**Thesis**



*Wang Yuxuan*

*Debrecen*

*2022*

**University of Debrecen**

**Faculty of Informatics**

**A ride-share web application based on Java and typescript**

Supervisor Candidate

Dr. Aniko Vagner Wang Yuxuan

*Debrecen*

*2022*

[I . Introduction 5](#_Toc16207)

[1. Introduction to ride share 5](#_Toc1519)

[2. Web application and Web development 5](#_Toc9256)

[3. Introduction to Implementation Technology Stack 6](#_Toc21341)

[II . Backend part 7](#_Toc18315)

[1. PostgresSQL 7](#_Toc18251)

[i. Introduction 7](#_Toc19999)

[ii. Database in carpooling 8](#_Toc5789)

[2. Springboot 8](#_Toc6309)

[i. Introduction to spring boot 8](#_Toc26807)

[ii. Spring boot beans and life circle 9](#_Toc31735)

[iii. Layers in spring boot 10](#_Toc26117)

[iv. Building Entity In Spring Boot 11](#_Toc29150)

[a. Base Entity 11](#_Toc4087)

[b. Attributes 12](#_Toc20913)

[c. Relationships 13](#_Toc21923)

[III． Map Entities into database using Liquibase 14](#_Toc22238)

[a． Building table using liquibase changelog 14](#_Toc12968)

[b． Dealing with relationships in liquibase 15](#_Toc11467)

[c． Insert data into database using Liquibase 15](#_Toc21005)

[ii． Repository layer 16](#_Toc12979)

[iii． Service Layer 17](#_Toc15662)

[iv． Controller Layer 18](#_Toc32613)

[v． DTO(Data Transfer Objects) 19](#_Toc7925)

[vi． Mapping with MapStruct 20](#_Toc11287)

[１． Introduction to Docker 22](#_Toc9711)

[i． What is Docker 22](#_Toc26276)

[a． Why are we using docker 22](#_Toc9214)

[ii． How docker works 23](#_Toc3825)

[a． Images and containers 23](#_Toc16575)

[b． How images are made 24](#_Toc84)

[c． Layer Caching 25](#_Toc31694)

[iii． Volumes in Docker 25](#_Toc28447)

[iv． Docker Compose 26](#_Toc26266)

[v． Dockerize our spingboot application 27](#_Toc17361)

[II． Frontend part 29](#_Toc31077)

[１． Connect Frontend with Backend 29](#_Toc26739)

[２． TypeScript 30](#_Toc17817)

[３． React 31](#_Toc25127)

[i． Introduction 31](#_Toc6061)

[a． multi-page applications and single page applications 32](#_Toc8187)

[b． React for SPA 33](#_Toc27935)

[ii． Our react structures 34](#_Toc29048)

[a． Components 35](#_Toc1874)

[b． Generated 36](#_Toc27270)

[c． Hooks 36](#_Toc27219)

[iii． Store layer with Mobx 36](#_Toc26526)

[iv． Service 38](#_Toc20179)

[a． Async and Await 39](#_Toc15459)

[v． Route Authentication 40](#_Toc2260)

[vi． Material-UI 41](#_Toc10258)

[a． Introduction 41](#_Toc12506)

[b． Text field 41](#_Toc18028)

[c． Grid System 43](#_Toc2984)

[d． Table 45](#_Toc10508)

[III． Conclusion 49](#_Toc19657)

[IV． Bibliography 50](#_Toc25668)

# Introduction

#### Introduction to ride share

Ride-share (Carpooling) is a brand new type of travelling ,the main idea of it is to share car journeys in order to reduce each person’s cost for the trip .drivers will post their ride on platform and passengers will join part of or the whole journey based on their needs , and share the travelling cost . Drivers and passengers will contact each other to make an appointment for pick-up places and may negotiate special need (such as large luggage) and price . It is more eco-friendly as well since it’s a good way to use up the full seating capacity of a car , which will be left unused if only the driver is using the car. Authorities often encourage ride-share , as it can reduce air pollution , traffic congestion on the roads .[1]

In order to post or share the carpooling information , we need such a platform . There are quite many types of platforms for it , website , carpooling agency , pick-up point , carpooling groups and so on . Nowadays , the we application is becoming more and more popular. And building a ride-share web application will suit our needs for such a platform .

