ECE 651 Deliverable 4

Metadata:

1. Project Title:
UW Carpool Application

2. Team Members Information

Name	Quest ID	Student ID
Jiatong He	j246he	20738590
Shuting Lian	s6lian	20718506
Zhilun Chang	z9chang	20727042
Zizheng Jiang	z225jian	20716960

Architecture

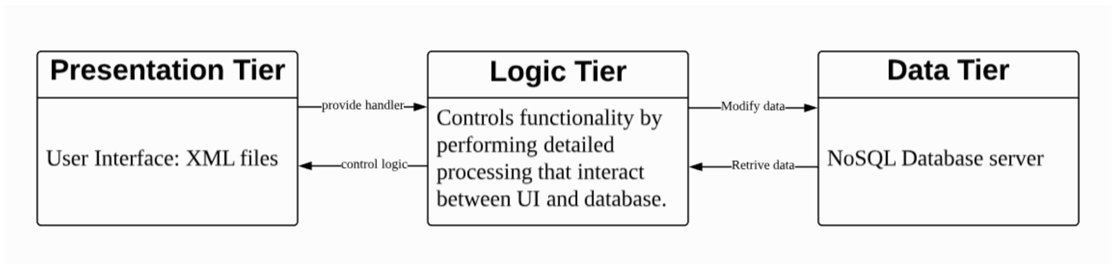
1. Application Introduction
The UW Carpool Application is an app that helps users publish and search carpool information. It is designed to provide convenience and save time for users. Compared with bulletin board system (BBS) and Facebook Groups, the application can only display the specific carpool information based on users' needs. Users no longer need to filter what they want from a variety of information.

2. Application Functional Properties Implementation
In our project proposal, we have proposed some functional properties such as login and register functional properties, driver functional property, passenger functional property, and database functional property.

There are 4 types of architectural styles we consider in the project. They are Object-oriented style, Client Server style, Blackboard style and Event-based style.

The whole Android project is based on Object-oriented architecture. Each of activity class is inherited from AppCompatActivity. Meanwhile, we encapsulate user authentication, user and trip collection etc. in different classes. The instances of these classes are created and used in various activities. All the concepts I mentioned are introduced in Object-oriented architecture.

Client-Server architecture is an architectural deployment style that describes the separation of functionality into layers with each segment being a tier. In particular, we divide the program into 3 different tiers: Presentation tier, Logic tier and Data tier. Presentation tier is the topmost level of the application. It provides the application's user interface in our project. Specifically, in Android project, the user interface is showed by several xml files. Logic tier controls an application's functionality by performing detailed processing that interact between UI and database. It retrieves data from Data tier, achieve functionality and show result to the UI. The Data tier is our database server which stores user information and trip information. Logic layer can access those data through API it provides. 3-tiers Client-Server architecture is shown on Fig. 1.



Presentation Tier
User Interface: XML files
provide handler
control logic
Logic Tier
Controls functionality by performing detailed processing that interact between UI and database.
Modify data
Retrive data
Data Tier
NoSQL Database server

Fig. 1. 3-tiers Client-Server Architecture

Blackboard architecture is used for the interaction with database. Various activities can access data from sUserAdapter which contains data retrieved from database.

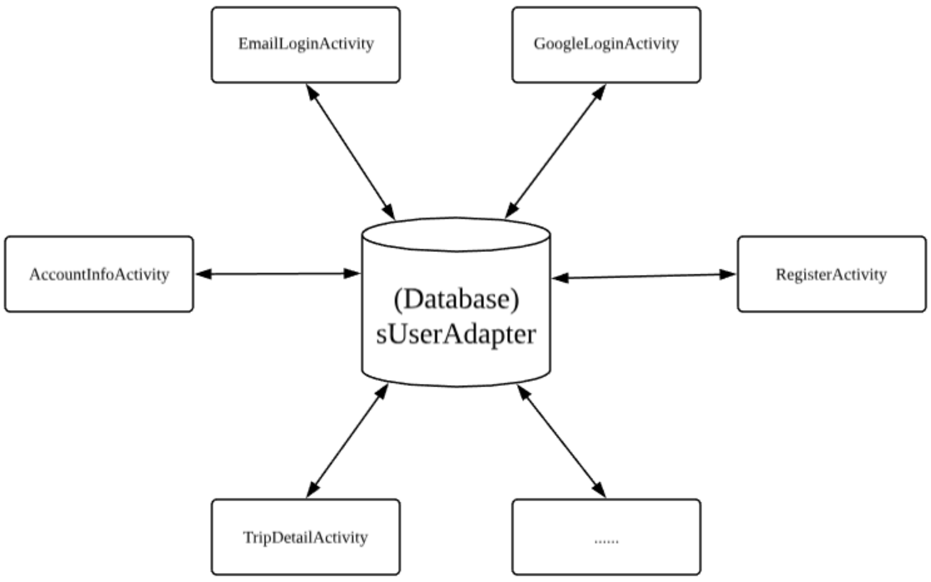


Fig. 2. Blackboard Architecture

Event-based architecture is an architectural design that uses the production and consumption of events to control behaviour. There is a lot of listeners that are used in the project. Each operation on buttons in UI is an event, program will change state and execute logic when a user presses a button.

