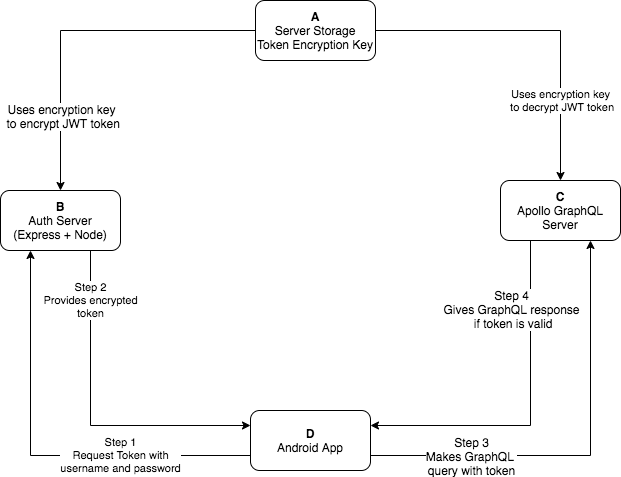
Apollo Client Architecture

Mainly this project has 4 components interacting with each other to achieve the goal. Each of them and their interactions are discussed in detail below.



## Components

All components shown in above diagram are discussed in detail below.

* Component A - Server Storage

For now, server storage is .env file associated with source code in node.js project. .env file is a local project property file, data can be stored as key-value pairs and can never be changed by the code. I have stored encryption key to encrypt tokens, admin auth key (which never expires), JWT token expiry in .env file.

* Component B - Auth server
  + Technical stack
    - Node.js
    - NPMs
      * express - To create post api call for login
      * dot-env - To load configurations from .env file
      * jsonwebtoken - To create encrypted JWT tokens
      * body-parser - To parse json request body in express
  + Installation instructions
    - Install Node.js (ignore if already installed)
    - npm i
    - node auth-server.js
    - Auth server is now running on 3000

We needed auth server to authenticate users with username/password and provide them secure tokens to be used to communicate securely with graphql server. We needed separate auth server because graphql specification does not include/enforce token generation from credentials (i.e. username/password) or validate it. And hence the apollo graphql server just provides a hook to validate the auth token on our own. Auth server is a node js app with express.js. It is a simple post API call, that takes username/password from android app and creates secure JWT auth token with expiry and gives it back. For now, it just locally validates username and password, but that can be verified with any DB, or third party auth service like AWS cognito etc.

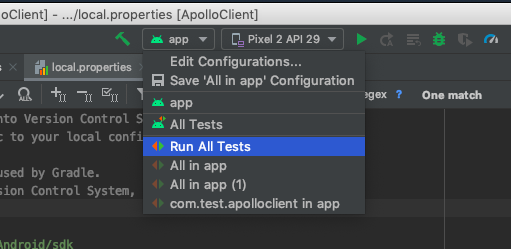
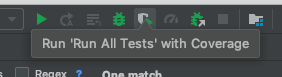
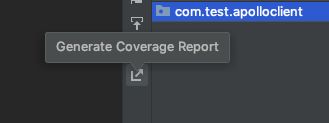
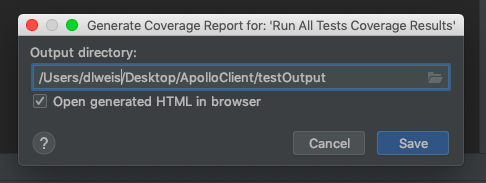
JWT token is Json Web Token, a standard way to maintain user’s sessions and authenticity across different web apps. It is encrypted with a key, which is secure on server(s). All server web apps, that share the user’s session, can have a copy of this key (if they are lying on different machines) and use it to decrypt the token client provides to validate them. For validation, it does two things. 1) If token is valid (whether the server can decrypt it with its own key) 2) if the token is not expired. There is some information also stored in the token itself which is available only after decryption. Information can be roles, access-rights to some actions etc. I have stored GraphQL auth access key (i.e. { loggedIn: true } ) in this token, which Apollo Server verifies after decryption and responds with data only if user has rights.

