# System design document for Watchr

**Authors:**
Alexander Lysholm
Johan Lindkvist
David Olsson

**Date:**
25/10/19

**Version:**
2.0

## 1. Introduction

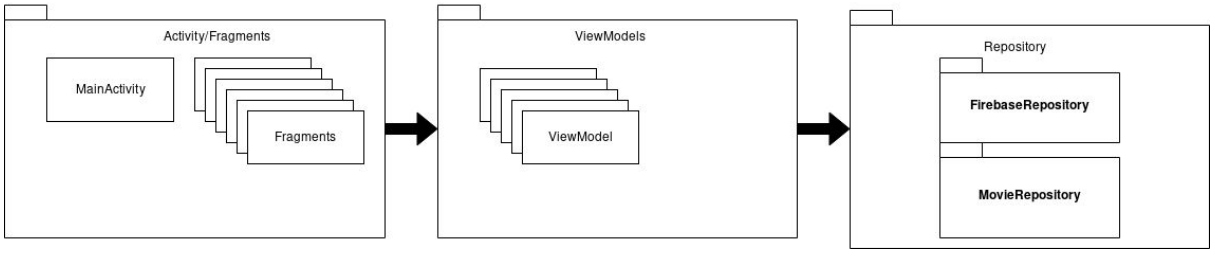
This is the system design document (SDD) for the app Watchr describing the architecture and design used. Watchr is an app to help users find movies to watch by letting the user add movies to different lists, searching and filtering through them. Also an algorithm is used to recommend movies to the user.

### 1.1. Definitions, acronyms, and abbreviations

MVVM - Model, View, ViewModel
SDD - System Design Document
RAD - Requirements and Analysis Document
UML - Universal Markup Language
GUI - Graphical User Interface
UI - User Interface
DAO - Database Access Object

## 2. System architecture

Our application follows the MVVM architecture recommended for android development. It has three main components, the view/UI-controller/(Activity/fragments), the viewmodels, and the repository.



## 2.1. Activities/fragments

The activity/fragment/UI-controller is a thin dumb layer that has the sole responsibility of passing on user input to the viewmodel and displaying whatever data the viewmodel has. This is done with observable data types called LiveData. When a UI-controller is created it will attach a listener to the LiveData in the viewmodel, mirroring and acting on what it returns and displaying it on the screen. When a UI controller receives user input it will pass it to the viewmodel which will in turn process it and respond by updating its livedata which propagates back to the viewmodel.

## 2.2. The ViewModel(s)

The ViewModel classes are a layer between the repository and the activity/fragment/UI-controller which handle presenter logic. They receive and process UI actions and then update the LiveData objects it holds accordingly, which propagates back to the view. It can also indirectly change livedata it holds by calling on the repository as a result of UI actions. The ViewModel holds both its own livedata which represents different states etc, but it can also pass on LiveData from the repository and manipulate that livedata by calling on the repository which then propagates back the results through LiveData same as with the activity/viewmodel relationship.

The superclass (ViewModel) links the viewmodels to the activity that created them. Even if the activity goes through reconfiguration and is destroyed the viewmodel attached to it will survive as long as the activity is in the stack, storing the state. When the activity is recreated it can sync with the viewmodel. This design allows us to save on resources and computing speed, making the application faster and less resource heavy.

This design allows separation of the UI logic from the presenter logic. Following the separation of concern. Moving the responsibility of fetching and managing/processing/storing data away from the activity/fragment/UI-controller  making them thin and easier to manage with the android life cycles.

## 2.3. The repositories

The repository is a module responsible for storing and fetching data from an API. In our case it will respond to requests with LiveData which in turn represents the most up to date data the repository has access to. When a resource in the repository is updated all LiveData objects that represents this resource in any shape or form are updated as well which propagates the change throughout the system, this reduces the risk of memory leaks and sync errors.

When identical resources are requested only one LiveData object is created, and when the data it represents is updated that same LiveData object triggers triggers its observers. Any action sent to the repository propagates back to the viewmodel/activity through this process. This syncs a certain resource with anyone wanting to use it over the entire application. Calls on the repository that change any date propagates back through this method.

### 2.3.1 FirebaseRepository

Firebase is api and localstorage combined into one, it handles user authentication/data and is the root source of many LiveData objects. All get methods on this repository returns livedata which is synced to the persistent local firebase data corresponding to the requested resource.The persistent storage is in turn in synced to the data on the firebase servers. Since changes on livedata propagates all the way back to the activity/fragment/UI-controller, in essence it syncs the servers to the views.