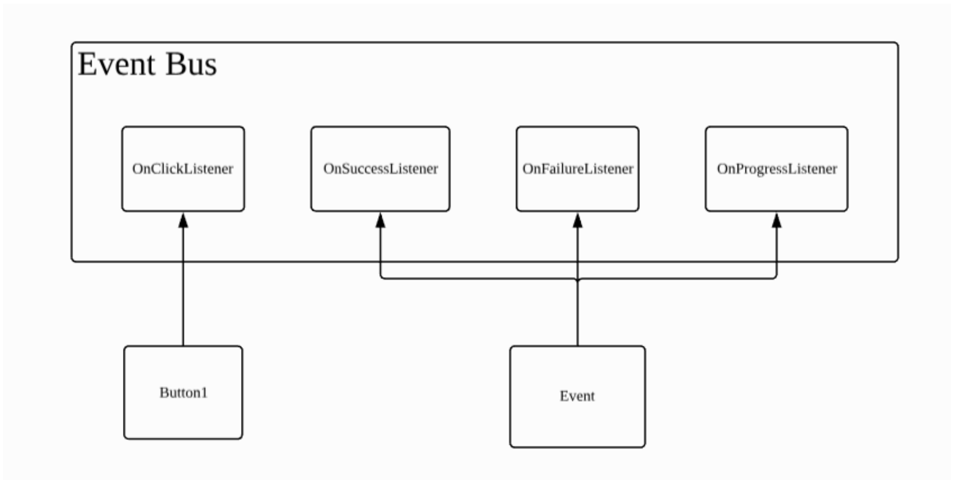


Fig. 3. Event-based Architecture

2.1. Login and Register Functional Properties

When the user opens the application, the default display interface is the main interface. Before that, the application will check whether the user has already logged in to the app. If the user has not logged in yet, the application will jump to the login interface, otherwise, it will jump to the main interface. After users select the corresponding login method and log in, the application will go to the database to check if they are new users. If they are new users, the application will jump to the registration interface, and the user will fill in the first name and last name to complete the registration. After they click the confirmation button, their names will insert into the database. If they are not new users, they will jump to the main interface after logging in. This implementation mainly solves the problem that the user wants to log in on multi-device and the user wants to log in to multiple accounts on one device.

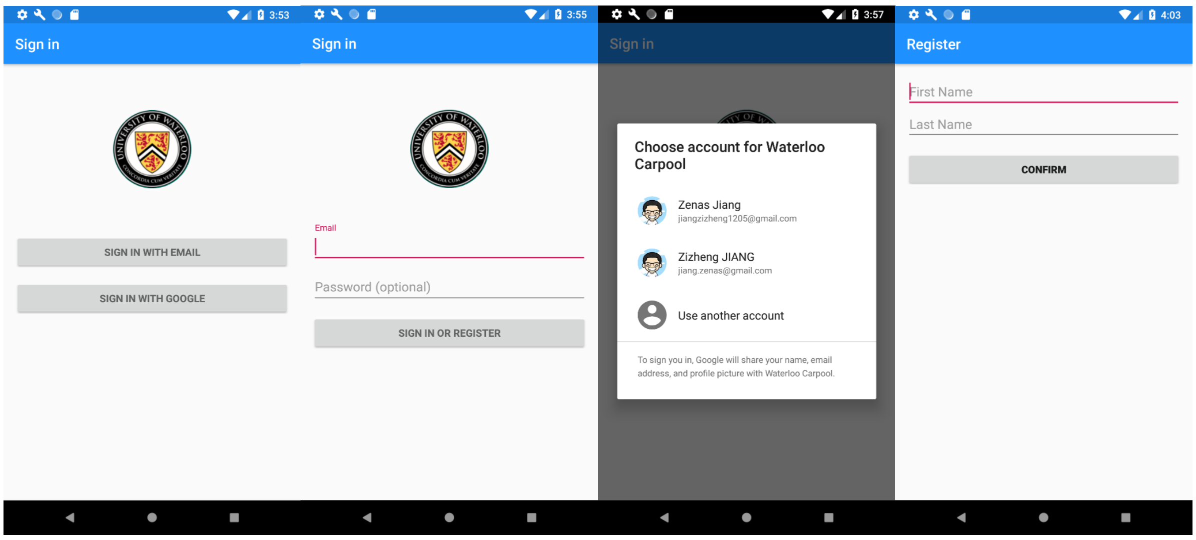


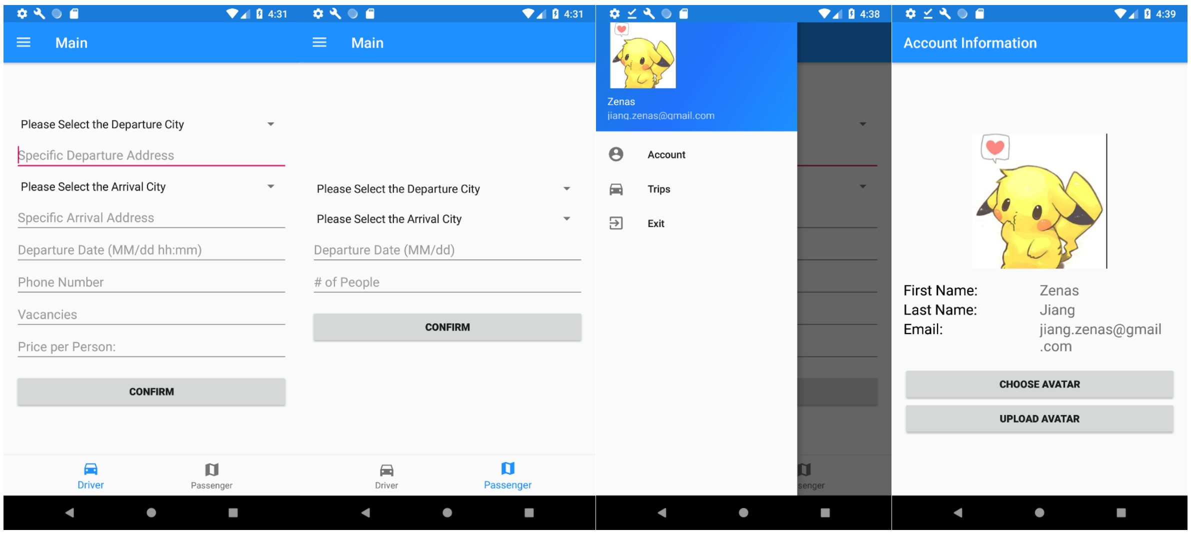
Fig. 4. Login and Register Interface

2.2. Main Activity Functional Properties

The main interface includes three sub-interfaces which are driver and passenger fragments interface, drawer interface and account information interface. Different from the proposal, we moved the navigation button (Driver and Passenger) from the top of the screen to the bottom of the screen, which is more user-friendly and convenient for the user to operate with one hand.