* component C - Apollo graphql server
  + Technical stack
    - Node.js
    - NPMs
      * jsonwebtoken - To decrypt and verify JWT tokens
      * node-fetch - To make api call to themoviedb.org
      * apollo-server - Actual apollo server dependency
      * dot-env - To load configurations from .env file
  + Installation instructions
    - Install Node.js (ignore if already installed)
    - npm i
    - node index.js
    - Apollo server is now running on 4000

Apollo GraphQL server provides easy way to handle graphql queries. It needs type definitions and query definitions. This is included in its schema. Apollo server provides a mechanism called resolvers, through which data can be resolved and gathered from DB or any third party APIs (in this case [themoviedb.org](http://themoviedb.org)). It verifies captured data with schema and responds to users according to their queries. It automatically determines response graph type based on user’s query request. No additional coding is required for that.

It does not enforce strict authentication mechanism, but it is possible via a context hook. This hook intercepts every requests, and provides authentication header supplied by client. They also provide provision to validate the auth header (i.e. JWT token) in my own way and accept or reject the request.

In this case, for each request, apollo server provides JWT token, which I decrypt with the encryption key, validate and then allow to proceed. If the token is not valid or its expired, graphql query will be responded with error.

* component d - android app
  + Technical stack
    - Android Native with kotlin
    - Major Dependencies
      * androidx.core - Standard compatibility library
      * androidx.constraintlayout - Using constraint layout to design UI
      * okhttp3 - Using as http client
      * picasso - To download images
      * espresso - To support Instrumentation tests
    - Major architectural components
      * Models extending BaseObservables
      * Bindable properties for two way data binding
      * MutableLiveData properties to reflect changes on UI
      * Helpers (mainly for validation and http client)
  + Installation instructions
    - Open project in latest android studio
    - Run project to install on device or emulator
    - To test, click on ‘app’ in task bar and select ‘Run All Tests’
    - Click on shield icon to right of play button, this will run all tests with coverage
    - From new window, click on lower left button, designated by right angle and arrow pointing to upper right
    - Designate output folder for generated coverage report, check ‘Open generated HTML in browser’, click ok

Android app has been made with activities, fragments and view models. Fragments are reusable UI components in native android that can be rendered on a whole screen or in portion of a screen. I have designed login screen and dashboard screen with fragments within activities.

All fragments closely work with models. These models extends to BaseObservable, to use first-class support of android’s two way data binding. I have wired email and password EditTexts with bindable properties, which dynamically update its value whenever user types anything in text field. I also have used MutableLiveData properties to dynamically show validation error messages on UI. All view models interacts with fragments via controller. These controllers make it testable without UI components. Under unit tests, UI can not be completely mocked. Therefore to test actual business logic, actual UI can be isolated with such interfaces. Unit tests uses mocked implementation for such controllers and to verify their integration.

Android app uses okHttp3 library to make Http API calls. HttpHelper class handles and provides client to different components. I have used apollo-android client library to communicate with apollo server. It uses same okHttp3 client to make calls.

Android app makes POST api call to auth-server first with okHttp3 and gets secure token. That token is used again to make GraphQL query with apollo-android client to get movies data. All view components (i.e. fragments) are updated via interfaces to update UI if there is any data or show error if there is any.

