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.
* // todo – perhaps make the two above package access
* **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.

* **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. 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. 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*.