From Fig. 5. we can see that the driver and passenger interface is basically consistent with our proposal. The logic behind driver interface is that when the driver enters the relevant information of carpool and clicks the confirmation button, the app will generate a Trip object, store the relevant information into the Trip and then add the Trip object to the database. The logic behind passenger interface is that when the passenger enters the relevant information of carpool and clicks the confirm button, the app will generate a search query based on the information entered by the passenger, then find the eligible data in the database and return it in a specific format.

After implementing the basic functionality in the proposal, we added some features, such as the drawer navigation interface. In this interface, users can see their basic information, set their avatars, and exit the application.

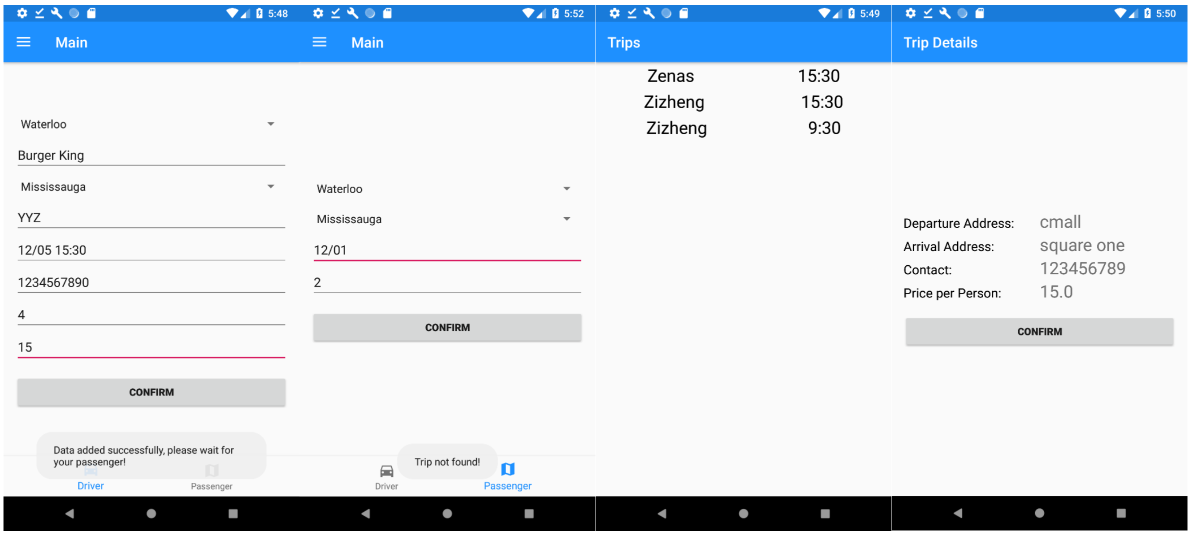


First Name:	Zenas
Last Name:	Jiang
Email:	jiang.zenas@gmail.com

Fig. 5. Drawer and Main Interface

2.3. Database Functional Properties

There are many occasions where the front-end interacts with the database, such as adding user information to the database in the registration interface, viewing personal trips in the Trips interface, and adding and searching in driver and passenger interfaces.



Waterloo
Burger King
Mississauga
YYZ
12/05 15:30
1234567890
4
15

Data added successfully, please wait for your passenger!

Waterloo
Mississauga
12/01
2

Trip not found!

Zenas	15:30
Zizheng	15:30
Zizheng	9:30

Departure Address:	cmall
Arrival Address:	square one
Contact:	123456789
Price per Person:	15.0

Fig. 6. Interact with Database

Fig. 6. shows how the driver and passenger interfaces interact with the database. There is a confirm button listener in the application which will create a Trip object based on the data entered by the driver, add it to the Trip collection in the database when the driver inputs relevant data into the interface and click the confirm button. If the data is added successfully, the Toast which shows the data added successfully will pop up. In the passenger interface, we have written a search query related to the input data the passenger provides. If the application can get the data based on the what passenger provides, it will jump to the recycler view, otherwise it will pop up the Toast showing the Trip not found. Each line in the recycler view shows one trip data including the drive’s name and the departure time. When the user clicks the trip-line shows on the recycler view, it will give us the trip details information such as departure address, arrival address, contact information, and the price.

3. Application Non-Functional Properties Implementation

3.1. Response time

In the project, we use Recycler View rather than List View because we have to make sure our app can respond as fast as possible. Compared with List View, the Recycler View has some advantages:

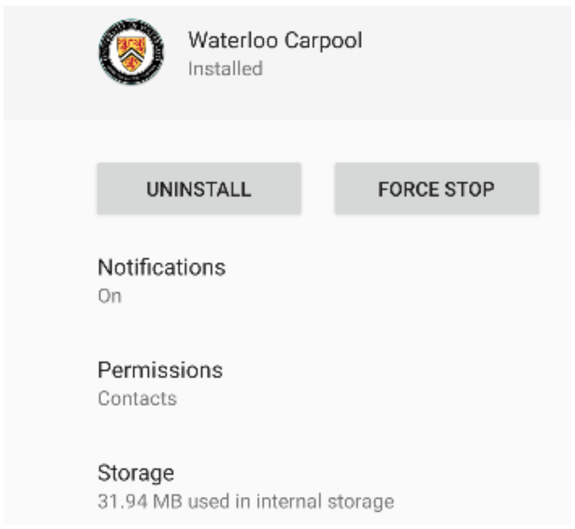
1) Every time when the activities get data from the database, the List View adapter loads all the data even if part of the data may not display on the screen. Different from List View, the Recycler View only loads the data that need to display on the screen.
2) Because the amount of data loaded by Recycler View each time is less than the amount of data loaded by List View, so using Recycler View can use less memory than List View.