#### Web application and Web development

A web application (or web app) is a software that runs on a web server .It is more portable , light than desktop applications. You can open it any where on any platform as long as you have a browser . Apart from that , as web apps are cloud based , it is more easier to sync all the data and info between all the devices .

There are several reason that why I am in favor of web application and choose to develop a web based ride-share application :

* **No need for installation and Cross-Platform Availability**

Since the application is web based , it does not require user to download any installation pack . On any device ,everything we need to access a web application is just a web browser , internet and a URL , since all the components are deployed at the sever side , no matter what platform we are using , we can easily access to it .

* **Automatic updates**

For desktop or mobile applications , we have to manually download and install updates , even though some programs can automatically downloads , we still need to approve it . But for web application we can do our update on sever side and the user can always access to the latest stable version without manually doing anything . Which is a huge improvement for user experience.

* **Light on Computer Resources**

Web applications takes significantly less resources on our computer , since most of the service are provide from the sever side , it takes significantly less memory and processor compared with desktop or mobile applications.[2]

#### Introduction to Implementation Technology Stack

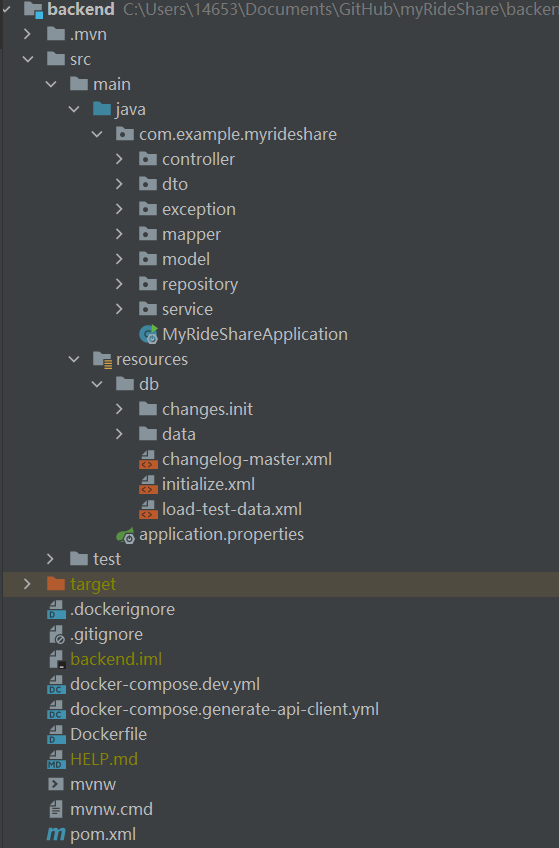
For web application , mostly we have frontend and backend . The backend is more about dealing with database , handling data ,server , and implementing main logic and then provide the apis to frontend , while frontend is more about user interface and interaction , send or request data from backend .

As for our Technology Stack to implement such a carpooling website , we are going to use PostgresSQL as our database ,Java as our backend language and spring boot as our backend framework . We will use typescript and react as our frontend language and framework . We will connect both sides using open api generator and deploy the whole application into docker .

//Add the reason why you choose them here .