### 2.3.1 MovieRepository

This repository is a singleton that stores downloaded data, saving bandwidth. When data is requested, the MovieRepository will return an observable data type (Livedata) which is synced to local storage (Room). The repository will then either find fresh data already present in localstorage or it will fetch it from the API's and put it there. When localstorage is updated or changes corresponding LiveData is updated which propagates the new dataset throughout the entire system.

# 3. System design

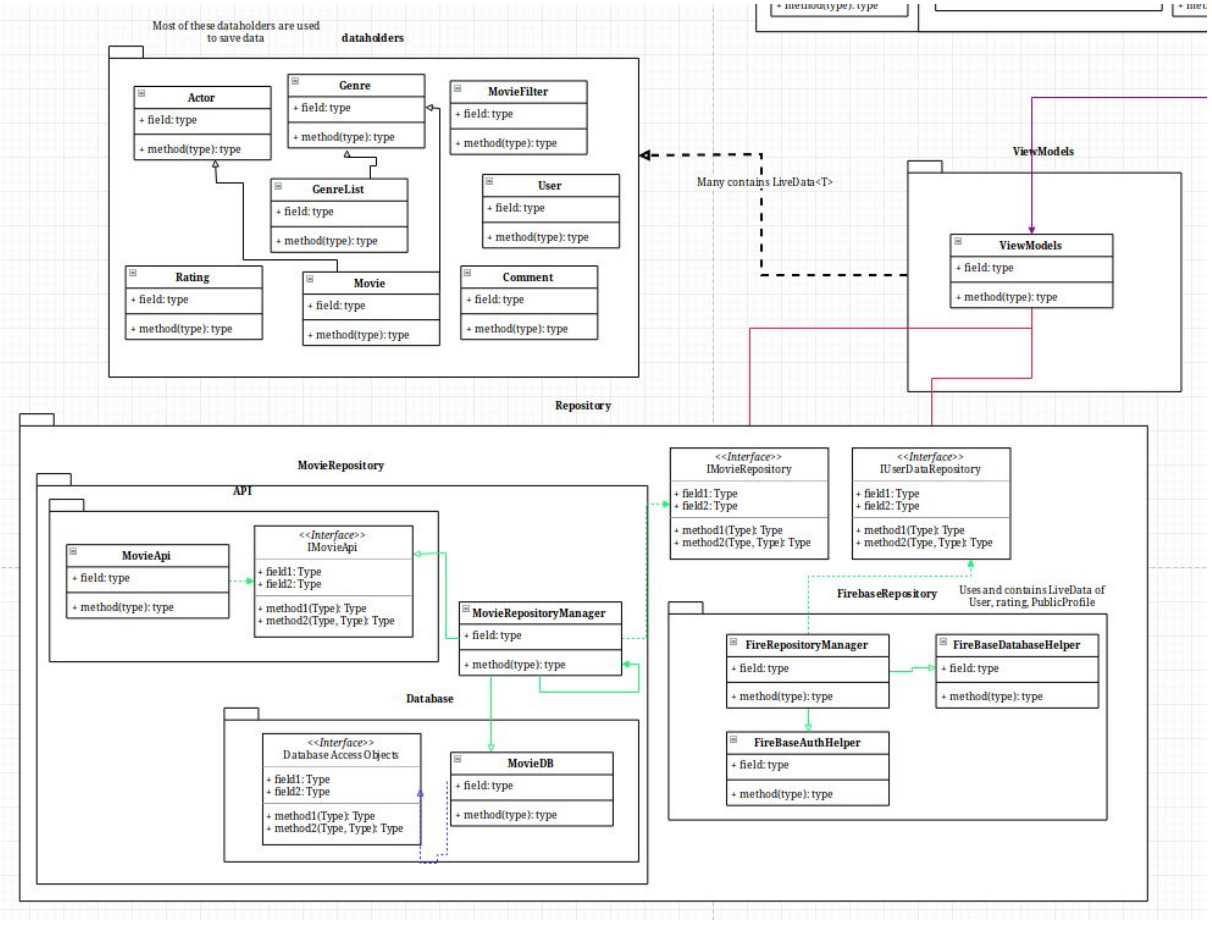



*Figure 1. UML of our model (simplified)*

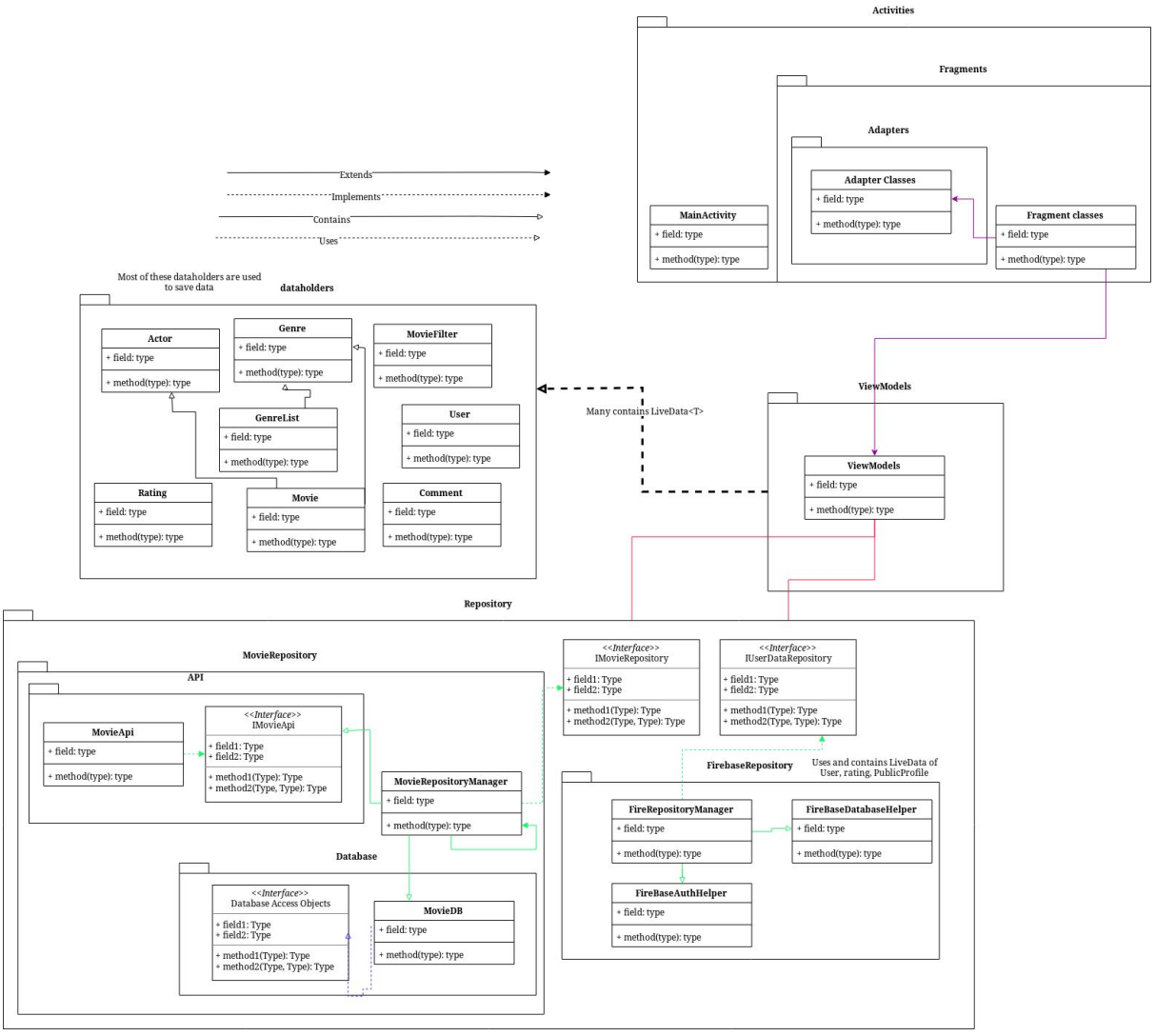



*Figure 2. UML of of the whole project (simplified)*

