REPUBLIC OF TURKEY
FIRAT UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND
APPLIED SCIENCES

DEVELOPING CROSS-PLATFORM LIBRARY USING INTEL MULTI-OS ENGINE (MOE)

Dilkhaz Y. Mohammed Haji Mohammed

Master's Thesis
Department: Software Engineering
Supervisor: Prof. Dr. Peter A. Cooper

JUNE-2019

REPUBLIC OF TURKEY
FIRAT UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DEVELOPING CROSS-PLATFORM LIBRARY USING INTEL MULTI-OS ENGINE (MOE)

MASTER'S THESIS

Dilkhaz Y. Mohammed Haji Mohammed

Submission Date to the Institute:
Thesis Presentation Date:

Supervisor : **Prof. Dr. Peter A. COOPER (SHSU)**
Other Jury Members : **Prof. Dr. Asaf VAROL (F.U.)**
Other Jury Members : **Assoc. Prof. Dr. Muhammed Fatih TALU (I.U.)**

JUNE-2019

DECLARATION

I announce that this thesis "Developing Cross-Platform Library using Intel Multi-OS Engine" is set up independent from anyone else an incomplete satisfaction of the necessities for the Master degree of Science in Software engineering.

Dilkhaz Y. Mohammed Haji Mohammed

ELAZIĞ-2019

ACKNOWLEDGEMENTS

I want to express my most profound gratitude for all who had supported and encouraged me during this report preparation as I couldn't accomplish this work without their help and guidance. I am very grateful to my supervisor Prof. Dr. Peter Cooper for providing the chance of carrying out this report under his supervision. I am deeply indebted to him as I couldn't complete the report without his continuous support, advice, and encouragement. I am also profoundly thankful to all the department staff frothier cooperation in resolving I faced during day-to-day work, and thanks to Firat University for giving me this opportunity to achieve my goal in life and to all teachers in Firat University, I would like to express my respect for my wife is always she beside me while doing any tasks in my life.

Sincerely

Dilkhaz Y. Mohammed Haji Mohammed

TABLE OF CONTENTS

ABSTRACT

Developing Cross-Platform Library Using Intel MULTI-Os Engine

Libraries are of great importance in the development of mobile applications. Mobile application development services have reached a higher level with APIs. When developers develop applications for mobile devices, they often rely on APIs for connectivity. In fact, APIs accelerate mobile development and provide exceptional agility for organizations undergoing their own digital transformation. As a result of developing a library and then sharing it with the world, others can benefit from it in their own projects. The programmer needs to make sure that he uses APIs available for both Android and iOS platforms. The programmer's creation of an interface for accessing platform-specific functions from the library and creating Android and iOS applications in its projects accelerates the development of software projects.

The purpose of this thesis is to create a taxi service library for developers using both Android and iOS using Java Object Oriented Programming, Bintray, Maven, Gradle and JCenter utilities were also used in this project.

Keywords: Cross-platform, Library, Uber web service , Retrofit, Gson

ÖZET

Intel MULTI-OS Motorları Kullanarak Çapraz Platformu Kütüphanesi Geliştirme

Mobil uygulamaların geliştirilmesinde kütüphanelerin büyük önemi vardır. Mobil uygulama geliştirme hizmetleri, API'ler ile daha yüksek bir seviyeye ulaşmıştır. Yazılım geliştiriciler, mobil cihazlar için uygulamalar geliştirdiğinde, bağlantı için çoğunlukla API'lere güveniyorlar. Aslında, API'ler mobil gelişimini hızlandırır ve kendi dijital dönüşümünden geçen kuruluşlar için olağanüstü çevikliği sağlar. Bir kütüphane geliştirmek ve daha sonra bunu dünyayla paylaşma sonucunda, başkaları da kendi projelerinde bu kütüphaneden faydalanabilir. Programcının hem Android hem de iOS platformları için kullanılabilen API'leri kullandığından emin olması gerekir. Programcının kütüphaneden platforma özgü işlevlere erişmek için bir arayüz oluşturması ve bu arayüzün projelerinde Android ve iOS uygulamaları oluşturması, yazılım projelerinin gelişimini hızlandırır. Bu tez çalışmasının amacı, Java Object Oriented Program kullanarak geliştiriciler için hem Android hem de iOS destekli bir taksi servis kütüphanesi oluşturmaktır. Bu projede Java, Bintray, Maven, Gradle ve JCenter yardımcı araçları da kullanılmıştır.

Anahtar Kelimeler: Cross-platform, Library, Uber web service, Retrofit, Gson

LIST OF FIGURES

Page No

LIST OF TABLES

Page No

SYMBOLS AND ABBREVIATIONS

API : Application Programming Interface

CRUD : Create Read Update and Delete

HTTP : Hypertext Transfer Protocol

IDE : Integrate Development Environment

JAR : Java Archive

JDK : Java Development Kit

JRE : Java Runtime Environment

JSON : JavaScript object Notation

JVM : Java Virtual Machine

POJO : Plain Old Java Object

SDK : Software Development Kit

UI : User Interface

URI : Uniform Resource Identifier

URL : Uniform Resource Locator

UX : User Experience

1. INTRODUCTION

1.1. Overview

Mobile applications are created to perform basic functions, and the basic technologies that underpin mobile application development have significantly changed. Mobile devices have it has affected the ways in which business objectives are achieved. Moreover, organizational changes in access and productivity have led to new ways for users to deal with internet businesses, interactivity, and data consumption.

Mobile application development services have evolved to take advantage of into a higher-level API. The concept of Application Programming Interface (APIs) is referred to as a set of protocols, subroutines, and tools for building applications. Application Programming Interface (APIs) are increasingly becoming important to mobile application development as they dictate the way application developers develop their applications and integrate popular web services into them without having to recode, and getting the benefit of tested and proven routines.

Developing a library and sharing it with the world later on so that others can make use of it in their projects. The programmer just needs to make sure only use APIs that are available on both for Android and for iOS. The programmer must create an interface to access platform-specific functionality from the library and create Android and iOS implementations of that interface in their projects, thereby speeding development.

In this project, the programmer plans to use Android studio as the IDE of choice, supplemented by Gradle. The programmer will build a taxi service library which other developers might use in their Android or iOS Apps. The library will be initially managed through Bintray which allows for code modification until testing has been completed and then published through Maven.

In Android Studio, the programmer can add a library to our application for the various process, some of the most processes are:

1. Add a library's jar file in the libs folder and then add it as a dependency.
2. Add the library as a module in the project.
3. Add the library as a dependency in-app (build. Gradle) file.

1.2. Research Objectives

This research is mainly aimed at building a taxi service library which other developers might use in their Android or iOS Apps (provide a way to share code across cross-platforms). libraries are a great way to create modular code that can be easily shared. However, this objective is sub-divided to the following objectives:

1. Using uber RESTFUL web service (ride sharing).
2. Using Retrofit Library to handle RESTFUL web services. This powerful library allows programmer to define REST API as an interface.
3. Distributed the library using JCenter - This helps developers using the programmer library since IDEs support downloading those JARs and displaying the sources directly in the editor.
4. Developing a Native Android App using the library.

1.3. Research Methodology

Using the retrofit approach to handle RESTFUL web services. This powerful library makes it relatively easy to retrieve and upload JSON via a REST-based web service. For making HTTP requests retrofit uses the OKHTTP library. With retrofit, the programmer doesn't worry about parsing the response- meaning de-serialization is handled in the background itself. The programmer just needs to configure Gson converter library and the job is done.

Writing library as a Java library, and package it as a jar. Then the programmer can distribute the library using JCenter, the developers can add a Maven dependency on their (build. gradle) file.

1.4. Definition of Terms

Defining the used terms is necessary for delivering the desired meaning and enabling the reader to conceive the ideas relevant to this study. Thus, the following terms are put forward with their relevant definitions:

Software Development Kit (SDK): A set of software collections dedicated to developing apps for a specific device or operating system.

Gradle: Gradle is a build tool which evolves itself independently on Android. Theoretically, Gradle can be used to build any type of project, for example, a typical Java project, Android project, and even iOS project.

Repository: Repository is a storage or a directory used for saving all library and project jars, plugins or any other project specific artifacts for the easy and convenient use by Maven.

JCenter: JCenter is one of the Maven repositories or it is a file hosted by Bintray.com for android libraries. It's a default repository for Android Studio.

Maven Central: Maven Central is the Maven's default repository as well as many other tools built using JVM.

Bintray: Bintray is a cloud platform (Cloud service) that gives the programmer full control over how you publish, store, and distribute software.

Maven: Maven is a tool developed by apache which is used to hold build artifacts and dependencies. There are few standard servers such as JCenter, Maven Central, etc. used to host the libraries for Android.

Artifacts: The term "artifact" covers all of the material collected or created during the life of the project.

Library: A library is a set of functions/modules/APIs which the programmer can use to solve a certain problem, but it doesn't change your code on a structural or architectural level.

Framework: Framework is a set of functions/modules/APIs but it does change your code on a structural or architectural level. (to help with building an application).

API: An API is the functions/methods (the front-end code) in a library that the programmer can call to ask it to do things for you- the interface to the library.

Native app development: Native apps are apps built by platform-specific programming languages. For Android apps, for instance, Java, Kotlin, and Python are some of the possible development languages. Whereas in the case of the iOS mobile operating system, the Objective-C and Swift are an example of possible development languages. When apps are developed specifically to run by the support of all native hardware components, such as camera and calendar, and technologies on the target platform, they are referred to by native apps.

Cross-platform app development: Cross-platform is a developed application which can suitably run on various different operating systems. Once the app's code has been written, it can be installed on a variety of devices and it can run on a number of platforms without any

adaptability or compatibility concerns. The approach here is widely used for saving both time and money by deploying an all-in-one development fashion.

JSON: JSON stands for (JavaScript Object Notation), which is a generic format widely deployed for sending and requesting data through a REST API. In many applications, such as Uber, the response sent back to us is also in JSON format. In JSON, double quotation marks must wrap all properties and values.

MOE: Multi-OS Engine Technology Preview, proposed by MIGERAN and developed by Intel, MOE is an open source framework that enables the programmer to develop native mobile apps for iOS with only Java expertise Apple macOS development machines, without compromising the native look-and-feel performance. The programmer can reuse as much common Java code as possible and add platform-specific UI code for each platform.

POJO: POJO stands for Plain Old Java Object, which is a Java object that doesn't necessarily deploy an interface implementation or a base class derivation. In addition, for POJO objects framework compatibility, there is no specific need for any annotations; rather, they can be arbitrarily designed as Java objects, which are generally kept simple.

