Tampere University of Applied Sciences

Review Movie Application

TIEN DAT NGUYEN

BACHELOR'S THESIS
May 2023

Bachelor of Engineering
Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Bachelor of Engineering

Tien Dat Nguyen:
Review Movie Application

Bachelor's thesis 37 pages
May 2023

This thesis presents the development of the Review Movie application, which is an entertainment application, and indicates which method and architecture have been used in its development. The purpose of this project was to create an application where people can find entertainment after stressful working hours. This application provides users with information, ratings, and genres of movies so they can find the right movies among the ever-growing selection of titles.

To build this application, Model-View-ViewModel architecture was used, which is known as the optimal synthesis of today's provider patterns. The database for the application was taken from the Application Programming Interface of https://www.themoviedb.org/, and by using Retrofit, which is a type-safe REST client for Android and Java, to make it easier to use the application programming interface efficiently. Android Studio was used in this thesis.

With the use of Retrofit and the Model-View-ViewModel, the Review Movie application has the ability to separate concerns and responsibilites in the codebase, making it easier to maintain and update the application. The result of combining Retrofit and Model-View-ViewModel patterns is a robust and scalable codebase. In addition, this combination allows for efficient data binding, which means that changes to the data in the ViewModel automatically update the user interface.

Key words: retrofit, mvvm, review movie application

CONTENTS

ABBREVIATIONS AND TERMS

API	Application Programming Interface
Gson	Java library that can be used to convert Java Objects into their JSON representation or convert a JSON string to an equivalent Java object
IDE	Integrated Development Environment
JSON	Javascript Object Notation
MVVM	Model-View-View Model
OS	Operating system
Retrofit	Type-safe REST client for Android and Java
SDK	Software Development Kit
TAMK	Tampere University of Applied Sciences
UI	User Interface
VCRs	Video Cassette Recorders

1 INTRODUCTION

1.1 History of movie industry

The film industry began in the late 19th century with the invention of motion-picture cameras and the cinématographe projector (University of Minnesota Libraries, 2019). These innovations allowed for the creation of moving images that captured the attention of audiences around the world. By 1915, Hollywood had become the center of the industry, and the introduction of talking films in the late 1920s marked a significant milestone.

Following World War II, the invention of television led to the development of new genres and formats, such as sitcoms and dramas. The adoption of Video Cassette Recorders in the 1980s marked another turning point, as audiences shifted from movie theaters to home viewership (Encyclopædia Britannica, n.d.). Today, the film industry continues to thrive with diverse movie themes and advanced technologies that create high-quality movies that captivate viewers with stunning visual effects and immersive sound.

Despite the challenges it has faced over the years, the film industry continues to captivate audiences worldwide. From the early days of motion-picture cameras to today's advanced technologies, industry has undergone numerous transformations and adaptations. It has set the foundation for prestigious awards ceremonies such as the Oscars, which celebrate the best films and performances of the year.

1.2 Project background and purpose

The movie industry originated and developed thanks to the development of technology with the inventions of contributed inventors and the creative ideas of film makers. It was established to meet the demand of society, with countless of purposes: from entertainment demand to the excited feelings that was actived by the substances released to stimulate the brain - the dopamine, the seratone, the oxytocin.

On the back side, with countless movie themes for the selection of the audiences, that would make it hard for them to easily pick out the ones that meet all their standards.

With the fast pace of modern life, people have strict routines and very little spare time, so with the invention of an application that combines all in one feature to review the movies would be the one the audiences are waiting for. To have better interaction with the audiences, an application on a smartphone store would be the best tool to reach the audience.

In this thesis, an entertainment application called the "Review Movie Application" has been developed. The application is designed to provide users with an enjoyable and relaxing experience directly on their mobile device. The goal of the application is to offer users a convenient way to access a wide range of movies, providing them with hours of entertainment.

2 APPLICATION REQUIREMENTS

2.1 User requirements

With the application users should be able to search and watch the hottest movies, classic or popular TV shows from many countries. Moreover, this application also provides other features such as viewing reviews, rankings, descriptions, and actors of the movies. Especially, this application should provide a feature that user can create an account to save their favorite movies in their playlist. From this we can draw user requirements for the application.

These requirements are:

- Application will start when click on the icon.
- Application shows the home page after loading.
- Application must provide the list of categories for user.
- Users can click on every single item to see more details.
- Users can see information for each item.
- Users can register an account through button sign up.
- Users can login using username and password through application's user interface (UI).
- After signing in, the playlist feature appears.
- Users can add the movie as their favorite movie in the playlist.
- Users can see all the movies in their favorite playlist and remove it.
- Users can sign out.

2.2 Technical requirements

Some challenges need to be solved during the application development to make sure it runs successfully. The biggest challenge with the application is the use of data traffic between the server and the user, accessing an API involves utilizing the HttpUrlConnection library, which is made available through the Android Software Development Kit (SDK), to perform operations with the API. However, there are some problems. The speed of data transfer may be delayed, creating a jerky

or laggy impression, which can result in increased consumption of system resources and reduced application performance. Along with that, the library does not support Restful API directly, which means that making HTTP requests will be difficult for developers.

Fortunately, there is a better library to optimize those problems; Retrofit is built on top of okHttp by Square developer, which provides convenient features for solving these problems (Figure 1. Data flow of Retrofit.). The diagram below shows that the application (My app) passes parameters to the Retrofit library for use in an API call. Retrofit uses the parameters to construct an HTTP request using the appropriate method (such as GET, POST, PUT, or DELETE) and sends it to the API endpoint. The API receives the HTTP request and processes it, generating a JSON or XML response. Retrofit receives the response and converts it into a Kotlin object based on the specified data model. The Kotlin object is returned to the application, where it can be processed and displayed to the user.

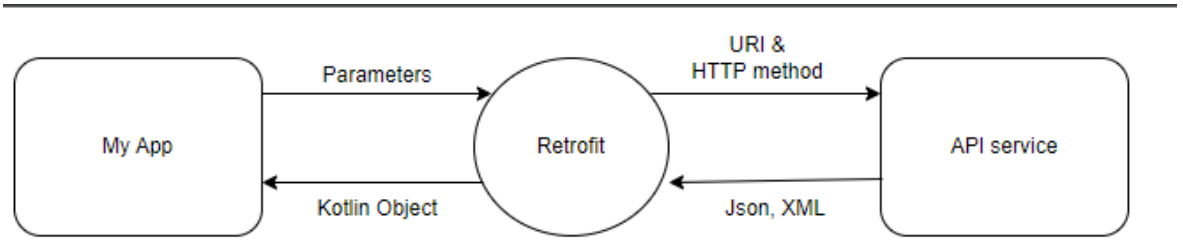


Figure 1. Data flow of Retrofit.

Besides, another problem when deploying a project is too many files and functions, which can make it a little dificult for developers to manage and work. To address this challenge, one technology was developed called Model-View-View Model (MVVM) architecture (Figure 2. MVVM architecture (GeeksforGeeks). With the logical separation of the application into separate parts of this pattern, it is easier to develop and manage the project. From the Figure 2. MVVM architecture (GeeksforGeeks), it can be observed that the View layer is responsible for displaying data to the user and receiving user input. The ViewModel layer acts as an intermediary between the View and the Model layers. It retrieves data from the Model and prepares it for display in the View. The Model layer is responsible for

retrieving data from a data source, such as a database or API. The arrows in the figure show the flow of data between the layers.