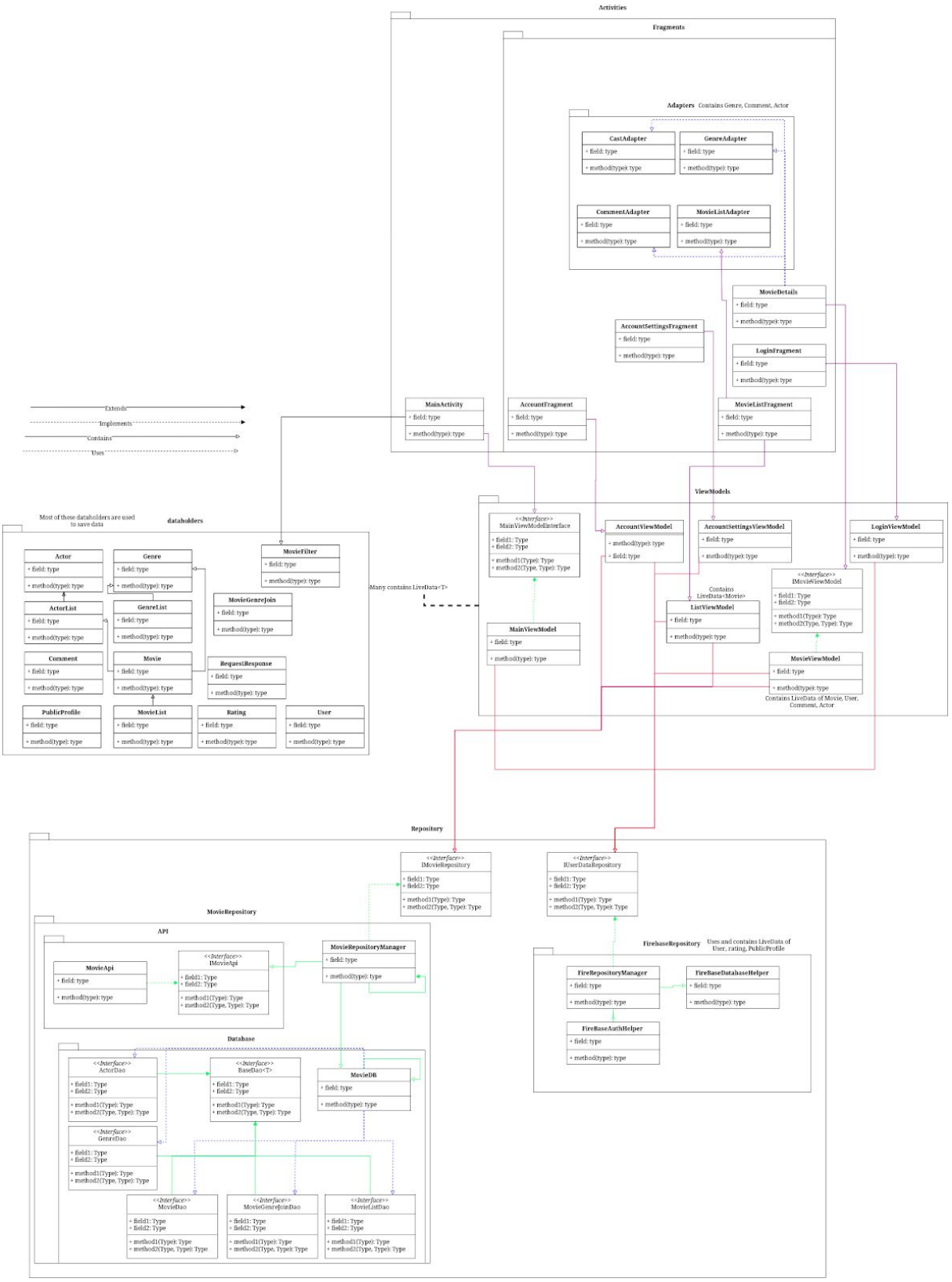



*Figure 3. UML of the whole project (more detailed)*

## 3.1. Explanation of the MVVM architecture

Our application has three main components, the view, the view model and the repository. The view is a thin UI-controller that mirrors LiveData from the view models. The view also passes on onClick events to the viewmodel which in turn updates Livedata objects accordingly, changes propagate to any fragment/activity listening to the Livedata object updating corresponding views.

The view models is a layer between the repository and the view that handles async data requests which comes in the form of requests such as login, users, movie info, list info, registration etc. These requests are triggered by either app activation or user events. The viewmodel also contains logic that manages UI events from the view.

The repository abstracts data retrieval and storage away from the view model. When a data request is made the repository returns a Livedata object. This LiveData object is updated and synced with the local database. When a data request is made the repository immediately returns a livedata object to the view model, if this data is not contained within the localstorage the repository downloads and puts that data in local storage. When localstorage is updated, LiveData objects linked to the updated data triggers their observers and the change propagates throughout the system.