Gson: Gson also known as Google Gson is a library that maintains a powerful facility for the conversion process. The conversion here is done between JSON strings and Java objects. Thus, avoiding the need for a self-designed or written code for JSON parsing purpose.

Retrofit: Retrofit is a REST client developed by Square and can be used for both Android and Java. It is mainly used as a powerful framework for APIs authentication and interaction, and also for sending OkHttp library's network requests. Retrofit supports API calls that need authentication, which is handled by one of the following two methods: deploying the annotations to manipulate the header and produce the request or the use of an OkHttp interceptor for achieving the same purpose. The download of JSON data from any web API is made convenient with Retrofit library. After downloading the data, the parsing process takes place to produce the Plain Old Java Object (POJO). These objects have to be defined for each "resource" in the response.

Bearer token: Bearer token is a security token with the property that any party in possession of it can. There is no specific request for possessing a cryptographic key material in order to use a bearer token. Bearer tokens are part of the OAuth2.0 standard and widely adopted by many APIs.

RESTFUL API: RESTFUL API is also known as a RESTFUL web service and it is an application programming interface (API) that uses specific HTTP requests for data manipulation, such as DELETE, GET, POST and PUT data. This API is developed relative to representational state transfer (REST) technology, which represents a communication architectural pattern and approach often deployed in web services development.

1.5. Thesis Organization

This thesis is systematic as follows:

- **Chapter I:** The general introductory section which presented the needed overview about how to developing a cross-platform library which will be work on Android and iOS. It also provided the objectives of the research and methodology which is followed in conducting the experimental work.
- **Chapter II:** this chapter which presented a review of the literature about how other developers notices to developing a cross-platform library as well as a background of mobile App.
- **Chapter III:** This chapter presents and discusses experimental results obtained from building a taxi service library using Java OOP, retrofit with the help of the Gson converter, and utilize the library for developing Android App in our study. This chapter also provides a way of publishing open source release with Gradle.
- **Chapter IV:** This is the final chapter which it is dedicated to concluding final remarks on the work of this thesis as well as providing the final insight on possible ways for further enhancing the work in future studies.

2. LITERATURE REVIEW

The trend of mobile app development has become very common among different programming communities. The reflection of such implication could be due to both time and cost, where the development of an app requires a relatively low cost as well as a very short time for realizing an idea in the form of a complete app compared to traditional computer programs. In addition, nowadays, the availability of plenty of useful learning, training and empowering resource suitable for apps development has enabled novice developers to join such development communities.

Over time, several development toolsets have been released to simplify cross-platform development. One of the popular toolsets is called Xamarin platform which enables developers with skills in C# to create mobile applications for Android, iOS, or Windows mobile on their Mac or Windows PC environment. Developers who are the most comfortable with Java were looking forward to using Robo VM to code for iOS until Intel working on a project called the Multi-OS-Engine Framework available today as a Technical Preview, that is aimed at enabling Java developers an easy way of cross-developing for both Android and iOS [1].

Many approaches to developing a library which will be work on Android and iOS. Go-Mobile approach provides a tool, go-bind generates target language (Java/Objective-c/Swift) bindings for each exported symbol in a Go package but has some limitations. There are some solutions to this, but there are some quite cumbersome. Like only a subset of Go types is supported. Also, as Go manages its memory in its own garbage-collected environment, passing pointers to another language must be cautiously done to prevent the destruction of objects that are still used in the other environment. Google J2ObjC approach a Java to Objective-C translator with still a few holes in the implementation [8].

Another option is code porting, which was considered to provide a warehouse for codes or codebase, then deploy suitable and needed tools to convert these codes into the desired platforms. However, this option does not provide such a possibility without any drawback. For instance, the codes written by developers are proven to be more efficient than the generated codes, debugging has to manually done as bugs are likely to be introduced, and imported binaries are unlikely to be translated as most such tools translate only source

code. Some of the code port tools like (J2ObjC, ObjC2J, Hyperloop, and XMLVM). However, none of them provided the required functionality [5].

Another option is developing the library in C++. In Android, the Native Development Kit (NDK) and the Java Native Interface (JNI) framework allow running/ interfacing C/C++ code from Java. In iOS, no naming conventions are available, and just one level of wrappers is required, using Objective-C. But the only drawbacks to this approach are that C++ does not have a garbage collector and you must manually handle it [5].

However, building a jar file and distributing that may be the easiest as other developers will just have to include the jar Apps. However, that requires a distribution mechanism, resulting in the potential for multiple competing jar files with different functionality. If the programmer talking about anyone accessing the code, a web service may make more sense as the programmer can provide access to the data by providing all of the necessary code. The programmer can also consider building jar file using Apache Maven and put the jar file in a public maven repository, public a new version of the library as the programmer finds the bug. Bug fixes are handled by updating the .jar file and proprietary code can be obfuscated to protect intellectual property.

There are many repositories that enable Java/Maven ecosystem to publicly publish JARs. For example, Maven Central (the repository deployed as default for Maven and many other JVM based build tools), and JCenter (a more comprehensive realization or a superset of Maven Central, which is served by Bintray, and besides including everything in Maven Central, it contains projects that are published directly to JCenter). This repository is set as the default repository in Android Studio) [15].

2.1. Observations

The two top mobile operating platforms that have been controlling the market in the past few years are undoubtedly Apple's iOS and Google's Android mobile, yet these two platforms are very dissimilar in their orientation.

Android

Android is a Linux-based apache-free operating system suitable for mobile or tablet devices. It is developed by Google to promote advances in the mobile devices industry, and its initiative was launched in 2007. The general architecture of Android, shown in Figure

2.1, is composed of four separate layers: the bottom-most is dedicated to the Linux kernel, next to it, various required Libraries, which form a base for the Application Framework, and top-most is the Applications layer [6].

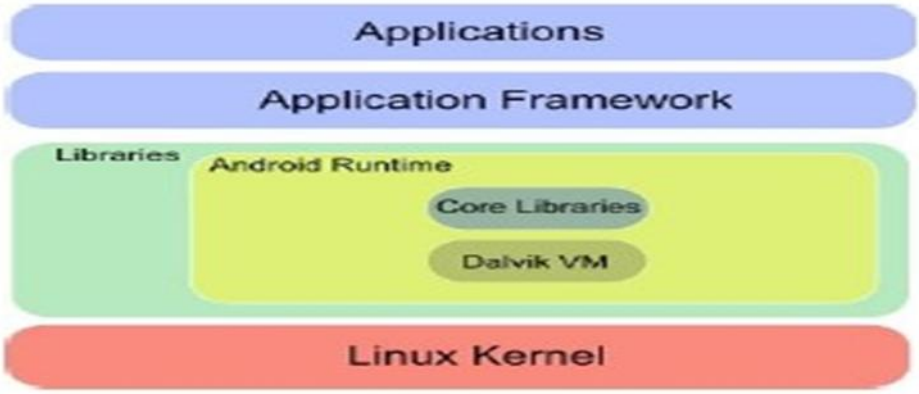


Figure 2.1. Android Architecture

iOS

iOS is the default operating system for Apple mobile and tablet devices. It was officially released in 2007. The developed of iOS is made using XCode, which is an IDE included in Apple's SDK. As shown in Figure 2.2, Like Android architecture, iOS architecture is composed of four distinct layers: the bottom-most is represented by the Core OS layer, which is the base of the Core Services layer, next to them the Media layer, and the top-most is dedicated to the Cocoa Touch layer [6].

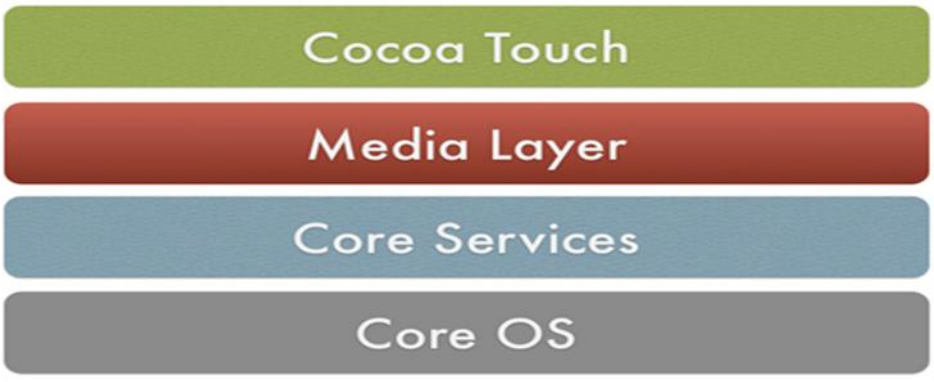


Figure 2.2. iOS Architecture

Although apps sound very similar on both operating systems, i.e., Android and iOS, the development of such apps, in reality, is largely an OS dependent where both operating systems have their distinct features. In fact, many differences can be encountered when

dealing with both Android and iOS, which include both the technical background of the development process and the mechanism of building-up the app itself.

It is important to mention that the concept of app development largely depends on the selected operating system as not only the technical level that matters but also the design and mobile strategy. This becomes very noticeable when a developer switches the development from one operating system to another. The differences here will cover the used programming language, apps testing tools and beds, deployed graphical interfaces. In addition, the following dissimilarities can be recognized in apps development:

• Delegate vs. Adapter: a delegate is deployed by iOS if delegation patterns are used. However, if the same patterns are used in Android, they will be represented by an adapter. Essentially, the concepts are very similar despite their different platform deployment and linguistic presentation.

• UIViewController vs. Activity: in mobiles devices running on Android, the Activity class is used for representing particular screenshots; while, to achieve the same task in iOS operated devices, UIViewController, a specific controller, is used. Both of these components can be deployed for managing events lifecycle, sub-views, and relevant tasks. Thus, they do attain the same duties even though they are not the same components.

• Unlock: unlocking Android devices is achieved by swiping up while unlocking iOS devices is made by swiping to the right.

• Preferences and access permissions: these features are properly organized in one category under the general preferences section in iOS devices. However, this is not the case in Android devices where they are distributed over different sections and specific settings can be found via some searching process through the available menus.

• Maps: both Apple maps and Google Maps can be used in iOS apps' development. Yet, it is usually common to restrict the use of Google Maps for the development of Android apps. Moreover, both beacons and geofencing have been widely used as a popular combination with these maps.

The development of cross-platform apps offers the opportunity to "develop once, deploy everywhere". Cross-platform apps are developed applications that are adaptable with different operating systems. Once the app's code has been written, it can be installed in a variety of devices and it can run on a number of platforms without any adaptability or compatibility concerns. The approach here is widely used for saving both time and money by deploying an all-in-one development fashion [11].