In the project, we use parcelable instead of serializable when the passenger wants to know the trip details for a specific driver. Compared with serializable, the parcelable has some advantages:

1) Parcelable process is much faster than serializable. One of the reasons for this is that we are being explicit about the serialization process instead of using reflection to infer it [1].
2) Serializable is more focused on the network data transfer and long-term preservation of data. It’s not suitable in our app since we don’t need to store huge data for a long period.

3.2. File size

In our proposal, we wrote that the overall size of the mobile application should be less than 100 MB. As can be seen from Fig8, the size of our app is 31.94 MB which meets our demand.


Waterloo Carpool
Installed

Notifications
On

Permissions
Contacts

Storage
31.94 MB used in internal storage

Fig. 8. UW Carpool app information

3.3. Scalability & Reliability

The NoSQL database of our project is deployed on a professional third-party platform which has 24 x 7 customer support and ensures reliability. Also, each user has no authentication of modifying data of other users in the database, so there are no conflicts between each user. We have tested that our application can accept at least 20 users online simultaneously.

The Login-Register part of our project allows extension on more login methods, such as Facebook Login, Twitter Login and etc.

3.4. Compatibility

Our Android project will use API 21: Android 5.0 (Lollipop), by targeting it, the app will run on approximately 85.0 % of devices. This data was referenced from Android Platform/API Version Distribution.

System Design

1. Preparation Before Start

1.1. IDE

When we first design an Android application. We need to know how to use Android Studio, which is the IDE to develop our project. After we download the Android Studio from the Internet, we need to download and install the related software development kit (SDK) to make sure the environment is fine for us to development the app.

1.2. Android Activities Lifecycle [2]

All the Activities actually inherit from the AppCompatAcitivty() class, the Activity class provides a core set of six callbacks: onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestory(). The system invokes each of these callbacks as an activity enters a new state. When we develop the application, we first need to know what each callback does and then add related code in each callback function. Here, we introduce three important callbacks in detail.

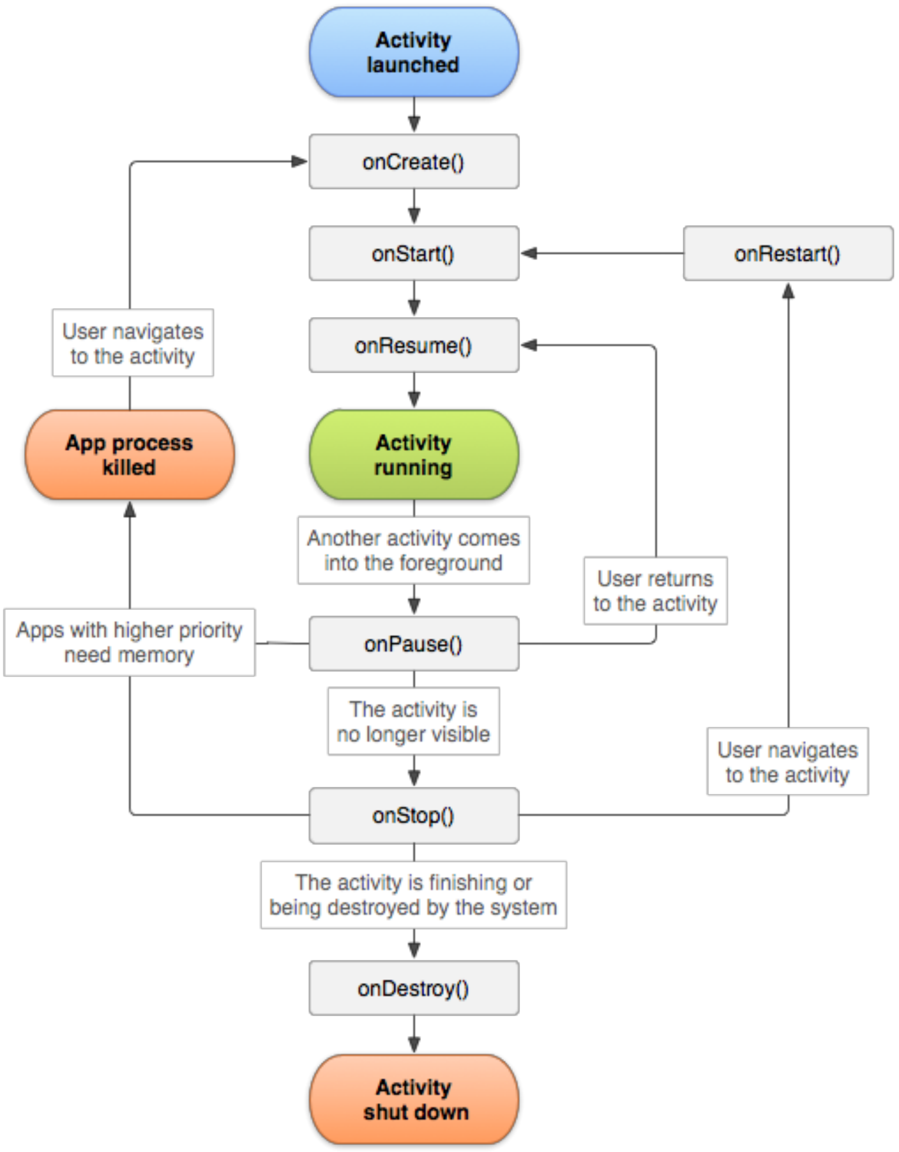

Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Fig. 9. Android Activity Lifecycle

1.2.1. onCreate()

On activity creation, the activity enters the Created state. In the onCreate() method, the basic application startup logic should happen only once for the entire life of the activity. For example, the implementation of onCreate() might bind data to lists, associate the activity with a ViewModel, and instantiate some class-scope variables. This method receives the parameter saveInstanceState, which is a Bundle object containing the activity's previously saved state. If the activity has never existed before, the value of the Bundle object is null.