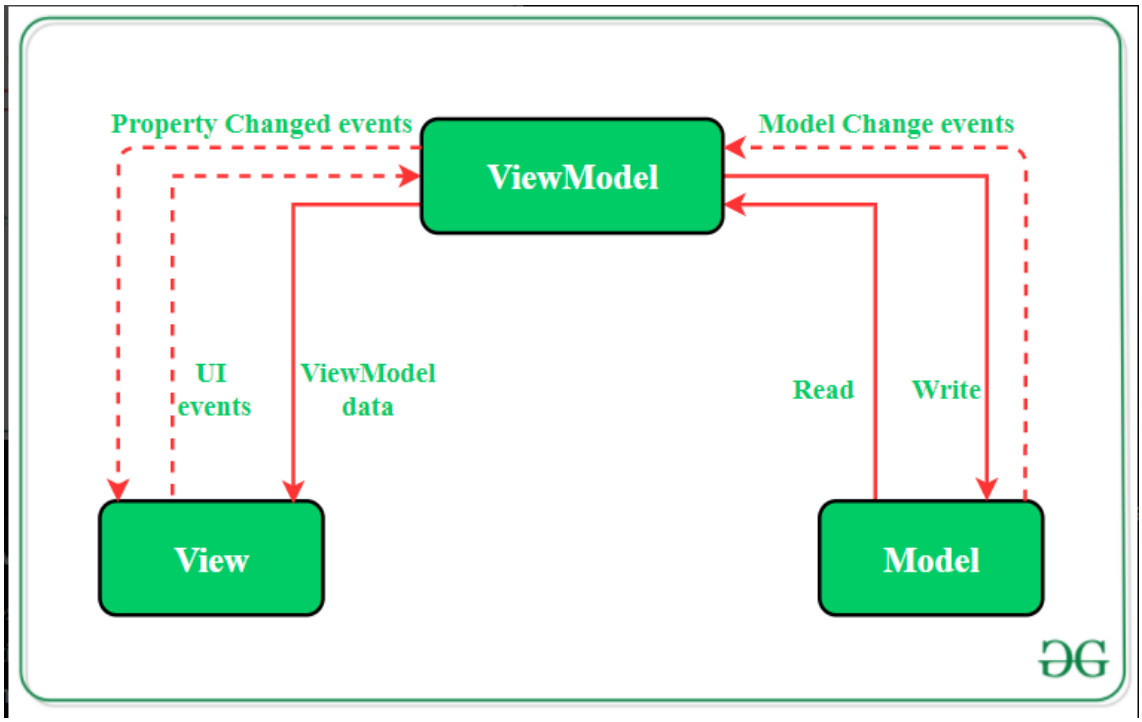


Figure 2. MVVM architecture (GeeksforGeeks)

3 TECHNOLOGIES AND TOOLS

3.1 Kotlin

Kotlin is a cross-platform programming language developed by JetBrains in 2011 and introduced in 2016 (Android Developers, 2016). With the goal of replacing Java by improving security, increasing performance and reducing complexity, Kotlin was designed as a statically-typed language that can be compiled into Java bytecode or JavaScript. Developers can use the Kotlin language together with Java in the same project because they can interact with each other.

The merits of Kotlin, such as its concise syntax, null safety features, and seamless interoperability with existing Java codebases, have made it a popular language for Android apps and web development. Kotlin is also supported by big companies like Google, JetBrains, and Gradle. For that, Kotlin is a good choice for this project.

3.2 Retrofit

Retrofit has been briefly summarized in term of affects as well as benefits above and can be seen at Figure 1. Data flow of Retrofit.. Retrofit is an open-source library for Android development. It is a type-safe HTTP client for Android and Java that allows developers to consume Restful APIs by mapping HTTP requests and responses to Java objects. Built on top of the OkHTTP library by Square, MVVM provides a simple interface for making network requests easily (DigitalOcean, 2018).

Retrofit also has a simple and easy-to-use interface that allows developers to make network requests easily. Additionally, it uses annotations to add metadata to the API interfaces, making the code easier to read and understand. Furthermore, Retrofit is built on top of OkHTTP, a widely-used HTTP client library, which provides support for features such as connection pooling, transparent compression, and response caching.

Overall, choosing Retrofit for API integration in Android development offers several benefits, including improved code maintainability, reduced errors during development, and ease of use.

3.3 MVVM architecture

The MVVM architecture pattern is known as an architecture widely used in developing UI for applications today. The pattern is derived from Microsoft and is also a variation of the MVC pattern. The MVVM architecture is divided into three components: Model-View-ViewModel (Techtarget, 2019).

The MVVM architecture provides a clear separation of concerns between the different components of an application, making it easier to maintain and extend the codebase. MVVM makes it easy to test the business logic of the application by isolating it from the UI. This leads to more reliable and predictable code and reduces the likelihood of bugs. Additionally, MVVM allows for more flexibility in the UI, as it can be updated independently of the data and business logic.

In general, the benefits of using MVVM include improved maintainability, scalability, testability, and flexibility. These benefits make it a popular choice for Android development, especially for larger and more complex applications.

3.4 Android Studio

Android Studio is the official Integrated Development Environment (IDE) for Android application development. Based on the powerful code editor and developer tools from IntelliJ IDEA, Android Studio offers even more features that enhance their productivity when building Android apps (Android Developers, 2016). Android Studio offers Android developers key benefits such as:

- Provides complete and powerful Android application development tools and features.
- Has a user-friendly interface.
- Integrate with the Android SDK, allowing convenient creation and management of Android projects.

- Supports Java and Kotlin programming.
- Features code autocomplete and error checking.
- Provides a suite of Android application testing and debugging tools.
- Integrate with Android Emulator, allowing to test and test the application on a virtual device.
- Allows creation and management of different versions of the application.
- Ability to integrate with version management tools and source code hosting services such as Git, SVN, and GitHub.

4 TECHNICAL IMPLEMENTATION

4.1 MVVM (Model-View-ViewModel)

The application was developed using the MVVM pattern. A TabFragment file was created to serve as the View component of the application. Its main role is to present data to the user. Please refer to Picture 1 for the implementation code of TabFragment. Code explanation will be provided below (Picture 7).

```
override fun initViewModel() {
    tabViewModelFactory = TabViewModelFactory(homeType, lifecycleOwner: this)
    tabViewModel = ViewModelProvider( owner: this, tabViewModelFactory)[TabViewModel::class.java]
    binding.lifecycleOwner = viewLifecycleOwner
    binding.viewModel = tabViewModel


    //display loading progress
    tabViewModel.loading.observe( owner: this, Observer { it: Boolean!
        if (it) {
            showProgress()
        } else {
            hideProgress()
        }
    })
    //display empty layout
    tabViewModel.showEmptyEvent.observe( owner: this, Observer { it: Boolean!
        binding.llEmptyLayout.visibility = if (it) View.VISIBLE else View.GONE
    })

    //display refresh indicator
    tabViewModel.showRefreshEvent.observe( owner: this, Observer { it: Boolean!
        binding.swipeRefresh.isRefreshing = it
    })

    //item click event
    tabViewModel.itemClickEvent.observe( owner: this) { it: Movie!
        context?.let { context ->
            val movie = it as Movie
            //open details screen
            val screen = DetailsFragment::class.java.name
            val intent = SecondaryActivity.createIntent(context)
            //bundle
            val bundle = Bundle()
            bundle.putParcelable(DetailsFragment.KEY_MOVIE, movie)
            //pass the screen key to bundle
            intent.putExtra(SecondaryActivity.KEY_SCREEN, screen)
            intent.putExtra(SecondaryActivity.KEY_BUNDLE, bundle)
            //start activity
            startActivity(intent)
        }
```

Picture 1. Initialize View for the app.