# Backend part

Structure in backend



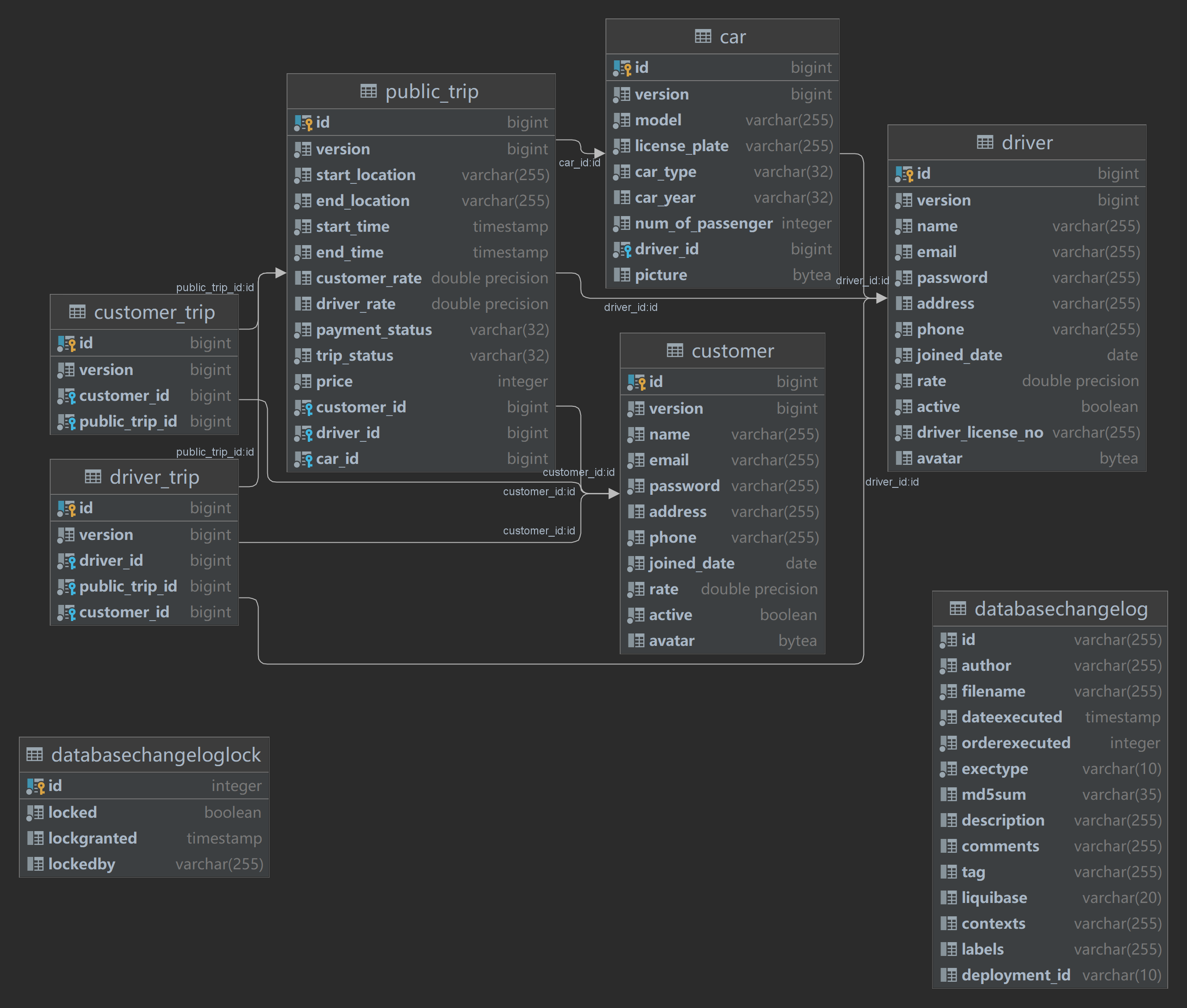
#### PostgresSQL

###### Introduction

PostgresSQL is a relational database management system (RDBMS) , which means it stores data in the form of tables. It was built to be feature-rich, extendable and standards-compliant. In the past, Postgres performance was more balanced - reads were generally slower than MySQL, but it was capable of writing large amounts of data more efficiently, and it handles concurrency better. [3]

###### Database in carpooling

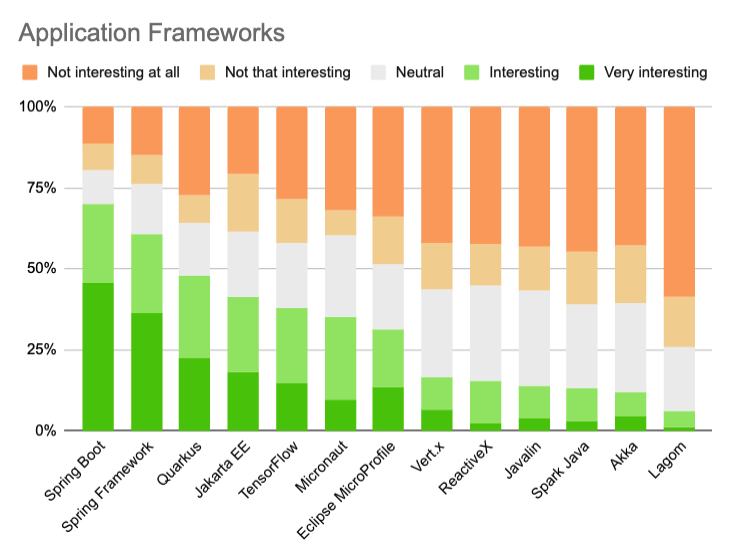
For our carpooling website , We will store data into database . The database design is shown below :



#### Springboot

###### Introduction to spring boot

Spring boot is one of the most famous and popular frame work ,it is open source, microservice-based , and make build backend or even full stack development using java or kotlin simple and easy , It gives us everything we need in order to build applications .