1.2.2. onStart()

When the activity enters the Started state, the system invokes this callback. The onStart() call makes the activity visible to the user, as the app prepares for the activity to reach the foreground and become interactive. For example, this method is where the app initializes the code that maintains the UI.

The onStart() method completes very quickly and, as with the Created state, the activity does not stay resident in the Started state. Once this callback finishes, the activity enters the Resumed state, and the system invokes the onResume() method.

1.2.3. onResume()

When the activity enters the Resumed state, it comes to the foreground, and then the system invokes the onResume() callback. This is the state in which the app interacts with the user. The app stays in this state until something happens to take focus away from the app. Such an event might be, for instance, receiving a phone call, the user's navigating to another activity, or the device screen's turning off.

When the activity moves to the resumed state, any lifecycle-aware component tied to the activity's lifecycle will receive the ON_RESUME event.

When an interruptive event occurs, the activity enters the Paused state, and the system invokes the onPause() callback. If the activity returns to the Resumed state from the Paused state, the system once again calls onResume() method.

2. Key patterns

2.1. Model – View - Presenter (MVP) pattern

Model – view – presenter is a derivation of the model – view – controller pattern and is used mostly for separating user interfaces from logical operation. In MVP, all presentation logic is pushed to the presenter.

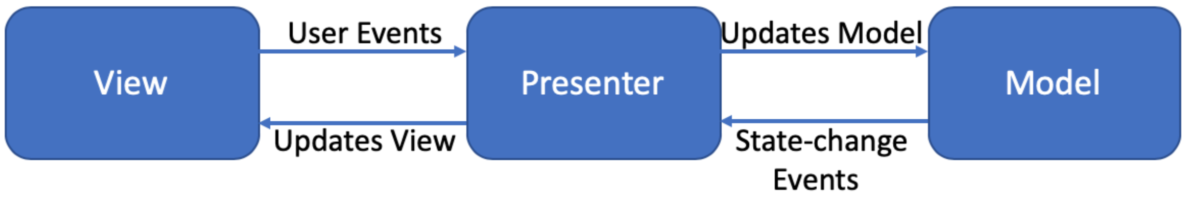


Fig. 10. Model – View – Presenter Pattern

The Android structure is mainly based on the MVP design pattern. In Android, we can consider the layout as view, the activity as presenter and the database as the model.

In an application with a good layered architecture, the model would only be the gateway to the domain layer or business logic. We can see it as the provider of the data we want to display in the view. Model's responsibilities include using APIs, caching data, managing databases and so on.

The view, usually implemented by an Activity's layout, will contain a reference to the presenter. The only thing that the view will do is to call a method from the presenter (activity) every time there is an interface action.

The presenter is responsible to act as the middle man between view and model. It retrieves data from the model and returns it formatted to the view, and it gets the action from the view and makes changes to the view or the model.

In our project, most activities interact with the database. For example, in the register activity, after the user inputs the first name and the last name, the application will store the data in the user's collection at the database. The activities can also jump to each other's interface according to the instructions sent from the view.

2.2. Adapter Pattern

The adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

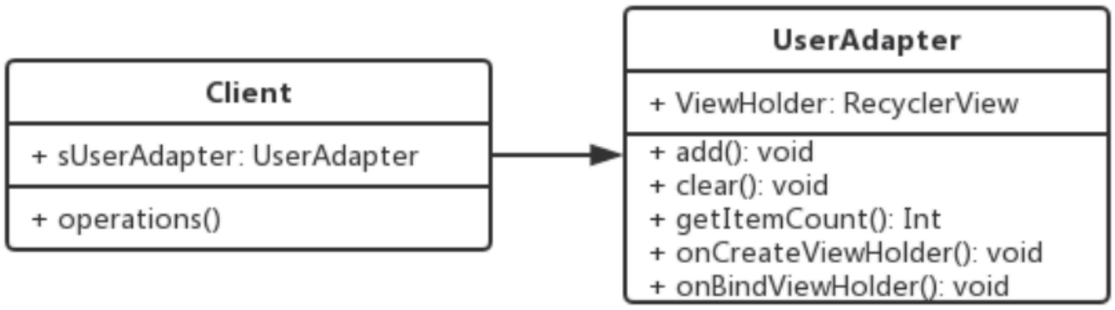

Client
+ sUserAdapter: UserAdapter
+ operations()
UserAdapter
+ ViewHolder: RecyclerView
+ add(): void
+ clear(): void
+ getItemCount(): Int
+ onCreateViewHolder(): void
+ onBindViewHolder(): void

Fig. 11. Adapter Pattern in Our Project

In our project, after the passenger retrieving the related trips information from the database, we need to display those data on the user's screen. We implemented this through the UserAdapter object in our code. The UserAdapter in our app provides the ViewHolder which can format the data we get from the database and display the driver's name and departure time on the screen (Fig. 7.).

2.3. Data Access Object Pattern

The data access object (DAO) is an object that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, the DAO provides some specific data operations without exposing details of the database.