## 3.2. MainActivity

MainActivity contains the navigation fragment which in turn holds all the fragments that make up the rest of our application UI. The MainActivity is for the most part just a container and a controller hosting both the user account navigation and the bottom navigation bar.

## 3.3. Fragments

Most of our application functionality is kept within easily switched fragments, this application architecture allows up to keep our app components independent from each other, simple and modular.

## 3.4. ViewModel

These classes are a layer between the repository and the view. They handle UI events and contain/update livedata which then through the observer pattern updates the view. The superclass of these models links them to the activity that created them. Even if the activity is destroyed and redrawn the viewmodel it initially created will remain the same until the activity is removed from the stack for good. This design allows us to save on resources and computing speed, making the application faster and less resource heavy. This allows separate the responsibility of fetching and managing data away from the views making them thin and easier to manage with the android system.

### 3.5. Repository

The repository is a singleton that stores downloaded data, saving bandwidth. When data is requested, the repository will return an observable data type (Livedata) which is synced to local storage(Room). The repository will then either find fresh data already present in localstorage or it will fetch it from the API's and put it there. When localstorage is updated or changes corresponding Livedata is updated which propagates the new dataset throughout the entire system.

### 3.6. Explanation of why we chose to use the MVVM architecture

The MVVM architecture is very modular in itself. The architecture is very modular in itself, we could easily swap out components such as viewmodels, or our models. However our model is "dependent" on some libraries we use which some may consider bad practice. But the dependencies aren't very "strong". The reason why we chose to do it like this is because it's how they use it in Androids developer guide to jetpack MVVM architecture, and everywhere else we've seen these modern libraries such as Room, Firebase, Gson been used. To try and totally get rid of these dependencies would be very hard and would go against the "Android way" of structuring your application. Not once have we seen people trying to abstract away dependencies such as Room, Gson or Firebase from their model.
On the other hand to try to build your application without these modern libraries would result in you having to do way more work.

When we say the model isn't very strongly dependent our libraries here's what we mean; for example all our data classes in the package dataholders contain very many annotations to either Gson or Room, but all you would have to do in order to get rid of these dependencies would be to simply remove the annotations and the dataHolders would work as normal.
On the other hand our MovieRepository and the stuff in Firebase packages is very dependent on Room respectively Firebase, but both of these can be easily swapped out. This is what is so great about the MVVM architecture, since all the Room code and Firebase code is pretty much hidden behind interfaces we could for example easily swap out MovieRepository for something else that doesn't use room, maybe even something that doesn't use any library and handle all the caching by itself without dependencies. All the 'major' classes are hidden behind interfaces and can be swapped out with something else really easily.

## 4. Persistent data management

Firebase is used for storing all things related to the user, such as user info but also the different lists related to the user. This includes watched, watch later and favorites. Firebase also includes a local cache for faster access. Along with the movie lists Firebase also stores the comments the user have made.

For caching movies from the API we are using the Room library. This library handles the storing of information inside a local SQLite database. All the SQL query operations are defined in Repository -> Database in all the DAO classes. The Room library verified these

queries on compile time and will notify you if there are any errors, for example trying to access something that doesn't exist or trying to compare a String with an int without the use of TypeConverters (Another Room feature).

