid. When a token expires, during the authentication process, if the existing token was valid, the secret key is refreshed and a new token is returned to the client to replace the expired one. If an invalid token is sent by the client, access is denied.
* **ImageConversionUtils** – contains function(s) dealing with the conversion of a String to a byte array and vice versa. This is used for the base64 encoded images that are received from the client but stored in the MySQL database as a byte array in the form of a MEDIUMBLOB.
* **JsonUtils** – contains function(s) dealing with the JSONs. It provides methods for creating json elements from objects, creating the standard response json wrapper that the client expects and other json related utilities.
* **CollectionUtils**– contains function(s) dealing with collections.
* **ResponseUtils** – contains function(s) creating Response instances for various basic situations like a simple error response or encapsulated try and catch blocks.
* **UncaughtExceptionMapper**, a class that implements the ExceptionMapper – this class deals with responding to all uncaught throwables that the server encounters. There are to main situations where this will be used. If an attempt to access a non-existent URI was made (within the host), and if an uncaught error was thrown while authenticating the user token.
* **ErrorCodes** – contains a list of constants representing all the possible error codes that the app may use (and one generic error message).
* **Result** – a class for representing the return value from a method which may or may not have succeeded. It has a different generic type for the value type (if the function succeeded) and the error type (if the function failed). This class is used throughout the app anywhere a result may more than a single option, or a single option or null, or a boolean type. For instance when the return type is a model type but the retrieving or creating of that model may have failed for various reasons. In the project the error type is always Integer.
* **Logger** – a class used for logging to the server console. It provides various options of logging. It is used throughout the app. It allows for logging to be turned off and on app-wise.
* **ResponseLogger** – a class that used the Logger to log a given Response instance.

## Model classes

Classes that represent the models matching either the schemas of the MySQL table, or the response format that will be returned to the client.

* **User** – an entity class corresponding to the schema of the users MySQL table and contains named queries for conditional selecting and a named stored procedure for refreshing a user’s secret key (used in tokens). It contains information on users of the app.
* **Stock** – an entity class corresponding to the schema of the stock MySQL table and contains a named query for conditional selecting. It contains information on stock of product, as such it contains a reference to the Product object whose stock it is defining, retrieved using JPA (making the class, more accurately, represent a join of multiple tables).
* **Product** – an entity class corresponding to the schema of the products MySQL table. It contains information about a product. This information includes its categories, as such it contains a reference to a collection of Category object that this product belongs to retrieved using JPA (making the class, more accurately, represent a join of multiple tables).
* **Category** – an entity class corresponding to the schema of the categories MySQL table and contains named queries for conditional selecting. It mainly contains the id and title of the categories that products can belong to.
* **Order** – an entity class corresponding to the schema of the orders MySQL table. It contains information about orders made by the users. This information includes the user who make the order and the products purchased in the order, as such it contains a references to a User instance and a collection of OrderProduct instances retrieved using JPA (making the class, more accurately, represent a join of multiple tables).
* **OrderProduct** - an entity class corresponding to the schema of the order\_products MySQL table. It contains information about products in an order at the time of the order. This information includes a reference to the Product this OrderProduct is expanding upon, retrieved using JPA (making the class, more accurately, represent a join of multiple tables).
* **CatalogProduct** – a model representing an alteration of the Stock class that matches the format sent to the client.
* **OrderSummary**– a model representing an alteration of the Order class that matches the format sent to the client when requesting order summaries. Namely, it doesn’t include the products of the order.
* **OrderDetails**– a model representing an alteration of the Order class that matches the format sent to the client when requesting order details. Namely, mostly includes the products of the order.
* **OrderDetailsProduct** - a model representing an alteration of the OrderProduct class that matches the format sent to the client within the OrderDetails class.

## Service classes

These classes are Singletons and LocalBean classes, as such they are encapsulated and accessible to other classes using the EJB (Enterprise Java Bean) annotation. They mainly contain methods accessing the database (via access to EntityManagerSingleton also using the EJB annotation itself) and service the resource classes.

* **BaseService** – an abstract superclass for the non-dummy service classes (see TransactionService below for more information about the “dummy service”). It contains various methods useful to the concrete service classes such a basis for all criteria queries, running an action within an entity transaction and so on.
* **UsersService** – a subclass of BaseService, is contains methods needed for the various endpoints (mainly of the UsersResource) such as checking user authentication, creating tokens, refreshing secret keys (used in tokens and as such invalidating them), finding users, and creating and updating users.
* **ProductsService** – a subclass of BaseService, is contains methods needed for the various endpoints (mainly of the ProductsResource) such as getting paged (and filtered) products, getting products but given ids, getting all categories, creating a category, getting categories by ids, and creating/updating/disabling a stocked product.
* **OrdersService** – a subclass of BaseService, is contains methods needed for the various endpoints (mainly of the OrdersResource) such as getting paged orders by user id, getting order details by id and creating a new order.
* **TransactionService** – this class is a dummy service, that is, it doesn’t actually do anything. It has methods that check nothing and always success. Its purpose is to simulate the verification process of a credit card and charging that credit card. As this app doesn’t actually charge nor provide real product, there is no real credit checking/charging service in use. This class takes that place in the process of checking out and creating an order.

## Resource classes

These classes are a collection of endpoints for each of the three services this app provides (Users, Product, and Orders). Each of the endpoints returns a Response instance which conforms to the stands response format (see the class ServerResponse near the end of the Client section). These classes contain references to the relevant service classes by using the EJB annotation as explained above.

* **BaseAuthenticatedResource** – an abstract superclass for the resource classes. It provides authentication methods and almost all endpoints must use before preforming any further action.
* **UsersResource** – a subclass of BaseAuthenticatedResource. It endpoint methods for the users services (see the UsersService class in the client section above for the different endpoint actions)
* **ProductsResource** – a subclass of BaseAuthenticatedResource. It endpoint methods for the products services (see the ProductsService class in the client section above for the different endpoint actions)
* **OrdersResource** – a subclass of BaseAuthenticatedResource. It endpoint methods for the orders services (see the OrdersService class in the client section above for the different endpoint action)

## Infrastructure classes

* **EntityManagerSingleton** – a class marked as with the Singleton, LocalBean and Startup annotations. The Startup annotation makes this class get instantiated when the server started and has life cycle annotations for methods, such as the PreDestroy annotation we use in this class to close the EntityManagerFactory opened in the constructor of this class. In that constructor a connection to the MySQL database is established, and an EntityManager instance is created using the apps persistent unit and stored in a field accessible through a public getter and accessed by BaseService class and used in the service classes in general.
* **JAXRSConfiguration**, a subclass of the Application class – this class serves as the web services configuration class, enabling the use of web services on the server.