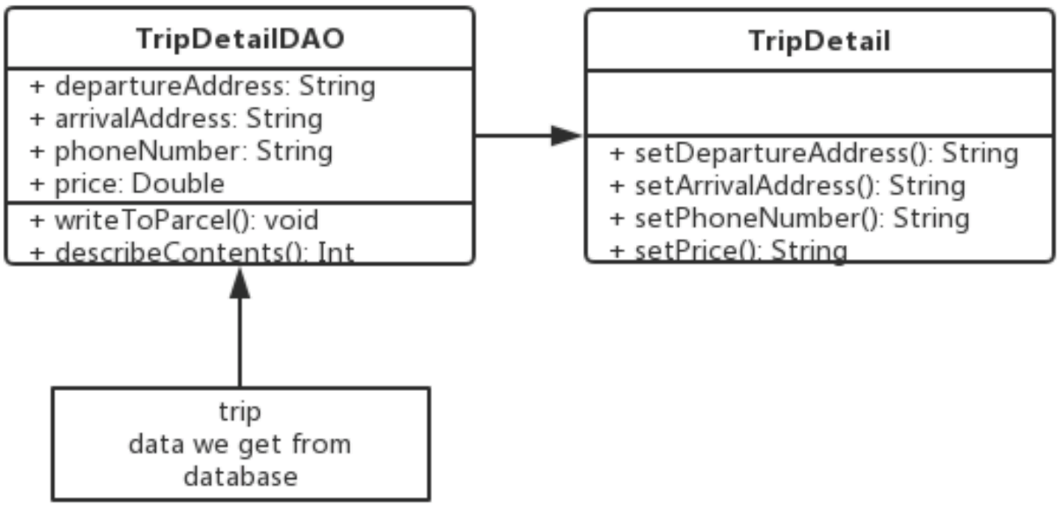
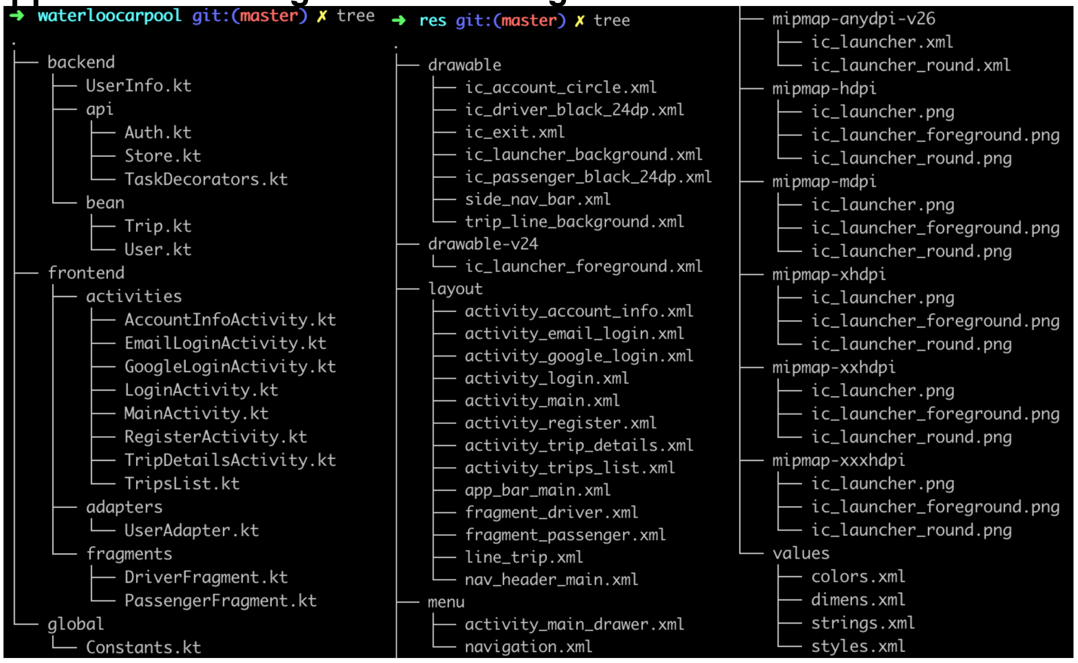


Fig. 12. Data Access Object Pattern in Our Project

In our project, after the passenger choose a specific drive's trip on the screen, the application will jump to the trip details interface showing the related information such as the departure address, the arrival address, the driver's contact information and the price for the trip. How does the data pass between those two activities and show on the screen? It's because we use the TripDetail data class as an interface in our code. The object tripDetail can get the data from the database and pass it to the TripDetail activity. In this way, the tripDetail provides some specific data operations without exposing details of the database.