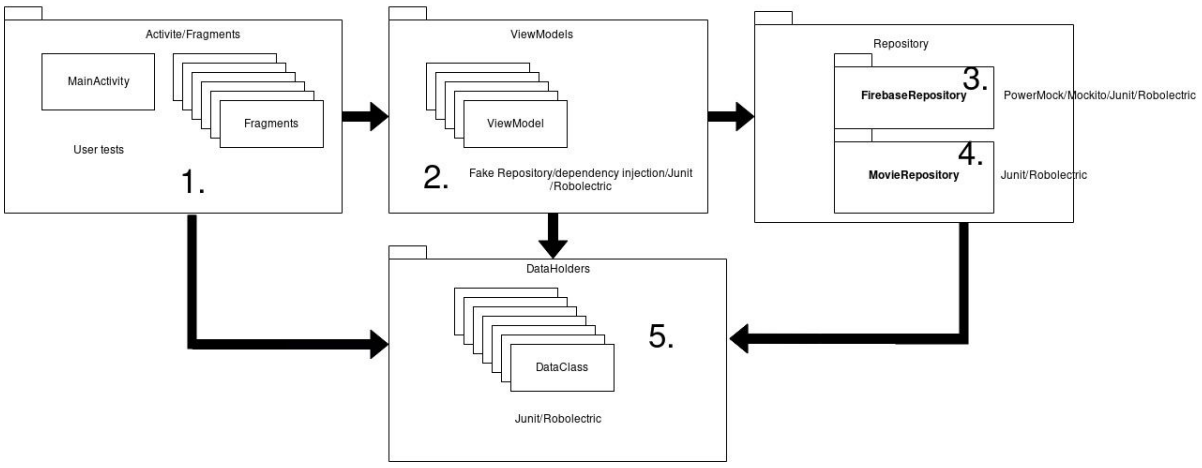
Every movie object is stored together with the date it was updated or inserted into the database. Since Room only can store primitive types such as int,string,boolean we use a TypeConverter to convert the Date object to a string before storing it in the DB and converting it back to a Date object when the movie object is fetched. When a movie is fetched from the DB it will check if the object is null (movie doesn't exist in the database), some movie information is missing (the movie was inserted through an operation on the API that didn't return all the necessary information) or the movie was inserted into the database over a day ago. If either of these are true it calls the API to get the necessary information in another thread and returns the LiveData object the Room database query gives us.

The reason for why we can call the API in another thread and just return the potentially null data object is that Room will update this LiveData object if something is inserted corresponding the that object into the database. Therefore even if the Livedata object is null, the call to the API will eventually insert that movie into the database and the LiveData object some viewmodel may hold will be updated.

Pictures are automatically cached and stored with the help of the Glide library. Glide helps us with downloading images from the web and automatically caches the images on the device so we don't have to deal with it; Images are thus also cached

# 5. Quality

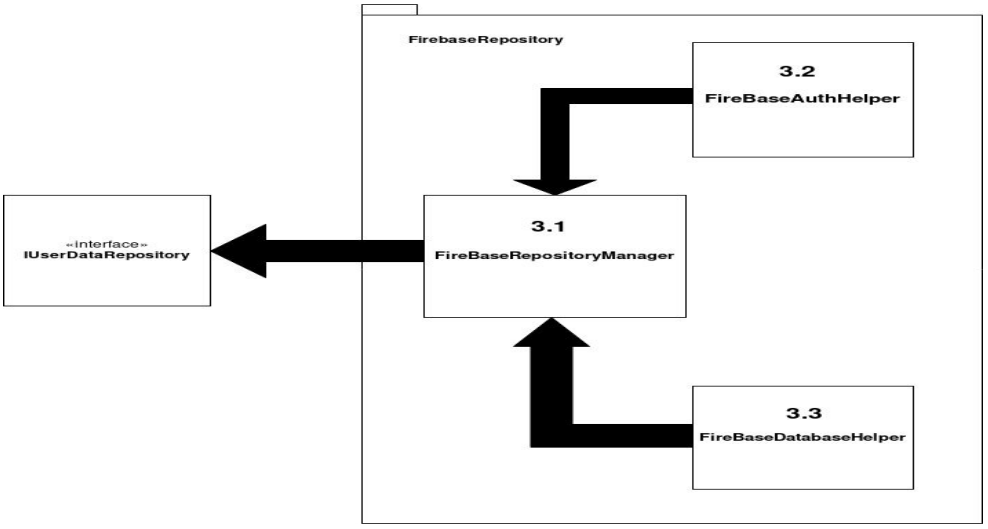
## 5.1. Testing




Since our application components function very differently and have very specific specialized functions we had to use a variety of methods/techniques to test each component. However our dependencies go one way which enables us to test most of our code with dependency injection. We can test our activities by injection of a fake ViewModel and we can then in turn

test our viewmodel by injecting a fake repository. We can test our repositories with injection but not completely since their bottom dependency is either a database library or an API.

The dependencies we do have to the dataholder package are very weak and are almost never a direct reference and almost always supplied through observing LiveData objects. This allows us to just inject a single fake class in almost all situations while maintaining high testability without having to resort to Mocking(Powermock, Mockito) however this does not apply to bottom classes that talk to the API.