Spring boot is dominating in java[4]

###### Spring boot beans and life circle

**What is spring boot bean**

By official document :

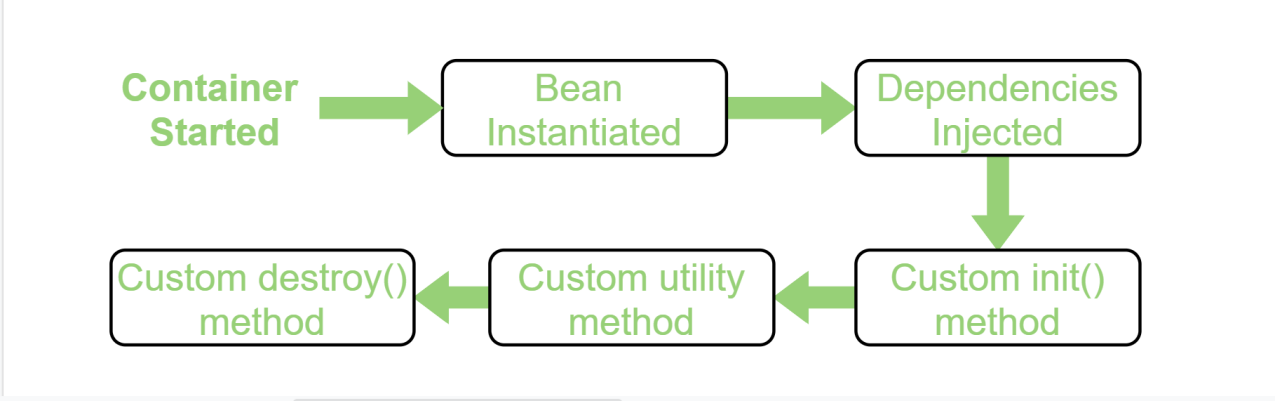
**In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.**[5]

In short , Spring boot bean is still a Java object , but managed by Spring IOC container . The bean life cycle refers to when and how the bean is instantiated, what action it performs until it lives, and when and how it is destroyed. Bean life cycle is managed by the spring container. When we run the program then, first of all, the spring container gets started. After that, the container creates the instance of a bean as per the request, and then dependencies are injected. And finally, the bean is destroyed when the spring container is closed.[6]

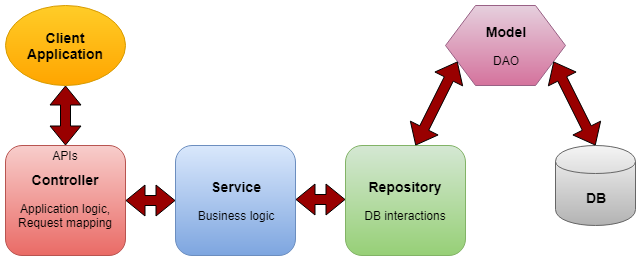
If we want to execute some code on the bean instantiation and just after closing the spring container, then we can write that code inside the custom init() method and the destroy() method.

By writing code in custom init() and destroy() method , we can execute our custom on the bean instantiation and destruction .

The following image shows the process flow of the bean life cycle.



###### Layers in spring boot



**Controller layer** : Controller layer is also called api layer , since it is providing rest apis for frontend . It handles HTTP requests, mapping them to particular functions and passing the user input to service layer to apply the business logic.

**Service layer :** Service layer is in the middle of controller and repository layer, which performs the business logic and validation logic.It manipulates the data that we get from repository layer before sending it to controllers. And also form the user input data before we interact with the database when calling repository layer .

**Repository layer :** It interacts with the database CRUD operations (create, retrieve, update, delete) via the DAOs(data access objects).

**Model :** It is the simple POJO classes which is acting as the DTO(Interact with application level data transfer) or DAO(Interaction with database operations) [7]

**Database Layer**: In the database layer, CRUD operations are performed and this is the layer where we store our data .

In the next session , we will start to build our backend application from bottom layer to top.

