It-integration OrderApp
2018-01-12
Written by: Jim Andersson | andersson.jim.sweden@gmail.com

This document intends to describe the current state of the OrderApp and the packages and techniques that are used.

The OrderApp is a simple app intended to let a user place orders on articles from a company that the user is associated with. The app is intended to extend the FlerlagerPlus system in an app format. The app shall (in the future) make use of a REST service to receive and pass data that will roughly do the following below.

Current implementation of the OrderApp uses a local db called "FlerlagerPluis" (the "i" is not a typo) to receive dummy data to work with.

Requirements:
(may be changed)

1 - Verify a user on login. **(NOT implemented)**
2 - Receive a unique user token. **(NOT implemented)**
3 - Token is used to fetch relevant dataset of the company that the user is associated with. **(NOT implemented)**
4 - The dataset is used to create a list of "article" objects which can be browsed and searched for, and also altered before being added to an order. **(implemented with local DB)**
5 - A User shall then be able to make multiple searches for articles, be able to edit each one, and add/remove them from the order-list. **(implemented)**
6 - Once in the order-list, User shall be able to review each added article, make additional changes, delete articles if needed and finally confirm the order. **(implemented)**
7 - User confirmation of orders should involve a REST service call to place the order to FlerlagerPlus(System). As well as email confirmation using User email that should be retrieved at login. **(NOT implemented)**

Additional requirements will involve the creation of saving order-history and retrieve historic orders that can be placed again to FlerlagerPlus, with changes if need be. User shall also be able to access a settings menu from the login screen in order to setup network access, or make a registration request.

Additional requirements should be confirmed with Lars-Göran "LG" Jönsson.

IDE:
Android Studio latest version. Run all updates before importing the project from git.

To import project, do: *File -> New -> Project from version control -> Git*
Now add **https://github.com/It-Integration/FlerlagerWebOrderApp** in the Git Repository URL:
Parent directory can be chosen at your discretion, I personally opt for something like **C:/dev/workspace**.
Directory name: **OrderApp**.

Ask for help if the Github repository is set to private making you unable to obtain the repository URL, as of now the repo is public.

Local DB setup:
Download Microsoft SQL Server Management Studio 17 or later and connect to Flerlager's testserver called "Flerlagerpluis" ("i" is not a typo). See Anders Pehrson or Roger Fisk for additional information on how to setup the local database. In order to work with the app you must be connected to the local db through the SQL management studio desktop app.

App local database connection:
Database connection details in the app (can be found hardcoded in *com.itintegration.orderapp/data/dbconnection/MsSQLConnection*) are the following:

```
private String IP = "172.18.40.196:50085/SQLEXPRESS";
private String CLASS = "net.sourceforge.jtds.jdbc.Driver";
private String DB = "Flerlagerpluis";
private String UN = "sa";
private String PASSWORD = "Need4speed";
```

To the dev:
It is assumed that you know the basics of the android framework and the **app lifecycle(activity and fragments)**, I will list some topics which are beneficial to understanding the overall relationships and structure of the app.

Dependecy injection: I user **dagger 2** for this project, and the example project I used to help implement it in my project for an appwide reference of the DataManager and it can be found here. I use it mainly to get access to the DataManager class which will connect the UI and Database/REST calls. It also holds session-scoped "provider" classes that populate list-Adapters with data. To make obtained data persistent when the app is idle for a long time and the memory is freed, we need to create a local SQLite db and save the info there. See DbHelper.java and DataManager.java for an example.

AsyncTask: How to use asynchronous tasks to process future REST calls while displaying a UI progress bar. This is not implemented as of now as a local DB can make do without it but it will be crucial to implement it later.

AdvancedRecyclerView: This is the library used to create the UI elements that are hosted in the RecyclerView's of AssortmentActivity/OrderActivity. It may be the source of some unintended bugs, and if it's too hard manage I would suggest using a different Adapter setup to implement the list UI. Frequent issues are discussed here: https://github.com/h6ah4i/android-advancedrecyclerview/issues. There are example guides to follow and I would suggest to look at some of them or download and run the sample project to get an idea of what it's intended to do.

Project structure:
/data
Contains relevant classes for database connections, POJOS (datamodels), provider classes to hold data structures for our lists. The eventual REST architecture should be placed in a new **/service/** folder and communicate through DataManager. DataManager is the class to process any and all petitions for data to and from the classes located in the **/ui** folder. Here all the POST and GET methods be implemented from REST service classes.

/di
Holds all relevant classes and declarations needed to implement dependency injection plus the OrderApp.java class. See "Dependency Injection" up top to learn more about it.

/tasks
Folder to hold AsyncTasks. May be easier to implement as a local implementation but it can create lots of boilerplate code. Better to place them separately. There are additional notes on AsyncTask implementation in AssortmentActivity.java

/ui
Holds all relevant classes for UI. Activity acts as container for the Fragment, and the Fragment are a container for the RecyclerView declared inside Fragment. We attach an Adapter to our RecyclerView to create and manage the UI elements that populate each list entry of the recyclerview. Each entry of the RecyclerView is hosted by a ViewHolder. ViewHolders are essential to understand and if you do not I would recommend you to study recyclerview examples online to understand how they work as they are prone to many different bugs otherwise.To communicate between these classes we implement *Interfaces* to do so. Be aware that certain actions such as notifyDataSetChanged() on the

adapter must be called from the fragment class otherwise such a call might interfere with animations and cause an exception.

We use *RecyclerView* primarily because they make effective use of the low memory capacity of mobile applications. Instead of creating and initializing 500 UI holders for each item in a list of 500 objects we may instead create 10 holders for 500 objects, and swap the data in and out of each item as we scroll up and down the list, greatly increasing performance. If you want I would recommend to read up on ways to improve app performance in android apps.

If there's anything technical that's still very unclear about the project, don't hesitate to send an email.