A folder containing all the models required for the application was created as shown in Picture 2.

Picture 2. Data Model

For instance, the Cast object (Picture 3) was used as an example to implement corresponding objects marked with different annotations. In this case, the @SerializedName annotation was used to specify the name when converting between Java object and Javascript Object Notation (JSON). This mapping enables the corresponding field in the JSON object to parse data when using the Gson library.

```
data class Cast(
    @SerializedName("adult")
    var adult: Boolean?,
    @SerializedName("cast_id")
    var castId: Int?,
    @SerializedName("character")
    var character: String?,
    @SerializedName("credit_id")
    var creditId: String?,
    @SerializedName("gender")
    var gender: Int?,
    @SerializedName("id")
    var id: Int?,
    @SerializedName("known_for_department")
```

Picture 3. Cast object

In order to transfer data between the View and Model components, the ViewModel was utilized to facilitate the process.

The LiveEvent class was used to retrieve and store data from the Movie Model as depicted in Picture 4. Three classes were created to inherit the LiveEvent class: showEmptyEvent, showRefreshEvent, and itemClickEvent.

```
class TabViewModel(val homeType: HomeType, lifecycleOwner:
    var movieRepository: MovieRepository
    var itemClickEvent = LiveEvent<Movie>()
    var showRefreshEvent = LiveEvent<Boolean>()
    var showEmptyEvent = LiveEvent<Boolean>()

    init {
        val movieService = MovieService.getInstance()
        movieRepository = MovieRepository(movieService)

        initData()
    }
```

Picture 4. Implement ViewModel

The code was then implemented to interact with the data and store it in the adapter, as shown in Picture 5. The items represent the data in this context.

```
override fun initData() {
    //init adapter
    adapter = HomeMoviesAdapter(items)
    (adapter as HomeMoviesAdapter).itemClickEvent = itemClickEvent

    //display loading for initialization
    loading.postValue( value: true)
    //load data
    loadData()

   super.initData()
}
```

Picture 5. Get and store data in adapter

Following the data acquisition, This code is a method called loadData() (Picture 6) that loads data from an API and updates the UI accordingly. It sets a flag to

indicate that loading is in progress, then uses a coroutine to make the API call on a background thread. Once the response is received, it updates the UI on the main thread by adding the results to a list, checking if more data can be loaded, and showing an empty layout if the list is empty. It also updates flags to indicate loading and refreshing are finished.

```
override fun loadData() {
    super.loadData()
    //set flag for loading
    isLoading = true
    //call api
    job = CoroutineScope( context: Dispatchers.IO + exceptionHandler).launch {

        //load the movies follow the home type
        val response = when (homeType) {
            HomeType.POPULAR -> movieRepository.getPopular(page)
            HomeType.NOW_PLAYING -> movieRepository.getNowPlaying(page)
            HomeType.UPCOMING -> movieRepository.getUpcoming(page)
            HomeType.TOP_RATED -> movieRepository.getTopRated(page)
        }

        //handle response
        withContext(Dispatchers.Main) { this: CoroutineScope
            if (response.success != false) {
                addItems(response.results)

                //check can load more
                canLoadMore = (response.totalPages ?: 0) > page

                //show empty layout
                showEmptyEvent.postValue(items.isEmpty())
            } else {
                onError(response.statusMessage)
            }
            //hide loading
            loading.postValue( value: false)

            //hide refresh
            showRefreshEvent.postValue( value: false)

            //reset flag value
            isLoading = false
```

Picture 6. Load data

Once the data was displayed on the View side, the observe() method is used to monitor changes in the data of the ViewModel and update the UI accordingly. The loading, showEmptyEvent, and showRefreshEvent variables are observed for changes and trigger the appropriate UI updates when their values change. The itemClickEvent variable is also observed to handle click events on a specific item and open a new activity with the selected item's details (as shown in Picture 7).

```
//display empty layout
tabViewModel.showEmptyEvent.observe( owner: this, Observer { it: Boolean!
    binding.llEmptyLayout.visibility = if (it) View.VISIBLE else View.GONE
})

//display refresh indicator
tabViewModel.showRefreshEvent.observe( owner: this, Observer { it: Boolean!
    binding.swipeRefresh.isRefreshing = it
})

//item click event
tabViewModel.itemClickEvent.observe( owner: this) { it: Movie!
    context?.let { context ->
        val movie = it as Movie
        //open details screen
        val screen = DetailsFragment::class.java.name
        val intent = SecondaryActivity.createIntent(context)
        //bundle
        val bundle = Bundle()
        bundle.putParcelable(DetailsFragment.KEY_MOVIE, movie)
        //pass the screen key to bundle
        intent.putExtra(SecondaryActivity.KEY_SCREEN, screen)
        intent.putExtra(SecondaryActivity.KEY_BUNDLE, bundle)
        //start activity
        startActivity(intent)
```

Picture 7. Data Observations

In simple terms, the ViewModel retrieves data from the Model and transfers it to the View. When there is a change in the data, the View updates the UI through the ViewModel.

4.2 Retrofit

4.2.1 Retrofit Service

Retrofit technology has been instrumental in facilitating communication with APIs. To configure and use Retrofit, the Retrofit library must be added to Gradle, as depicted in Picture 8.

```
//retrofit for api service conversation with BE
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-gson:2.9.0"
implementation "com.squareup.retrofit2:converter-scalars:2.9.0"
```

Picture 8. Import Retrofit library

Since Retrofit was built on top of the OkHttp library, some of its methods were utilized. Additionally, the Gson library was also employed to simplify data transfer (Picture 9).

```
//okhttp for retrofit logger
implementation "com.squareup.okhttp3:okhttp:4.10.0"
implementation "com.squareup.okhttp3:logging-interceptor:4.9.0"

//gson for model parse
implementation "com.google.code.gson:gson:2.9.0"
```

Picture 9. Import OkHTTP and Gson library

The API utilized for the application was obtained from https://www.themoviedb.org/. Initially, registration was required, and the api-key was obtained from the server. Once all the necessary APIs were available, a Gradle configuration was created for development as shown in Picture 10 to simplify the application development process, minimize development time, and enable easy management.

```
productFlavors {
    develop {
        buildConfigField("String", "APIServer", "\"https://api.themoviedb.org/3/\"")
        buildConfigField("String", "BaseImageUrl", "\"https://image.tmdb.org/t/p/w500\"")
        buildConfigField("String", "BaseYoutubeUrl", "\"https://www.youtube.com/watch?v= \"")
        buildConfigField("String", "BaseVimeoUrl", "\"https://vimeo.com/  \"")
        buildConfigField("String", "APIKey", "\"f7dc6370f39062b33006ec32484d2ca9\"")

        flavorDimensions "default"
    }
}
```