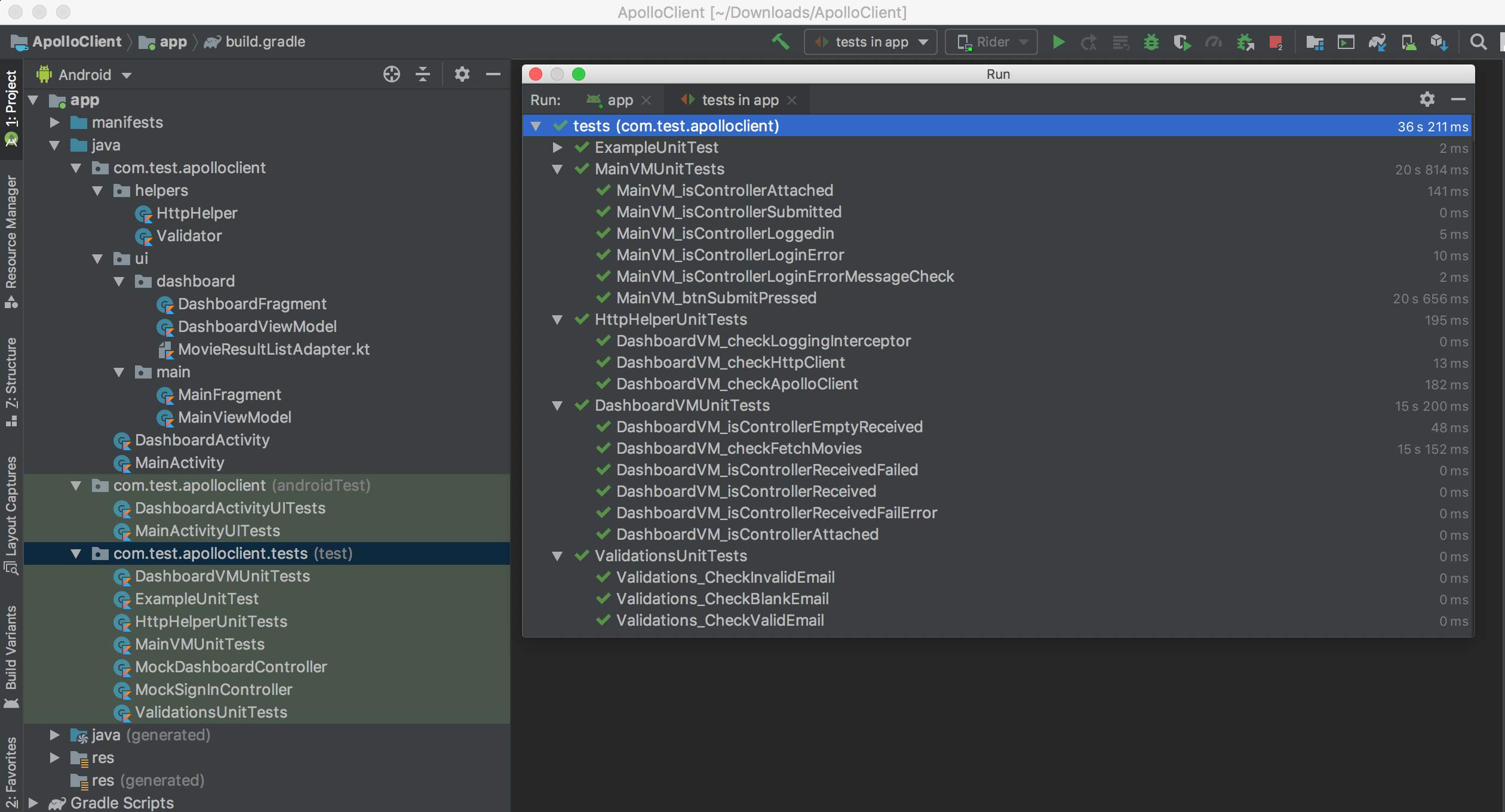
I have used unit tests as well as instrumentation tests. Unit tests checks business logic and integration tests checks its integration with UI via mocked controllers. Instrumentation tests actually runs app in real device/emulator and runs app automatically (based on UI automation tests written) and checks UI components (fragments and activities) in pieces. Here also I have to mock some things as functional pieces are tested independently. I have used admin token for graphql queries independently (without login).

## Call flow

* step 1 - Request Token (Android app >> auth server)
  1. Andriod app sends post call to auth server. Post body contains username and password
  2. It uses okhttp3 enqueue method to make API call. We use enqueue because it is cancellable. We cancel the current call in case of any error. Also it is async.
* step 2 - Provide token (auth server >> android app)
  1. Auth server validates supplied username/password locally and generates a standard JWT token with expiry. Auth server sends user object with token in response. In case of any error, it sends error response.
  2. Auth server uses “jsonwebtoken” npm library to create or verify JWT tokens.
  3. It uses configuration variables to encrypt token and set the expiry. All required configuration variables are defined in .env file.
  4. Once android app receives token, it caches in memory to be used later and shows movie list screen.
  5. If there is any error, app shows appropriate alert dialog and remains on same screen.
* step 3 - query graphql (android app >> apollo server)
  1. Android app sends graphql query to Apollo Server with the token cached in step 2.4 above
  2. Apollo server validates supplied token with getUser() hook and throw AuthenticationError in case getUser() method does not respond with a valid user.
  3. Apollo server has a resolver defined to fetch all popular movies and their reviews. Resolver makes call to [themoviedb.org](http://themoviedb.org) via their apis. Fetches all popular movies, and its reviews and respond to android app from single generated graph.
* step 4 - provide popular movies (apollo server >> android app)
  1. Android app receives graph response and renders list in recycler view.
  2. If there is any authentication error, android app parses the error response and show it to the user on screen.