1. Is user tested, which is not ideal and they could be tested by injecting a fake ViewModel if we abstracted our viewmodels behind an interface. However due to time limitations this was never an option, which is the main reason why the ViewModel package is not behind an interface. The effort spent putting them behind an interface would not pay off.
2. The ViewModels are tested by injecting a fake repository in combination with Junit and robolectric(Runs a fake android system) . Robolectric enabled us to run junit tests on things that would normally require either Mockito(Time consuming) or an emulator(Not tested in isolation).
3. 


   3.1 We tested the FireBaseRepository by injecting fake a FirebaseDatabaseHelper and a fake FireBaseAuthHelper while running Robolectric
   3.2 We tested the FirebaseAuthHelper by mocking the firebase library with powermock and mockito.
   3.3We tested the FirebaseDatabaseHelper by mocking the firebase library with powermock and mockito.

4. We tested the movie repository both directly and indirectly by testing the DAOs, one of the main things the movie repository does is inserting and getting stuff from the database so it's important that the DAOs are tested as well. The DAOs were tested through creating a new database and testing all the different operations the DAOs offer. The movie repository was then additionally tested to make sure that it's operations still work when it calls several different methods plus the DAO methods, and checked with expected output.

5. The DataHolders do not depend on anything so they could be tested without injection and plain junit, however sometimes small components depended on android libraries which forced us to use robolectric.

integration and the link can be found here:
https://travis-ci.org/ITJohan/tda367-objektorienterat-programmeringsprojekt

All the tests can be found in the “test” and “androidTest” folders. The “test” folder is for unit tests not requiring the user GUI and “androidTest” is tests that also uses the GUI.

## 5.2. Known issues

We did not abstract our data holders behind any interface, we considered doing this and hiding them behind a factory pattern however due to time limitations this was never done. However the dependency most classes have towards these classes are very weak already. And the payoff would be minimal. Same goes for the viewmodels

Some of our data holders classes that are passed around with livedata are not immutable, this creates a problem since the observable data can be modified where it should not be modified and that when it’s modified the LiveData observer is not triggered. All of us working on the project are aware of the fact that you should not modify data carried by LiveData so it has not posed a problem. However its a bad coding practice. We did not have time to fix this.

To reduce code duplication in the movie lists as they are pretty much the same, we tried to use a single XML, fragment and view model for them. This proved to be pretty difficult because of how the Navigation library works. The bottom navigation wants to create new a new fragment each time a list is revisited causing a new view model to be created. This destroys the purpose of using a view model as all data is lost. Also if you scroll a long way you will lose your position. A better way would be to extend the different list from a superclass to make the Navigation library happy.

There is also a lot of flickering when changing lists because of how the navigation in movie list is implemented. There is a lot of callbacks happening, for example first one needs to check if the user is logged in via a callback, then in that callback one needs to get the movie ids via a callback, and in that callback one needs to get the actual movies via a callback to finally choose what to display on the screen. This could maybe be done more cleanly.

Another problem is that movies doesn’t go away directly from the user lists when you remove them. We also didn’t have time to implement all of the functionality in the filter because of time constraints and under staffing.

There’s one problem with the database which occurs when the database is created. When the database is created it calls the API to get a list of all the genres, but sometimes it fails to do so (not often), so the database won’t contain the genres we need. Then when inserting movies and trying to associate a certain genreID with a genre that might not exist will cause

the app to crash. If you clear the apps database and open it, it might happen, but as I said it isn't common.

### 5.3. Access control and security

In order to use the full functionality of the application you need to have a verified user account that is signed into the application. The application handles/manages accounts on its own, these features include, registration, sign in ,reset password,  resend verification link, profile and account settings so. This removes the need for admin users, if someone that is not the owner of the user account needs to modify it, delete comment/rating/account that can be done from firebase. To manage/store/authenticate user account we use firebase which with the firebase android library allows us to authenticate/update users.

We have a login system in place where you can register a user profile. A user profile is required to be able to comment on movies and save movies to the different lists.
Only one kind of user is currently available as all the administrative tasks is being handled through the Firebase interface.

## 6. References


- Android Jetpack: https://developer.android.com/jetpack
- CircleImageView: https://github.com/hdodenhof/CircleImageView
- Navigation: https://developer.android.com/guide/navigation
- Glide: https://github.com/bumptech/glide
- Robolectric: https://github.com/robolectric/robolectric
- Butterknife: https://jakewharton.github.io/butterknife/
- Firebase: https://firebase.google.com/
- Retrofit: https://square.github.io/retrofit/
- Room: https://developer.android.com/topic/libraries/architecture/room