Picture 10. Configuration field

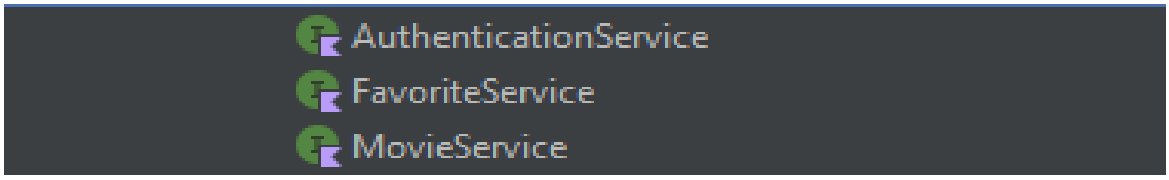
The next step involves building the API model, referred to as RetrofitService as shown in Picture 11, which will be utilized by other services to communicate with the API. This is a code snippet for adding an interceptor to an instance of OkHttpClient in an Android application. The interceptor adds a base query and header to each outgoing HTTP request, including an API key and optionally a session ID. The original request URL is modified using a builder object, and the modified request is then returned by the interceptor. The modified request is then used to proceed with the network call. This is commonly used in API communication to ensure authentication and authorization for API requests.

```
fun getInstance(): Retrofit? {
    if (retrofit == null) {
        val builder = OkHttpClient.Builder()

        //add base query and header
        builder.addInterceptor(Interceptor { chain ->
            val request: Request.Builder = chain.request().newBuilder()
            val originalHttpUrl = chain.request().url

            //add api key
            val builder = originalHttpUrl.newBuilder()
                .addQueryParameter( name: "api_key", BuildConfig.APIKey)

            //add session id
            MyApplication.instance().session?.sessionId?.let {  it: String
                builder.addQueryParameter( name: "session_id", it)
            }

            request.url(builder.build())
            chain.proceed(request.build())  ^Interceptor
        })
```

Picture 11. RetrofitService model

4.2.2 API Service

The application features three main functions: movie data display, login, and favorite movie storage. To manage each of these features, three services were implemented (Picture 12).



Picture 12. API services

Subsequently, implementation was carried out for each file, starting with the authentication service (Picture 13).**Error! Reference source not found.**

```
7   interface AuthenticationService {
8       companion object {
9           var authenticationService: AuthenticationService? = null
10          fun getInstance(): AuthenticationService {
11              if (authenticationService == null) {
12                  val retrofit = RetrofitService.getInstance()
13                  authenticationService = retrofit?.create(AuthenticationService::class.java)
14              }
15              return authenticationService!!
16          }
17      }
18
19      @GET("authentication/token/new")
20      suspend fun requestToken(): TokenResponse
21
22      @POST("authentication/token/validate_with_login")
23      suspend fun requestLogin(@Body loginRequest: LoginRequest): TokenResponse
24
25      @POST("authentication/session/new")
26      suspend fun createSession(@Body loginRequest: LoginRequest): SessionResponse
27
28      //the retrogit does not allow DELETE with request body
29      //so we do the trick delete with body
30      @HTTP(method = "DELETE", path = "authentication/session", hasBody = true)
31      suspend fun deleteSession(@Body request: DeleteSessionRequest): SessionResponse
32
33      @GET("account")
34      suspend fun getDetails(): AccountResponse
35  }
```

Picture 13. Authentication service

The getInstance() function (line 12) is responsible for returning an instance of AuthenticationService. It achieves this by creating a new instance or retrieving an

existing one if it already exists. If the AuthenticationService instance is null, the function obtains an instance of RetrofitService, creates a new AuthenticationService instance using the create() function, assigns it to the AuthenticationService property, and finally returns the AuthenticationService instance. The annotations present in the code are used to interact with the API.

The FavoriteService (Picture 14) and MovieService (Picture 15) classes also have a getInstance() function that returns an instance of the corresponding service. Annotations are used to interact with the API in these services, and they are similar to the ones used in the AuthenticationService. The FavoriteService is responsible for managing users' favorite movies, while the MovieService is used to retrieve movie data from the API.

```
interface FavoriteService {
    companion object {
        var favoriteService: FavoriteService? = null
        fun getInstance(): FavoriteService {
            if (favoriteService == null) {
                val retrofit = RetrofitService.getInstance()
                favoriteService = retrofit?.create(FavoriteService::class.java)
            }
            return favoriteService!!
        }
    }

    @POST("account/{accountId}/favorite")
    suspend fun markFavorite(@Path("accountId") accountId: Int,
                             @Body request: FavoriteMovieRequest): BaseModel

    @GET("account/{accountId}/favorite/movies")
    suspend fun getFavoriteMovies(@Path("accountId") accountId: Int,
                                  @Query("page") page: Int): ResponseModel<List<Movie>>

    @GET("movie/{movieId}/account_states")
    suspend fun getMovieState(@Path("movieId") movieId: Int): MovieState
}
```

Picture 14. Favorite Service

```
interface MovieService {
    companion object {
        var movieService: MovieService? = null
        fun getInstance(): MovieService {
            if (movieService == null) {
                val retrofit = RetrofitService.getInstance()
                movieService = retrofit?.create(MovieService::class.java)
            }
            return movieService!!
        }
    }

    @GET("search/movie")
    suspend fun searchMovies(@Query("query") query: String): ResponseModel<List<Movie>>

    @GET("movie/popular")
    suspend fun getPopular(@Query("page") page: Int): ResponseModel<List<Movie>>

    @GET("movie/now_playing")
    suspend fun getNowPlaying(@Query("page") page: Int): ResponseModel<List<Movie>>

    @GET("movie/top_rated")
    suspend fun getTopRated(@Query("page") page: Int): ResponseModel<List<Movie>>

    @GET("movie/upcoming")
    suspend fun getUpcoming(@Query("page") page: Int): ResponseModel<List<Movie>>
```

Picture 15. Movie Service

The API can be used to access MovieService for searching and displaying movie information. AuthenticationService offers token requests for user login. Additionally, users can store favorite movies using the FavoriteMovie feature in their personal account.

4.3 User authentication

Creating a separate authentication system for this app is not necessary. Instead, the API can be used to create one by following three steps:

Step 1: Request token (Picture 16)

```
private fun requestToken() {
    job = CoroutineScope( context: Dispatchers.IO + exceptionHandler).launch { this: CoroutineScope
        val response = authenticationRepository.requestToken()
        withContext(Dispatchers.Main) { this: CoroutineScope
            if (response.success != false) {
                requestToken = response.requestToken
            } else {
                onError(response.statusMessage)
                loading.postValue( value: false)
            }
        }
    }
}
```

Picture 16. Request Token

Tokens can be requested through the API using authenticationRepository.requestToken(), which provides a temporary token to ask the user for permission to access their account. After obtaining the token, proceed to step 2.

Step 2: Ask the user for permission (Picture 17)

```
fun login() {
    loading.postValue( value: true)
    val request = LoginRequest(username.value, password.value, requestToken)
    job = CoroutineScope( context: Dispatchers.IO + exceptionHandler).launch { this: CoroutineScope
        val response = authenticationRepository.login(request)
        withContext(Dispatchers.Main) { this: CoroutineScope
            if (response.success != false) {
                requestToken = response.requestToken

                generateSession()
            } else {
                onError(response.statusMessage)
                loading.postValue( value: false)
            }
        }
    }
}
```

Picture 17. Ask for user's permission

Once the request token is obtained, the user can be directed to the following URL to approve the request token via the sign-in button. Clicking the button indicates the user's approval and allows for proceeding to step 3.

Step 3: Create a session ID (Picture 18)

```
private fun generateSession() {
    val request = LoginRequest( username: null,  password: null, requestToken)
    job = CoroutineScope( context: Dispatchers.IO + exceptionHandler).launch {  this: CoroutineScope
        val response = authenticationRepository.createSession(request)
        withContext(Dispatchers.Main) {  this: CoroutineScope
            if (response.success != false) {
                val sessionId = response.sessionId

                //save session
                val session = Session(sessionId)
                MyApplication.instance().saveSession(session)

                //get account details
                getAccountDetails()
            } else {
                onError(response.statusMessage)
                loading.postValue( value: false)
            }
        }
    }
}
```

Picture 18. Create a session ID

After the user clicks the sign-in button in step 2, the following function will be initiated in step 3. The server will return a new session ID, which can be used to write user data. If the session is valid, the user can log in. Otherwise, an error message will be displayed.