Without a doubt, there are pros and cons of developing mobile apps to run on an individual target platform. Similarly, the development of cross-platform mobile apps has its own advantages and disadvantages.

Some of the native approaches in mobile apps development's advantages and disadvantages are listed in Tables 2.1 and 2.2 respectively.

Table 2.1. Advantages and disadvantages of using the native approach in mobile app development.

Pros of native apps	Cons of native apps
High performance	Costly and time consuming
Better user interface	Missed opportunities
Better positioning on app stores	

Table 2.2. Advantages and disadvantages of using the cross-platform approach in mobile app development.

Pros of cross-platform apps	Cons of cross-platform apps
Affordable and time-saver	Performance glitches
Easy and fast deployment	User experience issues
Wider audience reach	

3. METHODOLOGY

3.1. Overview

These days almost every mobile app connects to the internet to get and send data. The programmer should definitely learn how to handle RESTFUL web services, as their correct implementation is the core knowledge while creating modern apps. Retrofit one of the most popular libraries every Java programmer should know. This powerful library allows the programmer to define REST API as an interface. Libraries are a great way to create modular code that can be easily shared. Libraries are created and distributed as packages. In Java applications, once such a library has been created, managing and deploying it is very convenient. However, to share code across platforms using Java, to write platform-independent code and share it between Android and iOS with a Multi-OS Engine framework.

The followings are the key features of Multi-OS Engine:

- Supporting Java on iOS devices that lets programmer to run bytecode in iOS
- Providing a way to share code between Android and iOS
- Provides a custom bridge called Nat/J, which is bridges native iOS code and Java; with proper plugins in place, Java bindings for iOS native views can be generated using it.

To create a Multi-OS Engine, the following essential steps can be followed as a workflow:

- Initially an Android app module should be created.
- Once the Android app module is created, a Multi-OS Engine app module should be composed. This forms in reality the iOS app.
- A module should be created and it should be common to hold the shared app logic for both Android and iOS parts.
- Once the common module is created, it should be added as a dependency for the Android and iOS app modules.
- Gradle scripts have to be configured.
- The final step is building and launching the developed app.

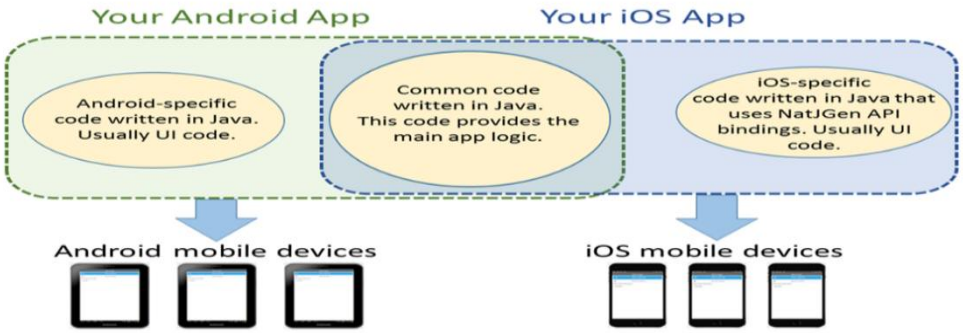


Figure 3.1. Multi-OS Engine Overall workflow

3.2. Project Requirements

- ✓ A computer running on Windows 10 OS x64 edition or MacOS.
- ✓ A minimum of 4 GB RAM (8 GB or higher is recommended):
- ✓ Version 8 of Oracle Java Development Kit (JDK):

JDK comprises useful tools required for both development and testing of Java written codes which run on a Java platform.

- ✓ Google Android Studio 3.2.1 IDE:

Built on JetBrains' IntelliJ IDEA software and designed specifically for Android development, Android Studio is the official integrated development environment for Google's Android operating system. This IDE has customized versions suitable for operating on Linux-based, macOS, and Windows operating systems.

- ✓ MacOS high sierra 10.13.6 + XCode 9:

Creating UIs, the programmer can use XCode's interface builder for designing and Multi-OS Engine for generating the Java bindings in Android Studio. Create the skeleton of the view controllers in Objective-C, and generate Java bindings for them. For successful UI binding generation, all necessary files (. storyboard, .h, etc.) should be placed in the XCode project directory.

✓ Multi-OS Engine plugin:

For Android Studio, this plugin can be installed as follows: Settings → Plugins → Browse repositories → Search on Multi-OS Engine Plugin → add

✓ Postman Native App for win64-6.6.1:

It is a rest client for test rest API. The programmer use curl because API documentation examples are normally written with reference to curling (is the general terminology of how to implement the request). Postman currently one of the most popular tools used in API testing.

3.3. Organizational Chart

There are three main parts of this work, to begin with building a cross-platform library (common module) using retrofit to handle RESTFUL web service, following that publish the library into the internet using specific tool that the other developers might use in their Android or iOS apps and last part of work is develop Android app using the library. Developing an iOS version will be in future work. Figure 3.2. work organizational chart.

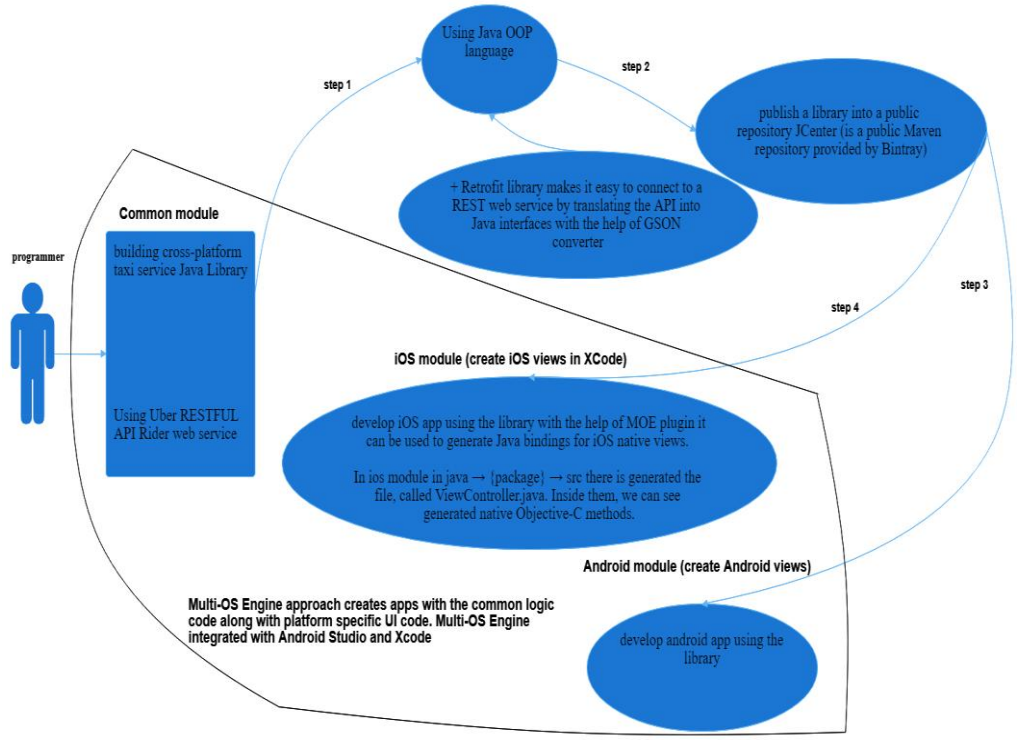


Figure 3.2. Organizational Chart

3.4. Application and Implementation Charts

3.4.1. Application Chart

In Figure 3.3. the programmer presents a simplified diagram of the state paths of an activities Android app. To request a ride for a rider or to a rider's trip history, the rider needs to authenticate with uber and authorize the app (have the rider sign in and authorize to access their uber account) and after authorization is completed. User can make requests to the uber web service.