3. Application Catalog and Class Diagram



```
➜ waterloocarpool git:(master) ✗ tree
.
├── backend
│   ├── UserInfo.kt
│   ├── api
│   │   ├── Auth.kt
│   │   ├── Store.kt
│   │   └── TaskDecorators.kt
│   └── bean
│       ├── Trip.kt
│       └── User.kt
├── frontend
│   ├── activities
│   │   ├── AccountInfoActivity.kt
│   │   ├── EmailLoginActivity.kt
│   │   ├── GoogleLoginActivity.kt
│   │   ├── LoginActivity.kt
│   │   ├── MainActivity.kt
│   │   ├── RegisterActivity.kt
│   │   ├── TripDetailsActivity.kt
│   │   └── TripsList.kt
│   ├── adapters
│   │   └── UserAdapter.kt
│   └── fragments
│       ├── DriverFragment.kt
│       └── PassengerFragment.kt
└── global
    └── Constants.kt
```

```
➜ res git:(master) ✗ tree
.
├── drawable
│   ├── ic_account_circle.xml
│   ├── ic_driver_black_24dp.xml
│   ├── ic_exit.xml
│   ├── ic_launcher_background.xml
│   ├── ic_passenger_black_24dp.xml
│   ├── side_nav_bar.xml
│   └── trip_line_background.xml
├── drawable-v24
│   └── ic_launcher_foreground.xml
├── layout
│   ├── activity_account_info.xml
│   ├── activity_email_login.xml
│   ├── activity_google_login.xml
│   ├── activity_login.xml
│   ├── activity_main.xml
│   ├── activity_register.xml
│   ├── activity_trip_details.xml
│   ├── activity_trips_list.xml
│   ├── app_bar_main.xml
│   ├── fragment_driver.xml
│   ├── fragment_passenger.xml
│   ├── line_trip.xml
│   └── nav_header_main.xml
├── menu
│   ├── activity_main_drawer.xml
│   └── navigation.xml
├── mipmap-anydpi-v26
│   ├── ic_launcher.xml
│   └── ic_launcher_round.xml
├── mipmap-hdpi
│   ├── ic_launcher.png
│   ├── ic_launcher_foreground.png
│   └── ic_launcher_round.png
├── mipmap-mdpi
│   ├── ic_launcher.png
│   ├── ic_launcher_foreground.png
│   └── ic_launcher_round.png
├── mipmap-xhdpi
│   ├── ic_launcher.png
│   ├── ic_launcher_foreground.png
│   └── ic_launcher_round.png
├── mipmap-xxhdpi
│   ├── ic_launcher.png
│   ├── ic_launcher_foreground.png
│   └── ic_launcher_round.png
├── mipmap-xxxhdpi
│   ├── ic_launcher.png
│   ├── ic_launcher_foreground.png
│   └── ic_launcher_round.png
└── values
    ├── colors.xml
    ├── dimens.xml
    ├── strings.xml
    └── styles.xml
```

Fig. 13. Application Catalog

Fig. 13. shows our application catalog, as the tree map. In the backend directory, there are UserInfo.kt, api, and bean. The main job of backend is to interact with the database. The UserInfo here is implemented based on the singleton pattern. Because one user can have one and only one id instance. In the api directory, there are Auth.kt, Store.kt, and TaskDecorators.kt. The Auth is mainly for the login authentication. The Store is mainly for the addition, deletion and modification of the database. The TaskDecorators class is implemented based on the decorator pattern, mainly provide information about success or failure of related operations in the Store (database). The bean here is the User collection and the Trip collection object, which we can add to or get from the database.

The frontend contains all the activities, which can be considered as the presenter in the MVP pattern. The activities contain the logic that can control the interface (the layouts in the layout directory) and can communicate with the backend.

In the res directory, the app layouts are in the layout directory. The layouts can be seen as the View in the MVP pattern. Also, the res directory contains other resources such as drawable, mipmap, and values. The drawable mainly implemented all the icons showing in our application. The mipmap includes all sizes of our app's icon. The values folder contains all the text, color, and style displays in our app.