To ensure information security, session encryption is necessary. The Hawk library with the function Hawk.put() (as shown in Picture 19) was utilized for this purpose. When passing the session, this function automatically encrypts and saves it in the machine's memory. This library simplifies the encryption process for developers and ensures the confidentiality and safety of the information

```
// 2. Encrypt the text
String cipherText = null;
try {
  cipherText = encryption.encrypt(key, plainText);
  log("Hawk.put -> Encrypted to  " + cipherText);
} catch (Exception e) {
  e.printStackTrace();
}
if (cipherText == null) {
  log("Hawk.put -> Encryption failed");
  return false;
}
```

Picture 19. Function of Hawk.put() from library (Obut, Orhan. Hawk)

4.4 Android Youtube Player

The application features a trailer viewing function in addition to the ability to read information and reviews for each movie. During API exploration, related videos were discovered, all of which are from YouTube. To enable video playback, the androidyoutubeplayer library was utilized (Picture 20).

```
//youtube lib
implementation 'com.pierfrancescosoffritti.androidyoutubeplayer:core:11.1.0'
```

Picture 20. Youtube library integration (Pierfrancesco Soffritti)

The API provides the key for the video, which can be used for the trailer feature. In the createIntent() function, the method arguments were utilized to pass the key to the dialog as shown in Picture 21.

```
fun createIntent(keyId: String): TrailerViewDialog {
    val dialog = TrailerViewDialog()
    // Supply num input as an argument.
    val args = Bundle()
    args.putString(KEY_KEY_ID, keyId)
    dialog.arguments = args
    return dialog
}
```

Picture 21. Pass key using argument

When the user clicks on the button to play trailer, dialog will appear on the device via the dialog.show method (Picture 22).

```
    //btn play trailer
    binding.btnPlay.setOnClickListener {  it: View!
        detailsViewModel.trailerVideo?.key?.let {  it: String
            val dialog = TrailerViewDialog.createIntent(it)
            dialog.show(parentFragmentManager, TrailerViewDialog::class.java.name)
        }
    }
}
```

Picture 22. Show dialog on device

Once the dialog appears, the onCreateView method will be initiated to construct the fragments according to the specified layout. The layout, which can be seen on line 75 as "dialog_trailer_view" in Picture 23, is used for this purpose.

```
66      override fun onCreateView(
67          inflater: LayoutInflater,
68          container: ViewGroup?,
69          savedInstanceState: Bundle?
70      ): View {
71          arguments?.let {  it: Bundle
72              videoId = it.getString(KEY_KEY_ID)
73          }
74
75          _binding = DataBindingUtil.inflate(inflater, R.layout.dialog_trailer_view, container,
76          binding.executePendingBindings()
77          return binding.root
78      }
```

Picture 23. Create view for items.

The method "onViewCreated" will execute right after the dialog's lifecycle. Within this method, the YouTube PlayerView is utilized, and the "loadVideo" method is called with the videoId key that was passed earlier, triggering the feature trailer video to launch (Picture 24).