Tests

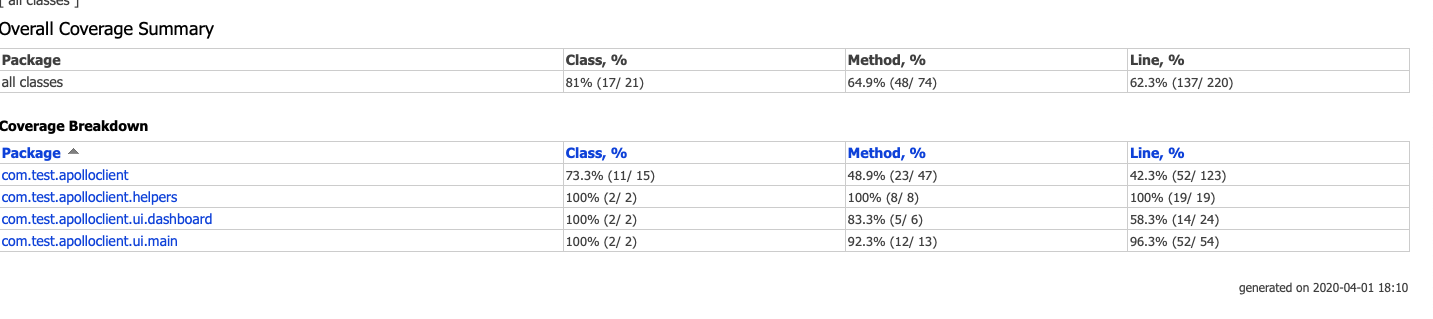
App has both unit test cases and instrumentation tests. There are 19 unit tests 2 instrumentation tests. Snapshot of all test cases executed.



* MainVMUnitTests
  + MainVMUnitTests.MainVM\_isControllerAttached
  + MainVMUnitTests.MainVM\_isControllerSubmitted
  + MainVMUnitTests.MainVM\_isControllerLoggedin
  + MainVMUnitTests.MainVM\_isControllerLoginError
  + MainVMUnitTests.MainVM\_isControllerLoginErrorMessageCheck
  + MainVMUnitTests.MainVM\_btnSubmitPressed - Checks all business logic with email and password validations and actual login with real API call.
* HttpHelperUnitTests
  + HttpHelperUnitTests.DashboardVM\_checkLoggingInterceptor
  + HttpHelperUnitTests.DashboardVM\_checkHttpClient
  + HttpHelperUnitTests.DashboardVM\_checkApolloClient
* DashboardVMUnitTests
  + DashboardVMUnitTests.DashboardVM\_isControllerEmptyReceived
  + DashboardVMUnitTests.DashboardVM\_checkFetchMovies - Fetches popular movies with actual graphql call and verifies its integrity with UI components
  + DashboardVMUnitTests.DashboardVM\_isControllerReceivedFailed
  + DashboardVMUnitTests.DashboardVM\_isControllerReceived
  + DashboardVMUnitTests.DashboardVM\_isControllerReceivedFailError
  + DashboardVMUnitTests.DashboardVM\_isControllerAttached
* ValidationsUnitTests
  + ValidationsUnitTests.Validations\_CheckInvalidEmail
  + ValidationsUnitTests.Validations\_CheckBlankEmail
  + ValidationsUnitTests.Validations\_CheckValidEmail

Code coverage

Code coverage is how much code is covered under tests. Whether it be unit tests or Instrumentation tests. I have both unit and instrumentation tests, but the coverage values varies according to the tool as it differs on what packages it considers for coverage.



Improvements

* Auth can be done via DB or some proper third party auth service like AWS cognito or Auth0
* Better UI framework can be used like material UI to make it look better. Can be organized better with dimensions and styles if we have more number of screens and components
* All Http calls from log interceptor of okhttp3 can be persisted on storage to review logs later
* I may have used more mock objects to test all corner cases, more specific Instrumentation tests if more number of features are there.
* …