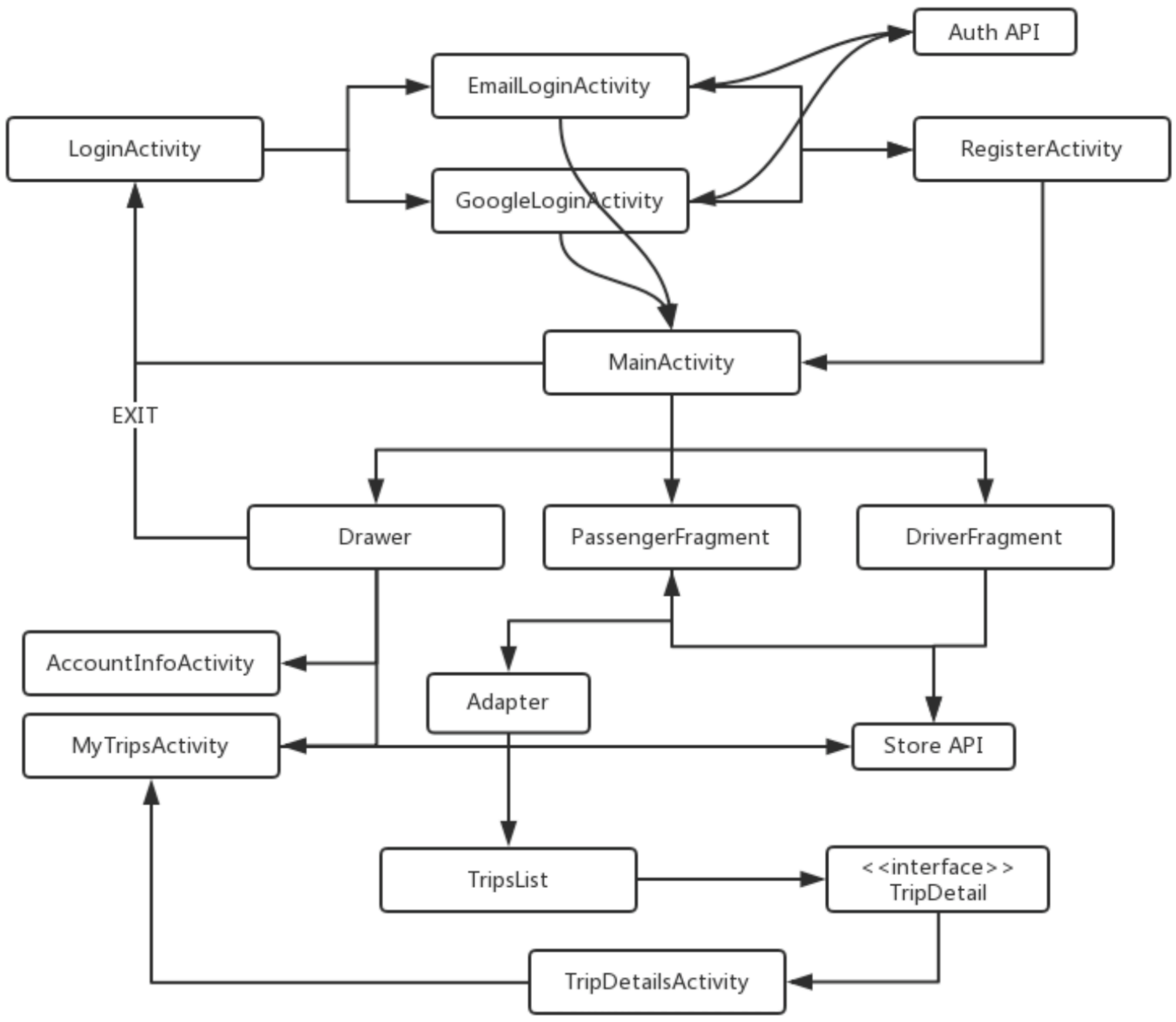


Fig. 14. Class Diagram of Our Project

We want to explain our program in general through the class diagram. When we first open the application, it first starts at the MainActivity. In the MainActivity, the application checks if the user already exists, otherwise it jumps to the LoginActivity.

The LoginActivity provides two login ways, the user can either choose email or google account to log in to the application. The Auth API here can help the application check if the account or email exists. After user logged in, the program checks if this is the new user (first-time login). If this is the new user, the application jumps to the RegisterActivity and user need to input his/her first name and last time to finish the registration. If not, after logged in, the application jumps to the MainActivity.

In MainActivity, there are mainly three activities, the Drawer activity, the PassengerFragment activity, and the DriverFragment activity. User can choose to be a driver or a passenger by click the navigation button at the bottom of the screen. The Drawer activities can show the user's account information, the user's trip information and exit the application. If the user clicks the account button in the drawer interface, the app jumps to the account information activity and shows the user's first name, last name, and the email address by getting related data from the database. Also, the user can set his/her avatar by clicking the upload avatar button and the choose avatar button.

There are some text boxes and one confirmation button in the DriverFragment interface. The logic behind the DriverFragment class is that the DriverFragment activity can collect all the input information from the user and generate a Trip object, then the application uploads the trip object to the Trip collection in the database.

In the Passenger Fragment, the passenger inputs the departure city, the arrival city, the departure date, and the number of people. We implemented a search query in the Store class which can based on the data provided by the passenger to find out if there exist trips or not. If trips information is found by the search query, then the program needs to use the adapter to format the data and display it on the TripsList interface. In our app, we put the driver's name and the specific departure time as one trip's information on the TripsList interface. When the passenger clicks one trip in the TripsList interface, the app jumps to TripDetailsActivity's interface and shows the detailed information such as the departure address, the arrival address, the driver's contact information and the price. After the user clicks the confirm button, the app adds the passenger's information to the database and then both the driver and the passenger can see the trip details in MyTripActivity's interface.

4. Future Improvement

To make our application more user-friendly, we are currently implementing some other details such as user profiles functionalities (change/upload their avatar, have already implemented). We also want to let users know their current trips information by communicating with the databases, but this means that our database needs to be redesigned, which may cause all of activities which communicate with the database need to be rewritten. We are still working on it and considering whether there is a better way to achieve this function.

Participation journal

Zizheng Jiang
Responsible for the overall architecture and structure of the project. Implemented the logic behind the login interface and register interface, which used the third-party login token and **Singleton Pattern**, respectively. Implemented part of the user-interfaces and logic (Activities related to Login, Main, and TripDetails) behind those user-interfaces. Created the database object and deployed the database on the third-party platform. Under the **Client-Server Architecture**, implemented specific database query statements and relevant interfaces are provided for the frontend to implement their own logic. Implemented data transfer between two activities based on the **Data Access Object Pattern**. Solved the animation stuck problem in Drawer interface and implemented the Recycler View based on the **Adapter Pattern**. Responsible for part of the paperwork and the slides.

Zhilun Chang
Based on the **Model-view-presenter Pattern**, implemented the AccountInfoActivity interface (**view**) and AccountInfo activity (**presenter**). Implemented the logic of showing user's information and uploading user's avatar by interacting with the database (**model**). Developed the logic of displaying the account info in MainActivity under the **Event-based Architecture**. Responsible for testing the functional and non-functional properties of the application. Completed text description and slides of the 4 types of architecture styles and non-functional properties.

Jiatong He
Responsible for the frontend of the application, e.g. driver input interface, passenger input interface, available driver list interface, driver information interface and so on, which implemented part of the code under **Event-based Architecture** and **Model-view-presenter Pattern**. Collected pictures and used the Google design library to generate icons in the application to make our app more user-friendly. Integrated the code for layouts and unified the styles for all the interfaces. Responsible for the proposal of the requirements and specification. Responsible for part of the paperwork, e.g. deliverable assignments, proofreading work, and the slides.

Shuting Lian
Designed part of the user interfaces and achieved part of the frontend functionalities of the application, e.g. registration, login, switching account, displaying account info and drawer menu. Adopted **Model-view-presenter Pattern** and **Event-based Architecture** to listen to events from the view and call methods from the **presenter** (activities in Android). Implemented part of the code for presenter to update the view or jump to another interface based on the action on the specific component (button). Responsible for testing part of the code of the application, organizing paperwork and presentation.

Reference:

[1] Difference Between Parcelable and Serializable: https://stackoverflow.com/questions/3323074/android-difference-between-parcelable-and-serializable Last accessed: Dec 01, 2018.
[2] Android Activities Lifecycle: https://developer.android.com/guide/components/activities/activity-lifecycle Last accessed: Dec 01, 2018.