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    videoId?.let { videoId ->
        lifecycle.addObserver(binding.youtubePlayerView)
        binding.youtubePlayerView.addYouTubePlayerListener(object : AbstractYouTubePlayerListener() {
            override fun onReady(youTubePlayer: YouTubePlayer) {
                youTubePlayer.loadVideo(videoId, startSeconds: 0f)
            }
        }) ^let
    }
```

Picture 24. Using YoutubePlayer to watch video

4.5 Review Movie application development

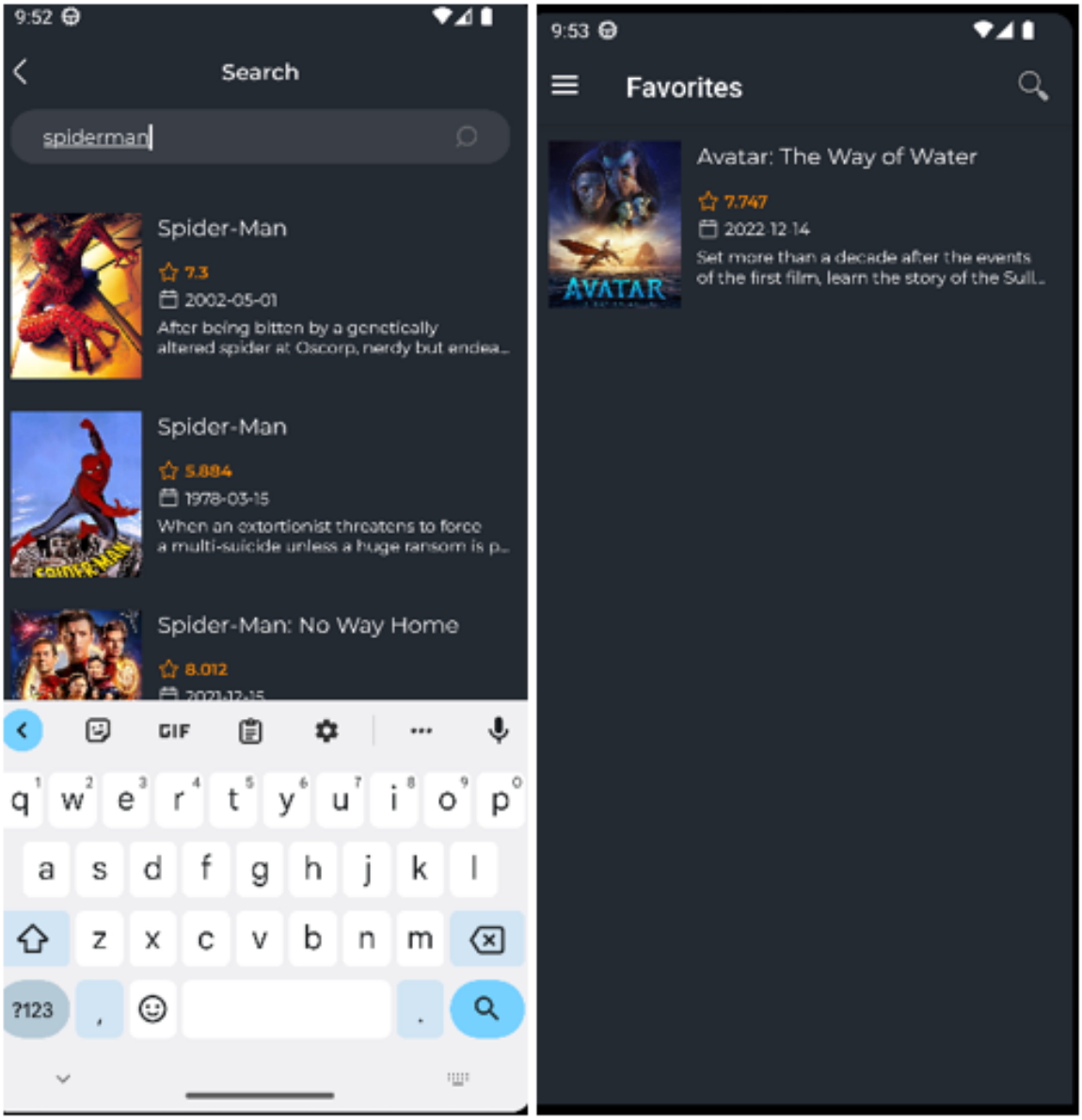
The development of the Review Movie Application was aided using the MVVM pattern, a popular design pattern used in software development. The MVVM pattern helps to simplify the application architecture by separating the application components into three distinct layers: the data (Model), the interface (View), and the logic (View-Model).

By separating these components, the MVVM pattern helps to make the codebase cleaner and easier to maintain. The Model layer represents the data, which can be obtained from various sources such as a database or web service. The View layer represents the UI, which interacts with the user to display data and receive input. The View-Model layer acts as a bridge between the Model and View layers, providing the logic and data necessary to update the View based on changes in the Model.

In addition to using the MVVM pattern, the Review Movie Application also utilizes Retrofit to access the API. Retrofit is a type-safe HTTP client for Android and Java, which greatly simplifies the process of making network requests and processing the response data. Retrofit helps to reduce the amount of code needed to access the API, which saves time and reduces the likelihood of errors.

4.6 Graphical user interface

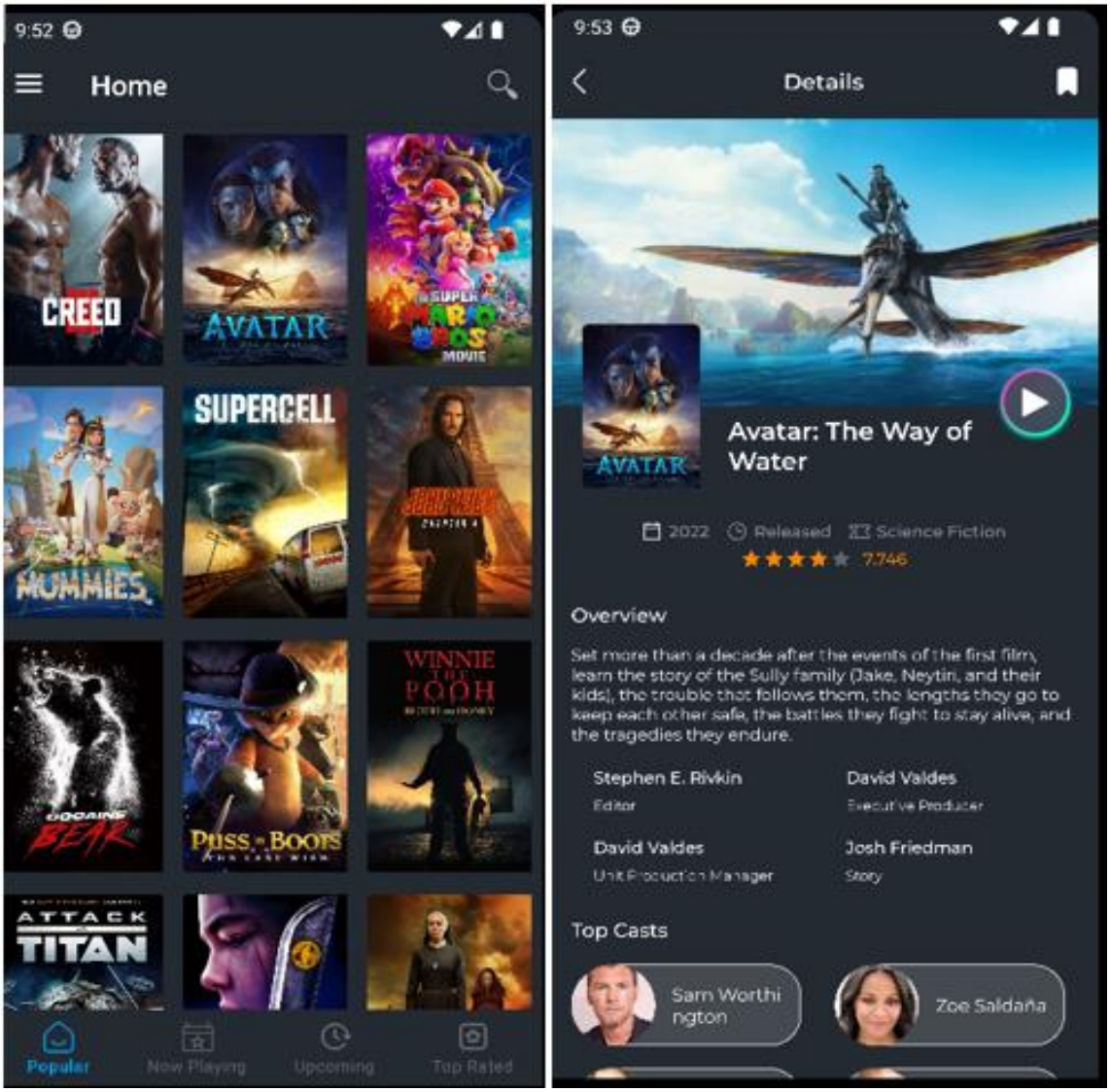
As can be seen in Picture 25, there are two functions: Search and Add Favorite Movie. This could trigger a search query to the backend API that retrieves a list of movies matching the search criteria. In addition to the search function, the app also shows an "add Favorite Movie" function. This feature would allow users to save their favorite movies to a personal account within the Review Movie application.



Picture 25. Searching and favorite feature

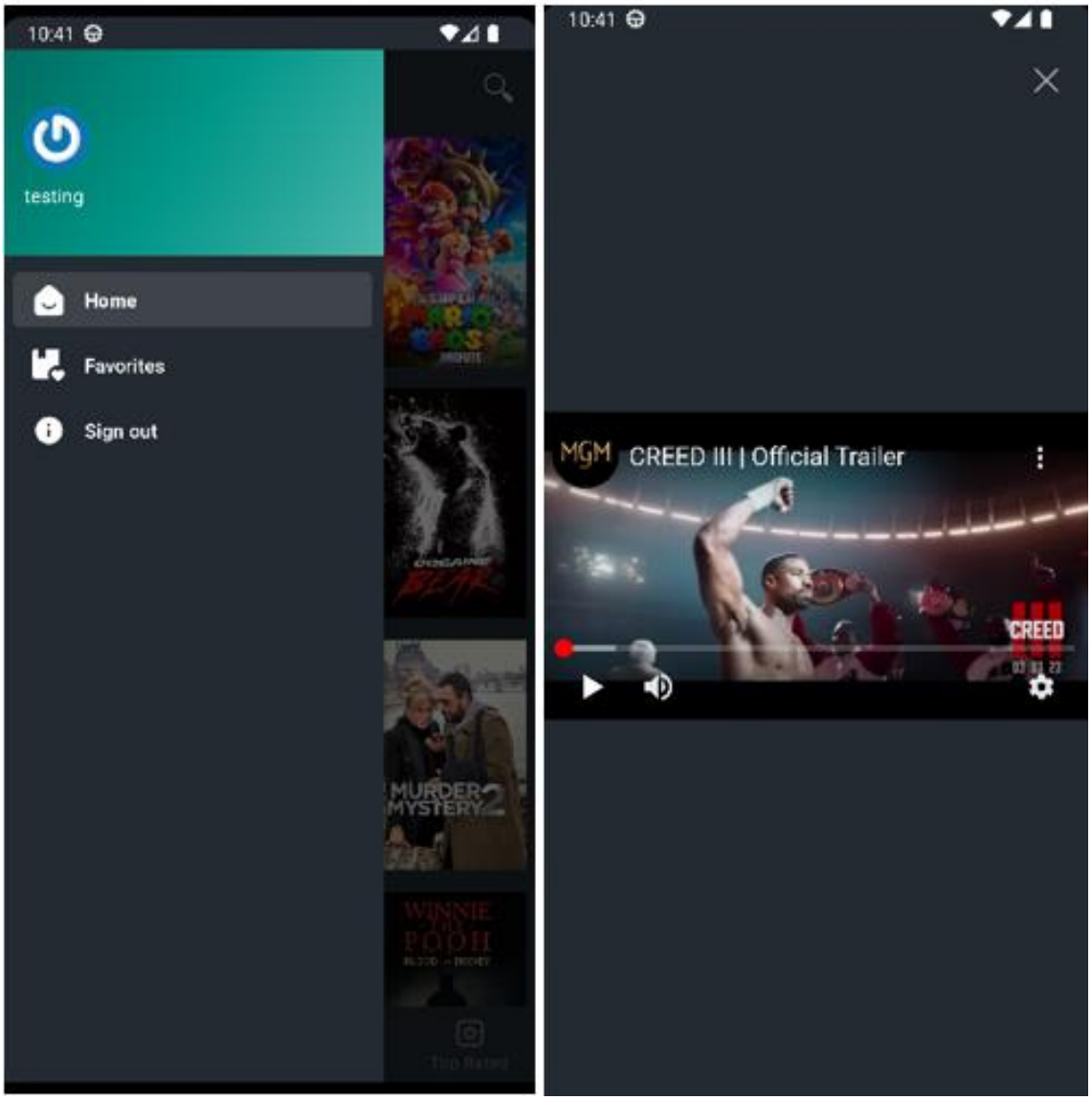
In Picture 26, the app displays a list of movies categorized as popular, now playing, up coming, and top-rated, each tab displays a list of movie titles and posters,

which are likely sourced from an API. When a user selects a movie from the list, additional information about that movie is displayed to the list. This information includes the movie's title, synopsis, rating, release date, and perhaps even user reviews or ratings.



Picture 26. Homepage and movie's details

The last two functions are authentication and video trailers (Picture 27). The authentication function allows users to create an account or log in with their existing account credentials. Once logged in, they can access their personal account's favorite list and add or remove movies as they like. The video trailer's function allows users to watch trailers for each selected movie directly in the app. This feature enhances the user experience by allowing them to preview the movie before deciding to watch it.