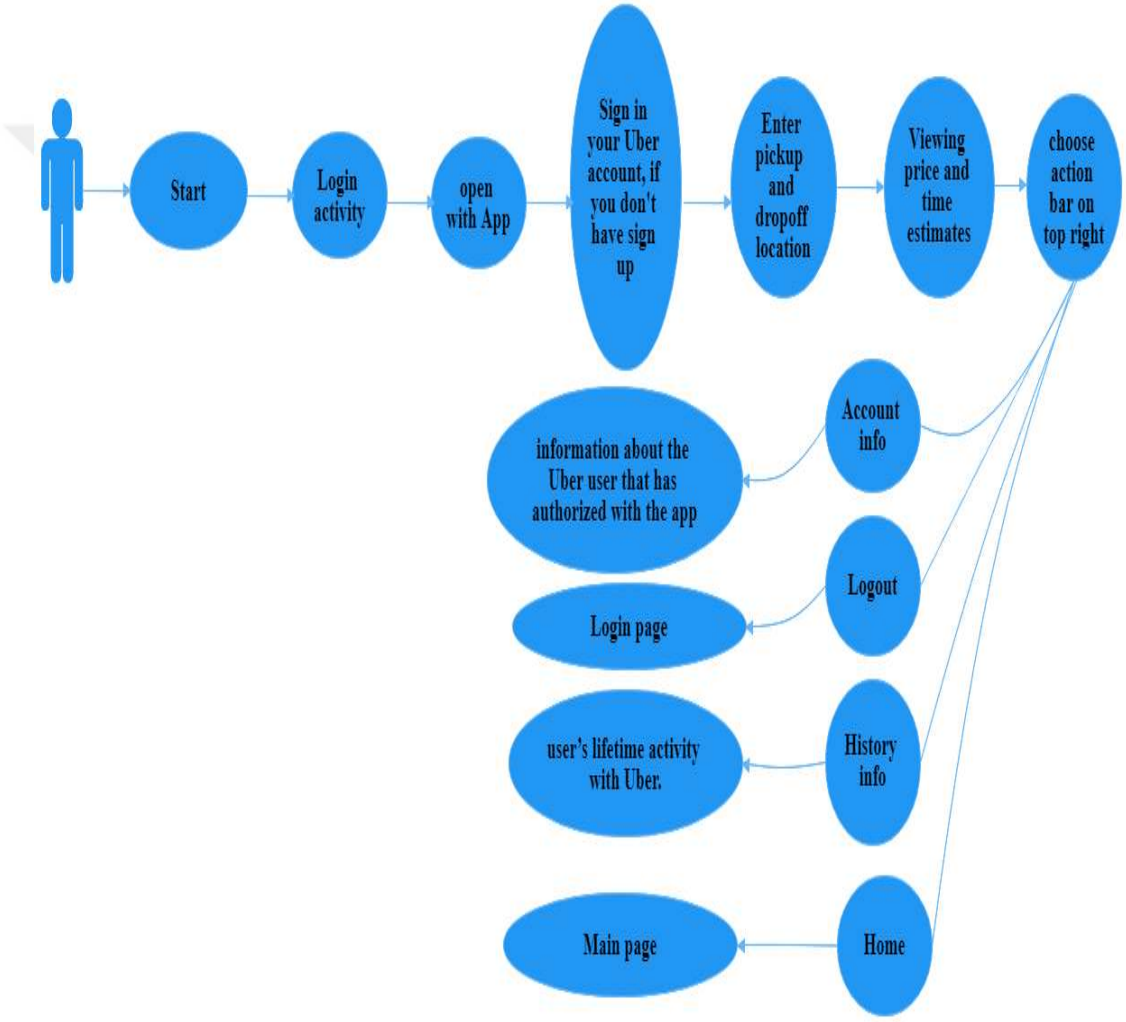


Figure 3.3. Taxi Service Diagram

3.4.2. Implementation Chart

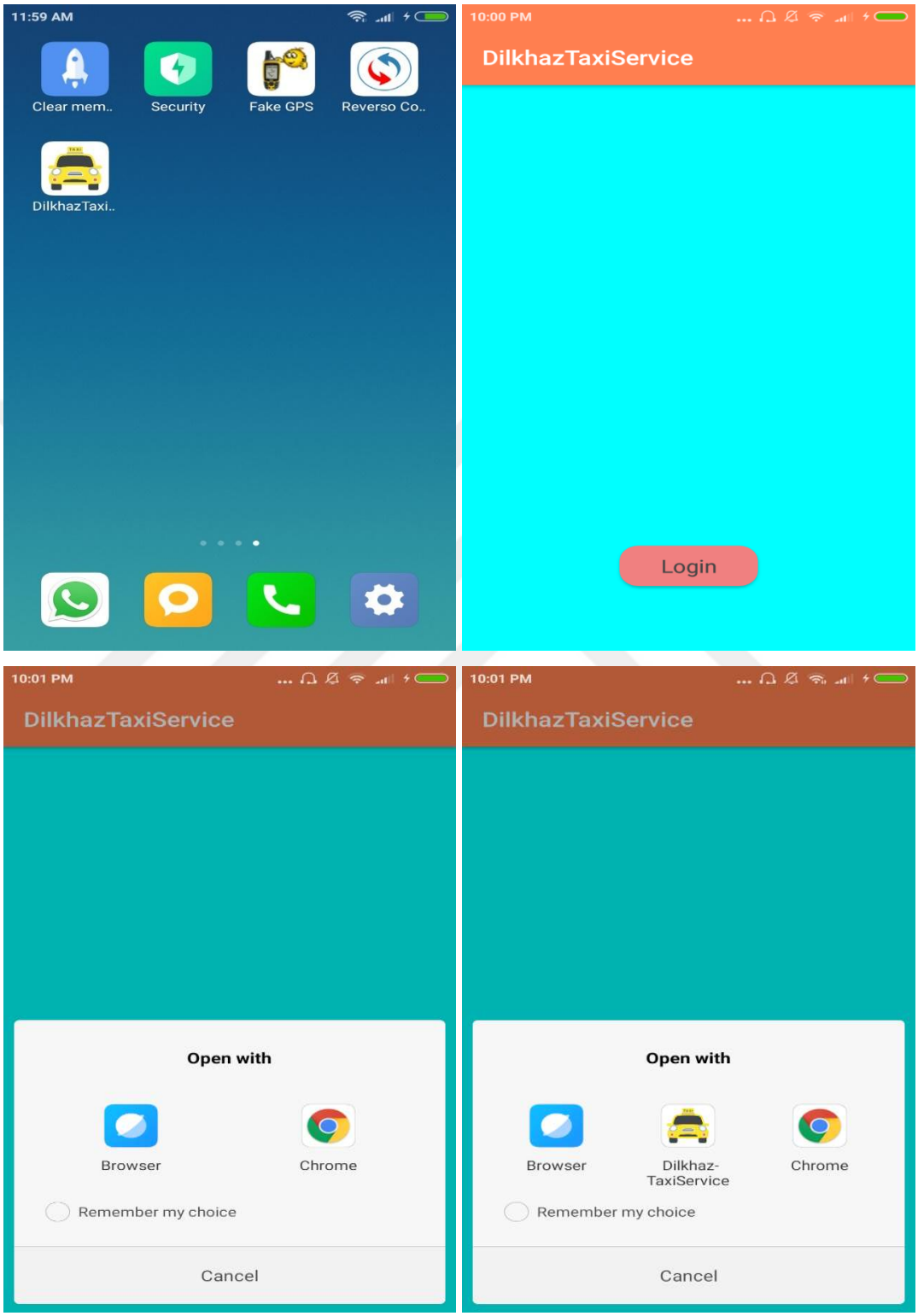


Figure. 3.4. Implementation Chart

Figure. 3.4. Implementation Chart Cont.

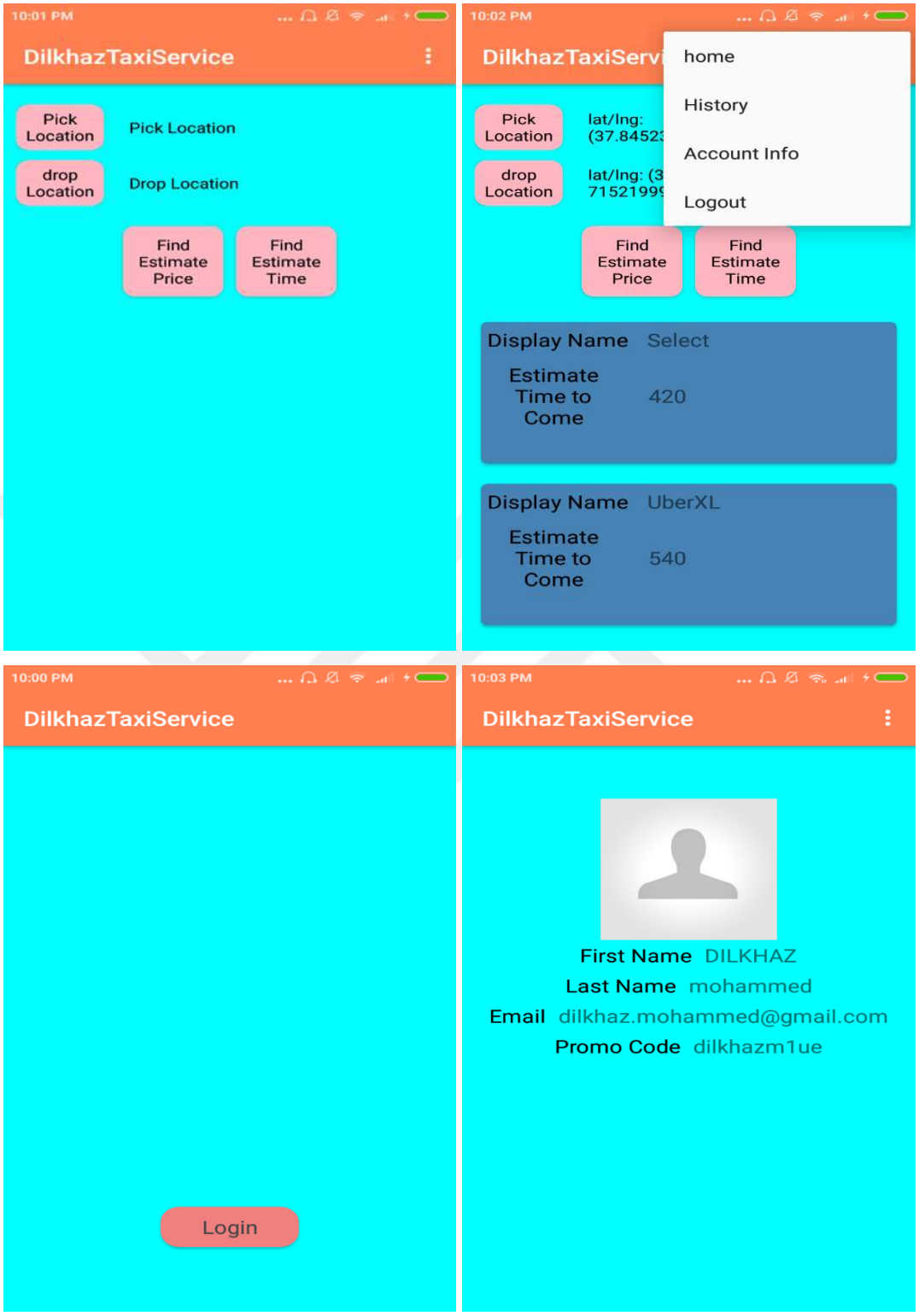


Figure. 3.4. Implementation Chart Cont.

3.5. Implementation Steps

3.5.1. Building a Taxi Service Library using Java

Uber Riders API References are:

1- All production API requests are made to https://api.uber.com/ v1.2/

2- The Uber API is a RESTful API. This means that the API is designed to allow you to get, create, update, & delete objects with the HTTP verbs GET, POST, PUT, PATCH, & DELETE.

3- The Uber API speaks exclusively in JSON. This means that you should always set the Content-Type header to application/json to ensure that your requests are properly accepted and processed by the API.

4- Endpoints calls require authentication using bearer. Bearer tokens allow programmer application to access the Uber API on behalf of a user and are obtained after a user has authorized programmer application through one of the supported OAuth 2.0 authorization grants.

RESTful service mainly focuses on resources themselves as well as providing access to these resources. Here, any resource can be simply visualized by an object, similar to the practiced concept in OOP. In addition, it is possible for a resource to include other resources. Thus, in relevant systems' design, resources have to be initially identified and their relations have to be determined. This resembles the initial step in designing a database: Identify entities and relations.

Using one of the most popular and often-recommended HTTP libraries available known as Retrofit. Retrofit is a REST client developed by Square as a type-safe, and it can be used for Android, Java, and Kotlin. It is mainly used as a powerful framework for APIs authentication and interaction, and for sending OkHttp library's network requests. To work with Retrofit, the programmer essentially needs the three classes listed below:

• Model class, which is used as a JSON model

• Interfaces that define the possible HTTP operations

• Retrofit. Builder class – Instance that uses the interface and the Builder API to allow defining the URL endpoint for the HTTP operations

Every possible API call is represented by an interface method represents one possible API call. To specify the request type and the relative URL, the request must contain an HTTP

annotation such as GET, POST, etc. The expected results determine the return value type, which gets the response in a Call object wrapped to maintain this type.