Picture 27. Menu navigation and trailer video

5 TESTING AND ANALYSIS

5.1 User experience

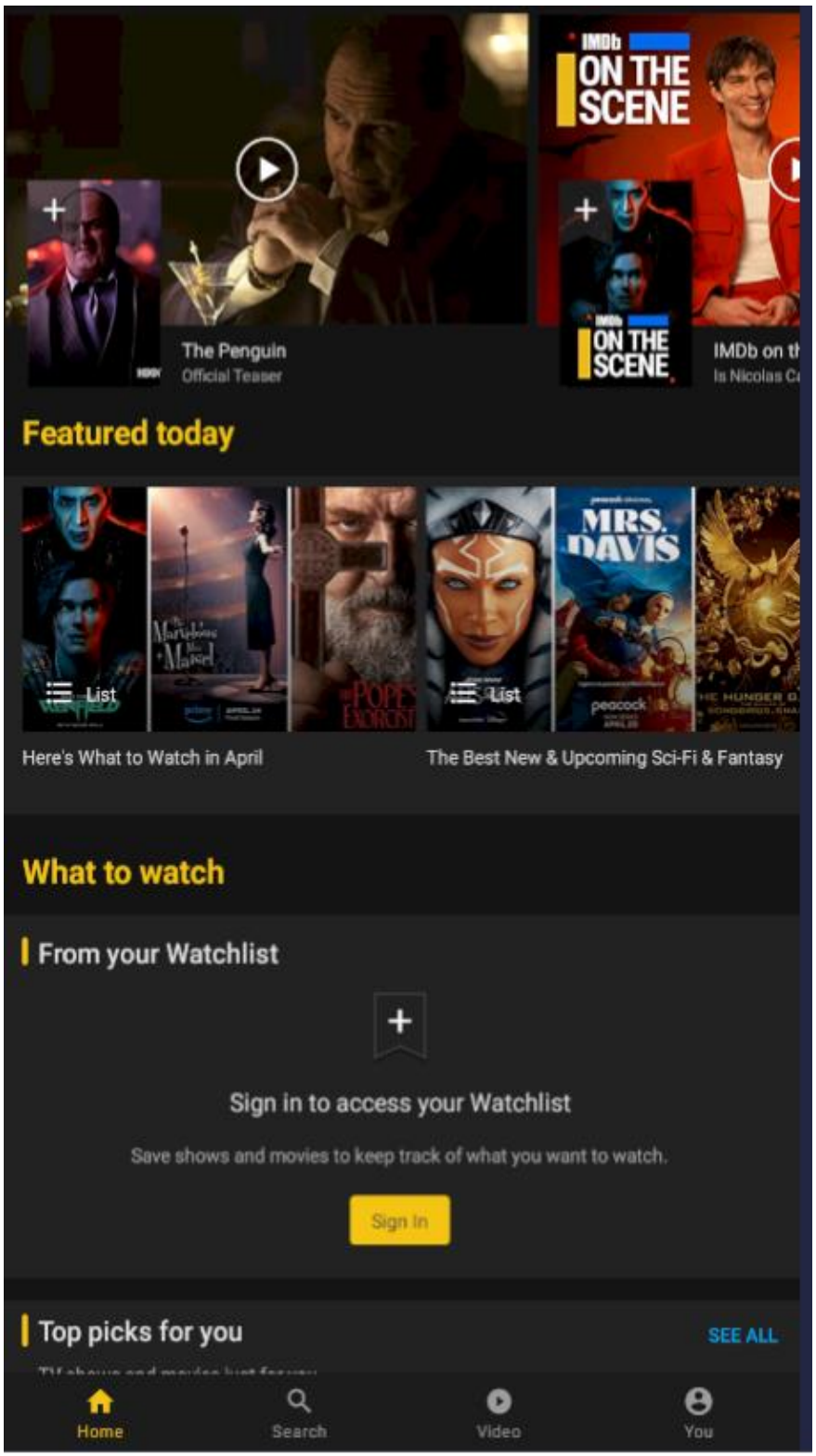
The application was then tested to ensure it worked as required. The testing process was conducted by users to verify the application's features, including logging into the application, saving movies to favorites, viewing related information about movies, and ensuring proper functionality of the video playback function. Most of users confirmed that all functions were fully operational and functioning properly, indicating that the API endpoints worked correctly with no lags when passing information from the frontend to the backend and vice versa.

While the current functionality of the Review Movie application is relatively simplistic, it was designed with a clean and intuitive UI that allows for easy navigation and access to important information. The UI features a minimalist design with a modern look and feel, which helps to create a more immersive and engaging experience for users.

Overall, while the Review Movie application has been successfully tested and offers a solid foundation for further development, there is still ample room for growth and improvement. By continuing to refine and enhance the application's functionality and user experience, developers can ensure that it remains a popular and indispensable tool for movie lovers everywhere.

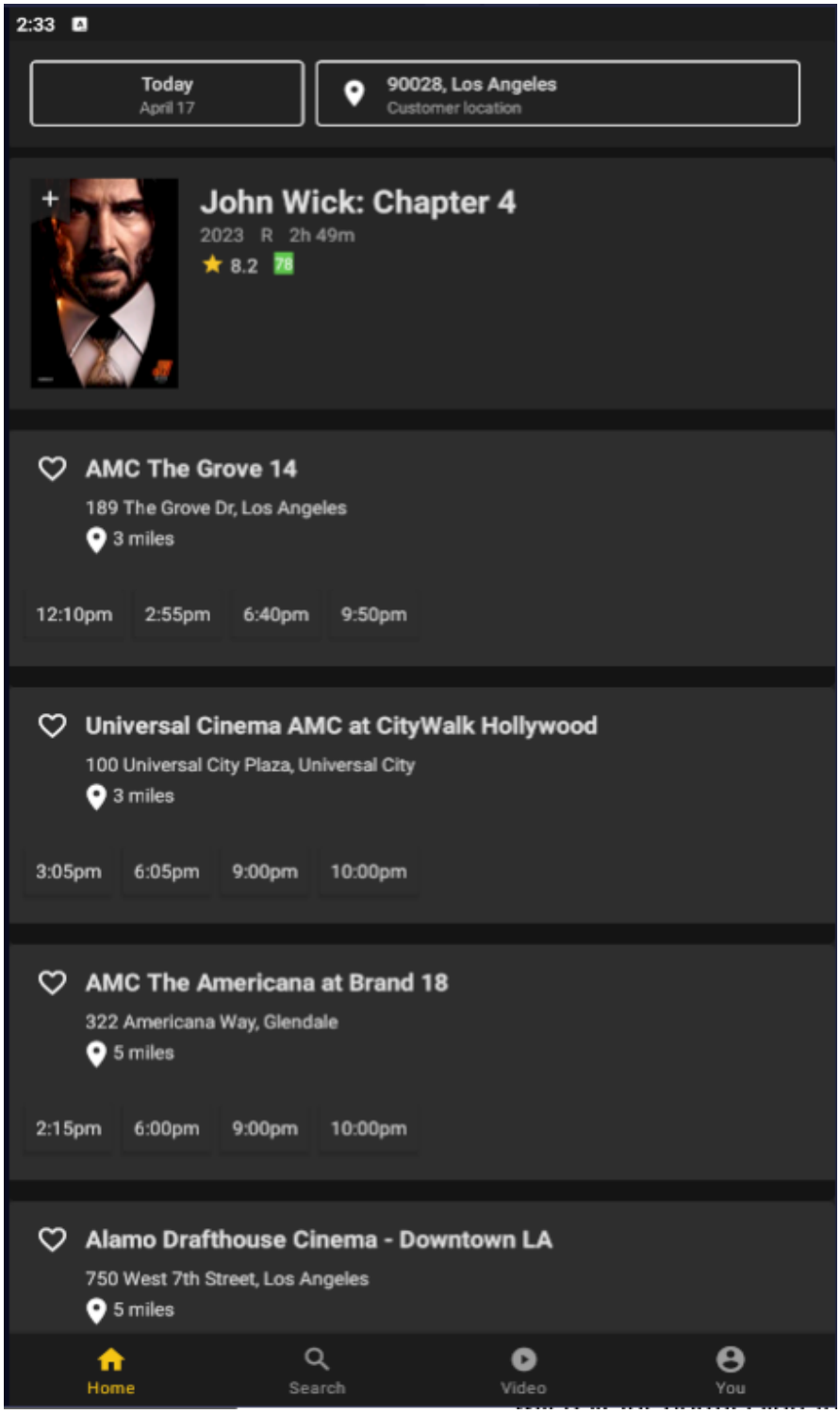
5.2 Comparative analysis

For the purpose of comparison and further development for Review Movie application. I tested another popular movie application, IMDb. IMDb is known as the go-to source for the world's most popular movies, TV, and celebrity information (Picture 28).



Picture 28. IMDb application (IMDb application)

To be honest, the Review Movie application is pretty sketchy compared to applications for movies. After experiencing the IMDb application, Review Movie application needs to be improved on the UI as well as add a few useful features for users. For example, IMDb application has a pretty cool feature: this application will ask for permission to access the user's location, and then the user can buy ticket at the cinemas near the user's location through the ticket purchase feature (Picture 29).



Picture 29. Ticket purchase feature (IMDb application)

Regarding the information of movies, Review Movie application needs to give more details related to every movie, such age restrictions and all the names of all actors, directors, and film makers.

Without a clear understanding of the technologies used by the developers to build the IMDb application, I still firmly believe that it was built using the most optimal and secure technologies available. The application provides users with an exceptional experience, and the seamless performance and reliable functionality suggest that the developers chose the right technologies for the job.

In short, Review Movie still has a lot of potential for development and needs to be improved in many aspects.

6 CONCLUSIONS AND DISCUSSION