Programmers initiate their requests from APIs where the starting point here is the root-endpoint. In this research's case, Uber's root-endpoint API is https://api.uber.com/v1.2. The endpoint's terminal is represented by query parameters. These, enable the programmer to modify the request with key-value pairs.

How does the programmer know if this endpoint works? Well, a request can be sent by the programmer realized by any programming language. The programmer will use the command line utility called curl and applied to Postman Native App is a REST Client for Test REST API. The programmer uses curl because API documentation is normally written with reference to curling. If the programmer understands how to use it, he will face no difficulties comprehending API documentation. Then, the programmer can conveniently initiate requested with his preferred language.

Accessing an API, for instance, dictates that the programmer should use an Access Token. An access token is a Bearer token that the programmer will have to add in all request headers to be authenticated as a concrete user. The Bearer Token is created for the programmer by the Authentication server. When a user authenticates your application (client) the authentication server then goes and generates for the programmer a Bearer Token which the programmer can then use to get an Access Token.

Before the programmer dive into the code, the programmer first needs to register an application on Uber Developer Dashboard as shown in Figure 3.5. This is where the programmer will find the information about his app. The dashboard consists of several tabs, but the programmer will only need to use the Auth tab. The Auth tab is where the programmer can find his app's Client ID, Client Secret, configure the URLs app will use for authentication.

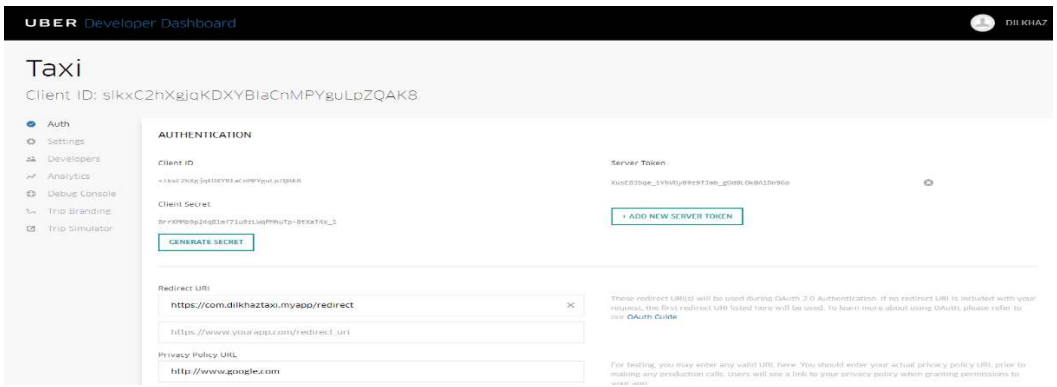


Figure 3.5. Uber Developer Dashboard

Multi-OS Engine (MOE) apps should have three modules. The views of each platform should be separately placed in two of these modules, and the third module is used for including the shared views of the platforms. Both views of Android and iOS do not have direct access to each other. Essentially, these views should include only methods that are in control of UI, and minimizing these methods is highly recommended. To write clean code and produce a separate view, it is recommended to us Model-View-Presenter architectures.

The first part of the Multi-OS Engine App is to create modules and set dependencies

Android module

The first module to be created is the app, which is regarded as a regular Android application. To create this module, a project has to be created from the Android studio as follows:

Android Studio > create project > Select Empty Activity

Common module (Library)

This is the second module, which includes all application's logic and it is shared between Android and iOS modules. This module is created as follows:

From project view > New > Module > Java Module and name it "commons"

iOS module

This is the third and last module, which includes all iOS codes written in Java and the entire XCode project. This module is created as follows:

From project view > New > Multi-OS Engine Module > Single View Application

The programmer is ready to start writing the uber web service rider Library

1. Add Retrofit, Gson, OkHTTP into dependencies (build.gradle file).

   Add dependencies for Retrofit 2 in common module in build.gradle file

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    // Retrofit
    implementation 'com.squareup.retrofit2:retrofit:2.5.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.1.0'
    implementation 'com.squareup.okhttp3:logging-interceptor:3.4.1'
}
```

   - Retrofit: Type-safe HTTP client for Android
   - GSON: A library to convert Java Objects into JSON and back
   - The programmer use OkHTTP for logging. An OkHttp interceptor which logs HTTP request and response data.

2. Create four sub packages named Models (Create Java Classes for Resources), RestCall (Setting Up the Retrofit Interface), RESTCallMethods (Making Request), and RESTInterface (Define the REST API Endpoints) in the main package and each package have Java classes.
   - With Retrofit 2, special Retrofit annotations are used for encoding details about parameters and the request method, which eventually are used for defining endpoints inside an interface. A Java interface is being deployed by Retrofit to eliminate the logic from describing the API like a charm. In Retrofit, API requests are individually represented by separate functions.
   - Every rest call will convert the json response to java object or list of java objects, and those java object's classes are defined in the package named models.
   - Downloading JSON data from any web API is made very convenient with Retrofit library. After downloading the data, the parsing process takes place to produce the Plain Old Java Object (POJO). These objects have to be defined for each "resource" in the response. POJO classes act as a bean which is used to set and get the value. @SerializedName annotation is used to specify the name of the field that’s in the JSON Response.

- Retrofit supports API calls that need authentication. The authentication here can be handled by two ways: deploying the annotations to manipulate the header and produce the request or using an OkHttp interceptor for achieving the same purpose.
- Logging Interceptor. Log Http Request and Response, it means

➢ Log Request and Response in LogCat

➢ logging levels: BODY (includes everything that the programmer had in case basic and headers. In additions, it helps the programmer to monitor request body and response body as well)

➢ To debug apps during unexpected errors

To use Retrofit, the programmer will need three major classes:

A. An Interface which defines the HTTP operations (methods)

According to Square, creators of Retrofit documentation, Retrofit turns your HTTP API into a Java interface. Sample codes for the interfaces and the methods declared in it are as below:

Sample codes RideProductsInterface.java

```
import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Header;
import retrofit2.http.Headers;
import retrofit2.http.Path;
import retrofit2.http.Query;

public interface RideProductsInterface
{
    @GET("/v1.2/products")
    Call<RideProductsModel> getProductList(@Header("Authorization")String token,
                                           @Query("latitude") String latitutde,
                                           @Query("longitude") String longitude);

    @Headers({"Accept-Language: en_US", "Content-Type: application/json"})
    @GET("/v1.2/products/{pid}")
    Call<ProductId> getProduct(@Header("Authorization")String token,
                               @Path("pid") String ProductId);

}
```

Sample codes EstimateInterface.java

```
import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Header;
import retrofit2.http.Headers;
import retrofit2.http.Query;

public interface EstimateInterface
{
    @GET("/v1.2/estimates/price")
    @Headers({"Accept-Language: en_US", "Content-Type: application/json"})
    Call<EstimatePrice> getEstimatePrice(@Query("start_latitude") String start_latitutde,
                                         @Query("start_longitude") String start_longitude,
                                         @Query("end_latitude") String end_latitutde,
                                         @Query("end_longitude") String end_longitude,@Header("Authorization") String token);

    @GET("/v1.2/estimates/time")
    @Headers({"Accept-Language: en_US", "Content-Type: application/json"})
    Call<EstimateTime> getEstimeTime(@Query("start_latitude") String start_latitutde,
                                     @Query("start_longitude") String start_longitude,
                                     @Header("Authorization") String token);
}
```

Sample codes RiderAuthentication.java

```
import retrofit2.Call;
import retrofit2.http.Field;
import retrofit2.http.FormUrlEncoded;
import retrofit2.http.GET;
import retrofit2.http.Header;
import retrofit2.http.Headers;
import retrofit2.http.POST;

public interface RiderAuthentication {

    @POST("/oauth/v2/token")
    @FormUrlEncoded
    Call<Authentiction> getauth(@Field("client_secret") String c
                                @Field("client_id") String clie
                                @Field("grant_type") String gra
                                @Field("redirect_uri") String r
                                @Field("scope") String scope,
                                @Field("code") String code);
    @GET("/v1.2/me")
    @Headers({"Accept-Language: en_US", "Content-Type: applicati
    Call<RiderMe> getme(@Header("Authorization") String token);

    @GET("v1.2/history")
    @Headers({"Accept-Language: en_US", "Content-Type: applicati
    Call<RiderHistory> getHistory(@Header("Authorization") Strir

}
```

Sample codes RideRequestInterface.java

```
import retrofit2.Call;
import retrofit2.http.Body;
import retrofit2.http.GET;
import retrofit2.http.Header;
import retrofit2.http.Headers;
import retrofit2.http.PATCH;
import retrofit2.http.POST;
import retrofit2.http.Query;

public interface RideRequestInterface
{
    @POST("/v1.2/requests/estimate")
    @Headers({"Accept-Language: en_US", "Content-Type: application/json"})
    Call<EstimatePrice> getRideRequestEstimate(@Header("Authorization") String token, @Body getRideRequestEstimate body);

    @GET("/v1.2/requests/current")
    @Headers({"Accept-Language: en_US", "Content-Type: application/json"})
    Call<RiderRequestCurrent> getRideRequestCurrent(@Header("Authorization") String token);

//    @PATCH("/v1.2/requests/current")
//    @Headers({"Accept-Language: en_US", "Content-Type: application/json"})
//    Call<>
}
```

Interface methods do individually represent API calls where each method is assigned to a specific call. To specify the request type and the relative URL, the request must contain an HTTP annotation such as GET, POST, etc. The expected results determine the return value type, which gets the response in a Call object wrapped to maintain this type. In order for parameters to distinguish the specific role of a connection, every parameter is characterized by its own annotation; for instance, @Body recognizes the post body, @Path recognizes the path parameter and @Query recognizes the query string, etc. The programmer can instruct Retrofit using @Header("Authorization") annotation to add the Authorization field to the request header with the value the programmer provides for the token.

B. Setting up the Retrofit. Issuing network requests to a REST API with Retrofit dictates that the programmer use the Retrofit.Builder class to create an instance, which should be configured with a base URL. If the programmer has more calls, which are required to be authenticated, an interceptor can be used for achieving this purpose. An interceptor here is employed for adjusting each request prior to performing it, and then altering the request header. Retrofit.Builder() can be used by the programmer to create a Retrofit instance. During the creation of a Retrofit instance, the programmer has to state the base URL and converter factory as given in the following example:

```
import okhttp3.OkHttpClient;
import okhttp3.logging.HttpLoggingInterceptor;
import retrofit2.Retrofit;
import retrofit2.converter.gson.GsonConverterFactory;

public class RiderRequestCall
{
    private static Retrofit retrofit = null;
    public static Retrofit getAuthenticate() {

        HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
        interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
        OkHttpClient client = new OkHttpClient.Builder().addInterceptor(interceptor).build();


        retrofit = new Retrofit.Builder()
                .baseUrl("https://login.uber.com/")
                .addConverterFactory(GsonConverterFactory.create())
                .client(client)
                .build();
        return retrofit;
    }
    public static Retrofit getAuthenticatetoken(String token) {

        HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
        interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
        OkHttpClient client = new OkHttpClient.Builder().addInterceptor(interceptor).build();


        retrofit = new Retrofit.Builder()
                .baseUrl("https://api.uber.com/")
                .addConverterFactory(GsonConverterFactory.create())
                .client(client)
                .build();
        return retrofit;
    }
}
```

The get Authentication () methods in the code will be called every time while setting up a Retrofit interface. The base URL will apply to all of the API call except those paths has domain definition already. The Converter Factory is the helper class to transform connection data to desire type as we define in the API interface.

Every rest call will convert the json response to java object or list of java objects, and those java object's classes are defined in the package named models. Samples codes as in the below.

1. Sample codes RideProductsInterface.java

```
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;

public abstract class RideProductCallMethod
{
    RideProductsInterface riderProductsInterface = null;
    public void getProductList(String lat, String lon, String token)
    {
        token = "Bearer "+token;
        riderProductsInterface = RiderRequestCall.getAuthenticatetoken(token).create(RideProductsInterface.class);
        Call<RideProductsModel> call = riderProductsInterface.getProductList(token,lat,lon);
        call.enqueue(new Callback<RideProductsModel>() {
            @Override
            public void onResponse(Call<RideProductsModel> call, Response<RideProductsModel> response) {
                successfun(response.body());
            }

            @Override
            public void onFailure(Call<RideProductsModel> call, Throwable t) {

            }
        });
    }

    public void getProduct(String token,String productid)
    {
        token = "Bearer "+token;
        riderProductsInterface = RiderRequestCall.getAuthenticatetoken(token).create(RideProductsInterface.class);
        Call<ProductId> call = riderProductsInterface.getProduct(token,productid);
        call.enqueue(new Callback<ProductId>() {
            @Override
            public void onResponse(Call<ProductId> call, Response<ProductId> response) {
                successfun(response.body());
            }

            @Override
            public void onFailure(Call<ProductId> call, Throwable t) {

            }
        });
```

2. Sample codes EstimateInterface.java

```
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;

public abstract class RideEstimateCallMethod
{
    EstimateInterface estimate;
    public void getEstimatePrice(String start_latitutde,String start_longitude,String end_latitutde,String end_longitude, String token)
    {
       token = "Bearer "+ token;
       estimate = RiderRequestCall.getAuthenticatetoken(token).create(EstimateInterface.class);
        Call<EstimatePrice> call = estimate.getEstimatePrice(start_latitutde, start_longitude, end_latitutde, end_longitude,token);
        call.enqueue(new Callback<EstimatePrice>() {
            @Override
            public void onResponse(Call<EstimatePrice> call, Response<EstimatePrice> response) {
                suucessfun(response.body());
            }

            @Override
            public void onFailure(Call<EstimatePrice> call, Throwable t) {

            }
        });
    }
    public void getEstimateTime(String start_latitude,String start_longitude, String token)
    {
        token = "Bearer "+ token;
        estimate = RiderRequestCall.getAuthenticatetoken(token).create(EstimateInterface.class);
        Call<EstimateTime> call = estimate.getEstimeTime(start_latitude,start_longitude,token);
        call.enqueue(new Callback<EstimateTime>() {
            @Override
            public void onResponse(Call<EstimateTime> call, Response<EstimateTime> response) {
                successfun(response.body());
            }

            @Override
            public void onFailure(Call<EstimateTime> call, Throwable t) {

            }
        });
```

3. Sample codes RideAuthentication.java

```
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;

public abstract class RiderAuthenticationCallMethod
{
    RiderAuthentication riderinterface = null;
    public void getAuthenticate(String code,String Client_secret, String Client_id, String grant_type, String redirect_uri, String scope)
    {
        riderinterface = RiderRequestCall.getAuthenticate().create(RiderAuthentication.class);
        Call<Authentiction> call = riderinterface.getauth(Client_secret,
                Client_id,
                grant_type,
                redirect_uri,
                scope,
                code);
        call.enqueue(new Callback<Authentiction>() {
            @Override
            public void onResponse(Call<Authentiction> call, Response<Authentiction> response) {
                Gson json = new Gson();
                String token = json.toJson(response.body());
                successfun(token);
            }

            @Override
            public void onFailure(Call<Authentiction> call, Throwable t) {

            }
        });
    }
    public void getinfoaboutme(String token)
    {

        token = "Bearer "+token;
        riderinterface = RiderRequestCall.getAuthenticatetoken(token).create(RiderAuthentication.class);
        Call<RiderMe> call = riderinterface.getme(token);
        call.enqueue(new Callback<RiderMe>() {
            @Override
            public void onResponse(Call<RiderMe> call, Response<RiderMe> response) {
                RiderMe obj = response.body();
```

4. Sample codes RideRequestInterface.java

```
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;

public abstract class RideRequestCallMethod
{
    RideRequestInterface rider = null;
    public void getRideRequestEstimate(String token, getRideRequestEstimate body)
    {
        token = "Bearer "+token;
        rider = RiderRequestCall.getAuthenticatetoken(token).create(RideRequestInterface.class);
        Call<EstimatePrice> call = rider.getRideRequestEstimate(token, body);
        call.enqueue(new Callback<EstimatePrice>() {
            @Override
            public void onResponse(Call<EstimatePrice> call, Response<EstimatePrice> response) {
                getRideRequestEstimate(response.body());
            }

            @Override
            public void onFailure(Call<EstimatePrice> call, Throwable t) {

            }
        });

    }

    protected abstract void getRideRequestEstimate(EstimatePrice body);
}
```

C. The last of the three needed class is a POJO that matches each field in the JSON response object gotten from querying an API. It's a class with getter and setter methods for each field.

Here are Java Classes for Resources sample codes inside model's folder

1. Product id

Information about the offered Uber product, at a given location, is returned by the products endpoint. It is expected for the response to contain the display name and other details about each product. Moreover, the response should list the products in

the proper display order. Some products, such as Uber EATS, are not returned by this endpoint.

Sample codes ProductId.java

```java
import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;

public class ProductId {

    @SerializedName("upfront_fare_enabled")
    @Expose
    private Boolean upfrontFareEnabled;
    @SerializedName("capacity")
    @Expose
    private Integer capacity;
    @SerializedName("product_id")
    @Expose
    private String productId;
    @SerializedName("price_details")
    @Expose
    private PriceDetails priceDetails;
    @SerializedName("image")
    @Expose
    private String image;
    @SerializedName("cash_enabled")
    @Expose
    private Boolean cashEnabled;
    @SerializedName("shared")
    @Expose
    private Boolean shared;
    @SerializedName("short_description")
    @Expose
    private String shortDescription;
    @SerializedName("display_name")
    @Expose
    private String displayName;
    @SerializedName("product_group")
    @Expose
    private String productGroup;
    @SerializedName("description")
    @Expose
    private String description;
```

```
public Product() {
}

public Product(Boolean upfrontFareEnabled, Integer capacity, String productId, PriceDetails priceDetails,
               String image, Boolean cashEnabled, Boolean shared, String shortDescription, String displayName,
               String productGroup, String description) {
    super();
    this.upfrontFareEnabled = upfrontFareEnabled;
    this.capacity = capacity;
    this.productId = productId;
    this.priceDetails = priceDetails;
    this.image = image;
    this.cashEnabled = cashEnabled;
    this.shared = shared;
    this.shortDescription = shortDescription;
    this.displayName = displayName;
    this.productGroup = productGroup;
    this.description = description;
}

public Boolean getUpfrontFareEnabled() { return upfrontFareEnabled; }

public void setUpfrontFareEnabled(Boolean upfrontFareEnabled) {
    this.upfrontFareEnabled = upfrontFareEnabled;
}

public Integer getCapacity() { return capacity; }

public void setCapacity(Integer capacity) { this.capacity = capacity; }

public String getProductId() { return productId; }

public void setProductId(String productId) { this.productId = productId; }

public PriceDetails getPriceDetails() { return priceDetails; }

public void setPriceDetails(PriceDetails priceDetails) { this.priceDetails = priceDetails; }

public String getImage() { return image; }
```

2. Estimate Time

The Time Estimates endpoint returns ETAs for all products currently available at a given location.

Sample codes EstimateTime.java

```java
import java.util.List;
import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;

public class EstimateTime {

    @SerializedName("times")
    @Expose
    private List<Time> times = null;

    /**
     * No args constructor for use in serialization
     *
     */
    public EstimateTime() {
    }

    public EstimateTime(List<Time> times) {
        super();
        this.times = times;
    }

    public List<Time> getTimes() { return times; }

    public void setTimes(List<Time> times) { this.times = times; }

}
```

3. Estimate Price

An estimated price range is returned for each product, offered at a given location, by the Price Estimates endpoint. The return value here is given in the form of a string that elaborates the full price range and the local currency symbol for the given price estimate.

Sample codes EstimatePrice.java

```
import java.util.List;
import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;

public class EstimatePrice {

    @SerializedName("prices")
    @Expose
    private List<Price> prices = null;

    /**
     * No args constructor for use in serialization
     *
     */
    public EstimatePrice() {
    }

    public EstimatePrice(List<Price> prices) {
        super();
        this.prices = prices;
    }

    public List<Price> getPrices() { return prices; }

    public void setPrices(List<Price> prices) { this.prices = prices; }

}
```

3.5.2. Publishing Open Source Release with Gradle

When developing a Java open source project, the common conclusion the programmer always ends up with is to share the produced outcomes with the developer community, which should be the least objective in the Java world. This is mainly achieved by publishing the artifacts to a public accessible Maven repository. The programmer should publish his artifacts to a custom repository not to Maven Central because Maven Central is the Maven repository that is used by default in most Maven and Gradle builds and thus is much more

accessible. The reason for this is the programmer can play around with his publishing routine in his own repository first and then publish it to Maven Central from there.

To publish artifacts on Bintray, the programmer naturally needs an account and following these steps:

1. Create a Repository, which is a container of artifacts of a particular type e.g. Maven
2. Setup build.gradle file

```
plugins {
  id "com.jfrog.bintray" version "1.7.3"
  id "maven-publish"
  id "java"
}

buildscript {
  repositories {
    mavenLocal()
    mavenCentral()
    jcenter()
  }
}

repositories {
  mavenLocal()
  mavenCentral()
  jcenter()
}
```

Our project's dependencies are searched for by the two repositories closures that list the Maven repositories, and these have nothing to do with publishing our artifacts.

3. Build Sources and Javadoc Artifacts

```
task sourcesJar(type: Jar, dependsOn: classes) {
    classifier = 'sources'
    from sourceSets.main.allSource
}

javadoc.failOnError = false
task javadocJar(type: Jar, dependsOn: javadoc) {
    classifier = 'javadoc'
    from javadoc.destinationDir
}

artifacts {
    archives sourcesJar
    archives javadocJar
}
```

In order to successfully publish an open source project, the normal JAR should be published along with a JAR including the sources as well as a JAR including the javadoc. Following this method enables the convenient use of the published project by developers as the direct download of JARs is possible by IDEs where sources can be used, checked or modified in their editors.

4. Define what to publish

   Next, the programmer determines the exact artifacts to be published as well as the relevant metadata that elaborates them. During the publishing process, the programmer places some metadata in the pom-Config variable, specifically in pom.xml. The interesting part is the publishing closure which is provided by the maven-publish plugin we applied before. The programmer defines a publication called Bintray-Publication. This publication should contain the default JAR file (components.java) as well as the sources and the Javadoc JARs. Also, we provide the Maven coordinates and add the information from pom-Config above.

5. Provide Bintray-specific information

```
bintray {
    user = System.getProperty('bintray.user')
    key = System.getProperty('bintray.key')
    publications = ['mavenPublication']

    pkg {
        repo = 'TaxiServiceModule'
        name = 'TaxiServiceModule'
        userOrg = 'dilkhaz'
        licenses = ['Apache-2.0']
        publish = true
        vcsUrl = 'https://github.com/dilkhaz/TaxiServiceModule.git'
        version {
            name = '1.0.0'
            desc = '1.0.0'
            released  = new Date()
        }
    }

}
```

Finally, the part where the action is. Add the following to your build.gradle to enable the publishing to Bintray:

6. Upload Library

```
gradlew bintrayUpload -Dbintray.user=<YOUR_USER_NAME> -Dbintray.key=<YOUR_API_KEY>
```

To obtain API key sign in on Bintray account, go to the "edit profile" page and click on "API Key" in the menu.

7. Then run the build and upload the artifacts on Bintray by running
8. The files have now been uploaded to Bintray. So, as a next step, the programmer might want to publish his artifacts to the well-known JCenter repository.

In summary, to upload a library on Bintray Cloud Platform, the programmer naturally needs an account there. The reason for uploading to Bintray is that the programmer still has control over his files even after uploading and publishing his files. Next, the programmer needs to create a Repository and define what to publish. Once a library is up on Bintray, simply hit add to JCenter button to sync with JCenter as shown in Figure 3.6.

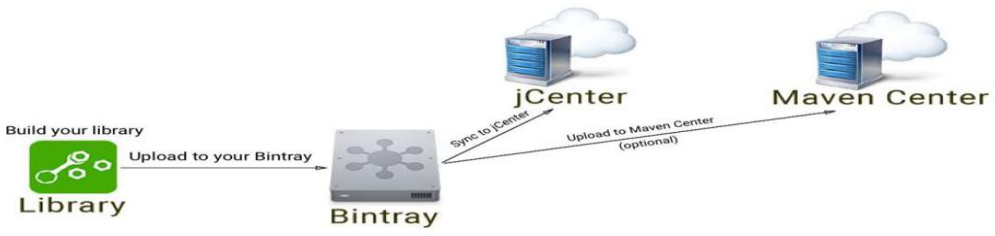


Figure 3.6. Upload Library

3.5.3. Develop Android App using the Library

Android projects can employ codes included in the JAR files (Java libraries). If the programmer wants to use libraries, then this can be only achieved by using APIs available in Android. For instance, both Java java.awt and javax.swing packages are not available in Android.

The second part of the Multi-OS Engine App is to create an Android App

1. Adding Internet Permission

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Because the programmer is working with the network so the programmer has to add INTERNET permission into AndroidManifest.xml file.

2. Declaring Dependencies

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    //noinspection GradleCompatible
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    implementation 'com.android.support:cardview-v7:28.0.0'
    implementation 'com.android.support:recyclerview-v7:28.0.0'
    implementation 'com.google.code.gson:gson:2.8.5'
    implementation 'com.squareup.picasso:picasso:2.71828'
    //implementation 'com.android.support:design:28.0.0'
    implementation ('com.google.android.gms:play-services:7.3.0')
            {
                exclude module: 'support-v4'
            }
    implementation project(path: ':common')
    //implementation 'dilkhaz:TaxiServiceModule:2.0.0'
}
```

add Recycler view and Picasso dependencies, since the programmer will be using Picasso to load image URLs into an image View and the programmer will display the data details in a Recycler View. There are other libraries the programmer will also add that will provide new functionalities and view components for us.

3. create three sub packages named Activates, Adapter, and ViewHolder in the main package.

Sample codes for loginActivity.java. Have the rider sign in and authorize your app to access their Uber account.

```
public class LoginActivity extends AppCompatActivity {
    private static final String Oauthurlv2 = "https://login.uber.com/oauth/v2/authorize?response_type=code&" +
            "client_id=slkxC2hXgjqKDXYBIaCnMPYguLpZQAK8&redirect_uri=https://com.dilkhaztaxi.myapp/redirect";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);
    }

    public void Login(View view) {
        Intent intent = new Intent(Intent.ACTION_VIEW,Uri.parse(Oauthurlv2));
        startActivity(intent);
    }
}
```

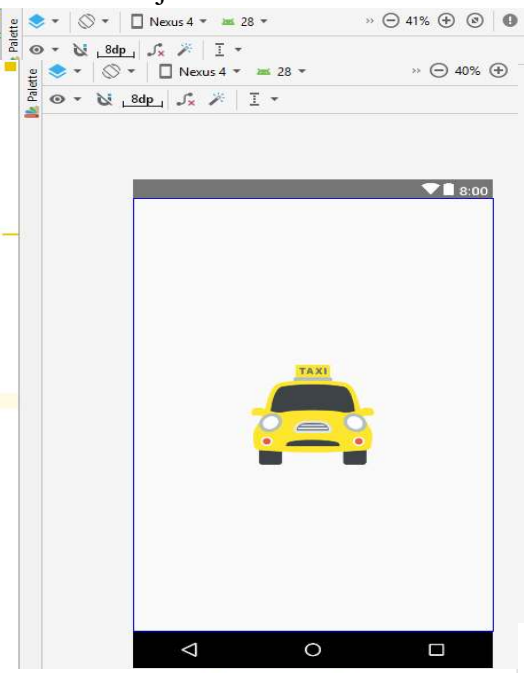
Sample codes MainActivity.java

```
package dilkhaz.com.dilkhaztaxiservice.Activities;

import android.app.Dialog;
import android.app.ProgressDialog;
import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.GridLayoutManager;
import android.support.v7.widget.RecyclerView;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.widget.ProgressBar;
import android.widget.TextView;
import android.widget.Toast;

import com.google.android.gms.common.GooglePlayServicesNotAvailableException;
import com.google.android.gms.common.GooglePlayServicesRepairableException;
import com.google.android.gms.location.places.Place;
import com.google.android.gms.location.places.ui.PlacePicker;
import com.google.gson.Gson;

import dilkhaz.com.dilkhaztaxiservice.Adapater.EstimatePriceAdapter;
import dilkhaz.com.dilkhaztaxiservice.Adapater.EstimateTimeAdapter;
import dilkhaz.com.dilkhaztaxiservice.R;
import pialpha.com.common.Models.Authentication.Authentiction;
import pialpha.com.common.Models.Estimate.EstimatePrice;
import pialpha.com.common.Models.Estimate.EstimateTime;
import pialpha.com.common.RestCallMethods.RideEstimateCallMethod;

public class MainActivity extends AppCompatActivity {
    public static final String MyPREFERENCES = "MyPrefs";
    SharedPreferences sharedpreferences;
    Dialog mdialog;
    RecyclerView myrv;
    private Authentiction user;
    RideEstimateCallMethod rideEstimateCallMethod;
```

```
private String startlat="";
private String startlong="";
private String endlat="";
private String endlong="";
ProgressDialog progressDoalog;
PlacePicker.IntentBuilder builder;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mdialog = new Dialog( context: this);
    init();
    initdialog();
    builder = new PlacePicker.IntentBuilder();
}


//starts here
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    Intent i;
    // Handle item selection
    switch (item.getItemId()) {
        case R.id.menu_Home:
//              i = new Intent(getApplicationContext(), MainActivity.class);
//              i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
//              startActivity(i);
            return true;

        case R.id.menu_History:
            i = new Intent(getApplicationContext(), HistoryActivity.class);
            i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
```

Samples codes for creating an adapter and viewholder java classes for EstimatePrice

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.co
    xmlns:app="http://schemas.android.com/apk/res-auto"
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"

    tools:context=".Activities.SplashScreenActivity">

    <ImageView
        android:id="@+id/accountinfo_photo"
        android:layout_width="140dip"
        android:layout_height="140dip"
        android:layout_gravity="center_horizontal"
        android:src="@drawable/taxi_dilkhaz"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</android.support.constraint.ConstraintLayout>
```




```
    @Override
    public void onBindViewHolder(@NonNull EstimatePriceViewHolder holder, final int i) {
        holder.Name.setText(mlist.get(i).getDisplayName());
        holder.Distance.setText(mlist.get(i).getDistance().toString());
        holder.Time.setText(mlist.get(i).getDuration().toString());
        holder.Fare.setText(mlist.get(i).getCurrencyCode()+" "+mlist.get(i).getEstimate());
        holder.Card.setOnClickListener((view) -> {
                Toast.makeText(mcontext,mlist.get(i).getProductId(),Toast.LENGTH_SHORT).show();
        });
    }