As a conclusion, developing the Review Movie application using Retrofit technology and MVVM architecture brings a lot of benefits to developers and users. Thanks to Retrofit, it provides a seamless, lag-free experience, reduces latency, and can efficiently and easily retrieve data from the API. In addition, the MVVM pattern has ensured that the management and maintenance of application are easier. Using ViewModel and LiveData in the MVVM pattern enhances the responsiveness of the application and avoids many risks of data loss.

Through the development of the application and the writing of this thesis, it can be clear that the two technologies used have greatly simplified the management and implementation of coding. Application development can continue, and more features can still be added more easily.

Overall, people's entertainment needs are increasing, and with the development of technology, they can easily meet those needs. Developing applications with the use of advanced technologies and good methods creates a sense of fun for users. That also proves that the market and development potential of entertainment applications are huge now and in the future.

Furthermore, as technology continues to evolve and new advances are made, there is the potential for even more exciting and immersive entertainment applications to be developed. From virtual reality experiences to advanced artificial intelligence, the possibilities for entertainment applications are endless. By staying up to date with the latest trends and technologies, developers can stay at the forefront of this exciting industry and continue to create applications that bring joy and entertainment to users around the world.

REFERENCES

Android Developers, Google LLC. 2021. Android Studio. Read on 31.03.2023. https://developer.android.com/studio/intro

Android Developers, Google LLC. 2021. ViewModel overview. Read on 04.04.2023. https://developer.android.com/topic/libraries/architecture/viewmodel

DigitalOcean. 2018. Retrofit Android Example Tutorial. Read on 01.04.2023. https://www.digitalocean.com/community/tutorials/retrofit-android-example-tutorial

Encyclopædia Britannica. N.d. History of the Motion Picture. Read on 30.03.2023. https://www.britannica.com/summary/history-of-the-motion-picture.

GitHub. 2016. Obut, Orhan. Hawk: HTTP authentication for Android & Java. Read on 07.04.2023. https://github.com/orhanobut/hawk

GeeksforGeeks. 2019. Introduction to Retrofit 2 for Android. Read on 01.04.2023. https://www.geeksforgeeks.org/introduction-retofit-2-android-set-1/.

GeeksforGeeks. 2019. MVVM (Model View ViewModel) Architecture Pattern in Android. Read on 04.04.2023. https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/

Open Textbook Library, University of Minnesota Libraries. 2019. The History of Movies. Read on 30.03.2023. https://open.lib.umn.edu/mediaandculture/chapter/8-2-the-history-of-movies/.

Pierfrancesco Soffritti. 2021. Android YouTube Player API. Read on 06.04.2023. https://pierfrancescosoffritti.github.io/android-youtube-player/

Square, Inc. 2021. Retrofit. Read on 02.04.2023. https://square.github.io/retrofit/

The Movie Database (TMDb) API, TMDb. 2021. Introduction. Read on 31.03.2023. https://developers.themoviedb.org/3/getting-started/introduction

Techtarget. 2019. Model-View-ViewModel (MVVM) pattern. Read on 05.04.2023. https://www.techtarget.com/whatis/definition/Model-View-ViewModel