    @Override
    public int getItemCount() { return mlist.size(); }
}
```

the RecyclerView component will use.

4. Creating layout files

   A layout resource is used for defining the architecture of the UI in an Activity or a component of the same UI.

   Sample codes for activity_login.xml

   Sample code for activity_splash_screen.xml

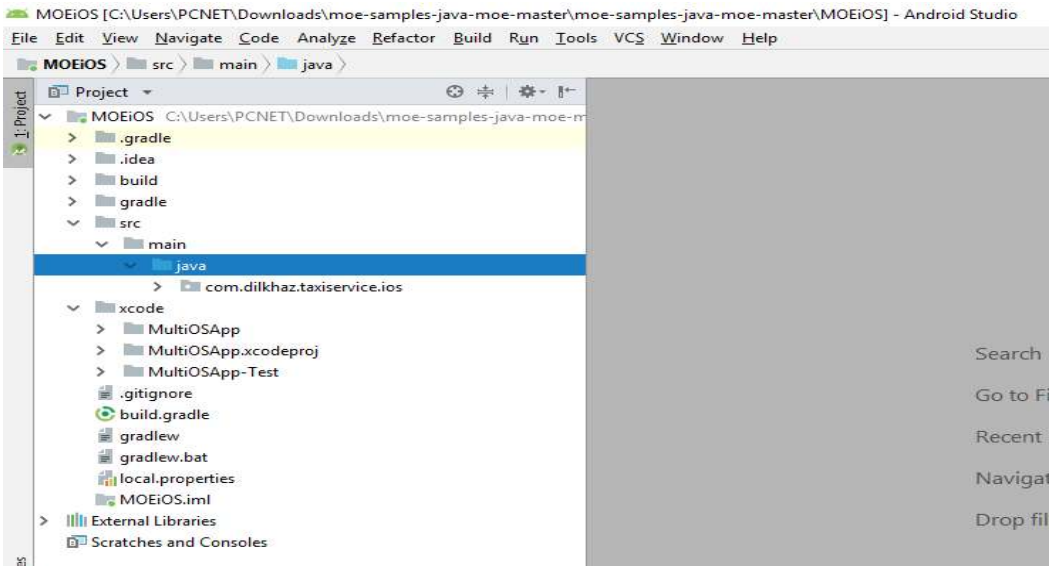
5. Android app is finished. Build and Test the App

3.5.4. Develop iOS App using the Library

Before diving, to the iOS App, the programmer will need a MacOS (two extra actions in Multi-OS Engine and those options only available on Mac).

The last part of the Multi-OS Engine App is to create iOS App

1. Since the shared application Java code is ready, the programmer should be able to start with creating views. Initially, the programmer needs to remove the UI package with the generated class.
2. Selecting Open project in XCode from the Right-click menu of iOS is used for getting started with iOS views,
3. Create the View Controller and Storyboard
4. To continue the programmer should have his views created and linked to view controllers' classes headers
5. After these steps, the task relevant to the XCode development should be finished. Next, binding the created UI to the rest of the app parts written in Java is necessary. doing so is achieved as follows: from Android Studio > Right click on iOS module > select Multi-OS Engine Actions > Create New Binding named NBC file MyViewController > next select green +, select Header option and named again MyViewController



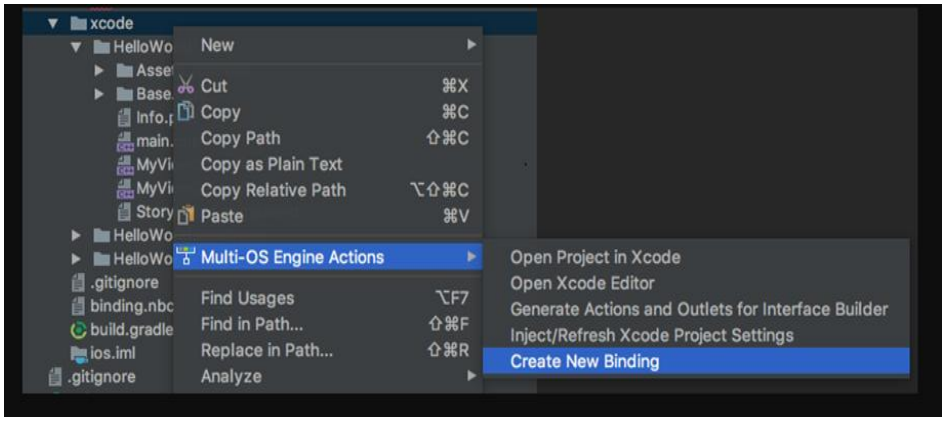


Figure 3.7. Create Binding

6. Set Header path to xcode/MoeiOS, Base package name to the project package, and Import headers section write #import "MyViewController.h"
   Next, click the settings icon and choose Generate Bindings

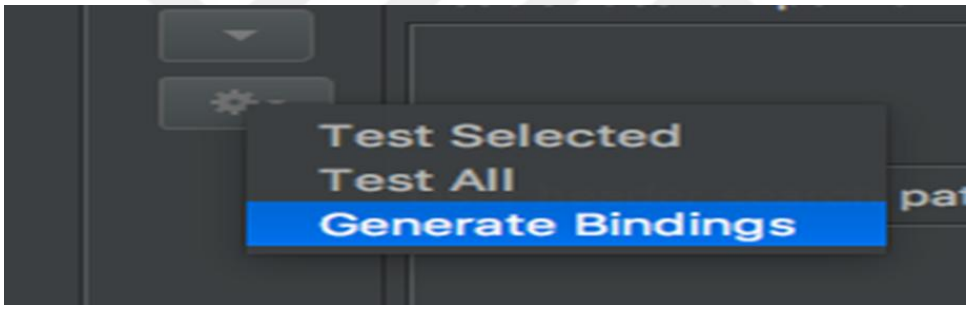


Figure 3.8. Generate Binding

7. Generate binding finished successfully. In iOS module in java → {package} → src there is generated file, called MyViewController.java
8. Now you are able to use your Java classes from commons
9. Write code in Java
   Open MyViewController class for editing. First of all, it is necessary to add some annotation on the top of class definition. Next override required method. Access desired method by removing the @Generated annotation, native keyword and adding a body.
10. iOS app is finished now. Run app normally.

4. CONCLUSION

Libraries are well-defined and are designed for reuse throughout implementation. For example, a website may have multiple web pages that implement the same navigation bar or text-field, but none of these objects have a relation to one another. And the mobile application development services have evolved into a higher level with APIs and when developers build apps for the mobile, they rely heavily on APIs for connectivity. Which allows them to communicate seamlessly with the enterprise. In facts, APIs accelerate mobile development and enable tremendous agility for organizations that are going through their own digital transformation.

The advantage of this work is the programmer provided a way to share code between Android and iOS. When developing a Java open source project, the common conclusion the programmer always ends up with is to share the produced outcomes with the developer community, which should be the least objective in the Java world. This is mainly achieved by publishing the artifacts to a public accessible Maven repository. The programmer should publish the artifacts to a custom repository not to Maven Central because Maven Central is the Maven repository that is used by default in most Maven and Gradle builds and thus is much more accessible. The reason for this is that the programmer can play around with his publishing routine in his own repository first then publish it Maven Central from there (or JCenter, for that matter, which is another well-known Maven repository). Another reason for uploading to Bintray and not to Maven Central is that the programmer still has control over his files even after uploading and publishing his files whereas in Maven Central the programmer loses all control after publishing.

4.1. Future Work

The programmer will work on the iOS version of the App in the future. Most developers choose to build an app for one for one platform to start and then launch the app on the other platform later once the first version of the app is established and successful.

Here are the reasons why the programmer did not complete the iOS version:

- Multi-OS Engine Framework is still technology preview not mature.
- Unpopularity currently the biggest issue with Multi-OS Engine

- The tutorial and documentation are out of date
- Not fully support to windows OS
- It takes a while just to generate default iOS module
- Huge RAM usages
- Unstable work of UI binding generation feature. With correct. storyboard; .h files UI binding generation procedure may require more than one attempt.
- No community supports

REFERENCES

[1] Developers | Uber. (n.d.). Retrieved from https://developer.uber.com/docs/riders/ride-requests/tutorials/api/introduction.

[2] Using Retrofit 2.x as REST client - Tutorial. (n.d.). Retrieved from http://www.vogella.com/tutorials/Retrofit/article.html.

[3] Consuming APIs: Getting started with Retrofit on Android. (2018, October 25). Retrieved from https://www.androidauthority.com/retrofit-android-tutorial-906928/.

[4] Download Android Studio and SDK tools | Android Developers. (n.d.). Retrieved from https://developer.android.com/studio/

[5] Developing a mobile cross-platform library - Part 1. Exploring | Skyscanner's Travel Blog. (2014, February 11). Retrieved from https://www.skyscanner.net/blogs/developing-mobile-cross-platform-library-part-1-exploring.

[6] Sharma, K. Android in opposition to iPhone.

[7] Bintray - Download Center Automation & Distribution w. Private Repositories. (n.d.). Retrieved from https://bintray.com/.

[8] Log Packer. (2016, March 11). Go Mobile: Library development for IOS/Android? Log Packer? Medium. Retrieved from https://medium.com/@LogPacker/gomobile-library-development-for-ios-android-8711e76817d8.

[9] Welcome to Multi-OS Engine Documentation! — Multi-OS Engine Documentation. (n.d.). Retrieved from https://doc.multi-os-engine.org/.

[10] Gradle Build Tool. (n.d.). Retrieved from https://gradle.org/.

[11] Michael J. Garbage. (2018, August 23). Native vs. cross-platform app development: pros and cons. Retrieved from https://codeburst.io/native-vs-cross-platform-app-development-pros-and-cons-49f397bb38ac.

[12] Build software better, together. (n.d.). Retrieved from https://github.com/.

[13] Java SE Development Kit 8 - Downloads. (n.d.). Retrieved from https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

[14] Retrieved from https://jcenter.bintray.com/.

[15] Tom. (2017, December 4). Publishing Open Source Releases with Gradle. Retrieved from https://reflectoring.io/guide-publishing-to-bintray-with-gradle/.

[16] Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies? Empirical Software Engineering, 23(1), 384-417.

[17] Yener, M., & Dundar, O. (2016). Expert Android Studio. John Wiley & Sons.

CURRICULUM VITAE

Dilkhaz Y. Mohammed Haji Mohammed was born in 1980 at Duhok city - Iraq. Attended primary School at Zawita Primary School from 1986-1992 and Secondary School at Kawa High School from 1993-1999. He obtained a BSc from Duhok University -College of Science- Computer Science Dept.2000-2004. In 2017, He became a Master Degree Student at FIRAT University - Software Engineering Department.

Contact

Address : Iraq- Al-Duhok – Diyari.

Email : dilkhaz.mohammed@gmail.com