###### Building Entity In Spring Boot

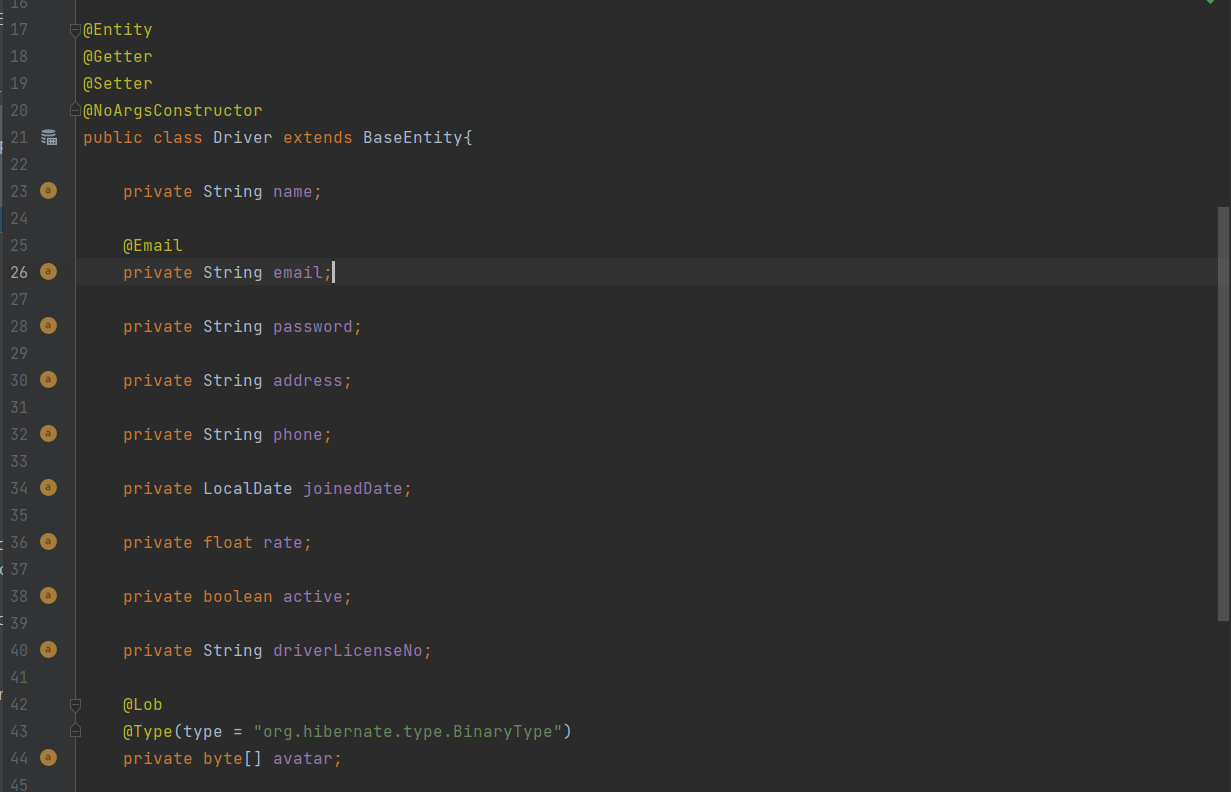
After we have our database design , we need to map it to spring boot using java in order to manipulate data directly using java , where we need to build our entity classes.

**Base Entity**



For every entity they do needs something in common : an id and version . Id is the primary key for each table or entity , version is for concurrency control . Since all the entities share some attributes , it would be wise to put them into a abstract class and let the other entity classes to extend it . So that they will have a auto increased id for their primary key .

After the preparation , we are ready to map our entities . Let’s take our Driver entity as an example :



Firstly we need to extend our Base Entity class and mark **@Entity** annotation to tell spring boot it’s an database entity .

**Attributes**

For attributes we have :

name : driver’s name ,

email with **@Email** annotation in order to validate the string is a valid email form .

password , address ,

driver’s phone for contact ,

joinedDate : Driver’s join date in the community ,

Rate :driver’s rate from customer ,

active : when driver registered himself with an email , the active will be set to false since the email has not confirmed the email yet ,and driver cannot login when active if false . it will be set to true when the email is confirmed,

driverLicenseNo : confirm the driver is able to drive,

avatar : drivers can upload their avatar to the database .Note that the pictures in database in an byte array .

**Relationships**

After the attributes we have to handle our relationships between tables , here is an example in driver class that how we can handle it in spring boot :

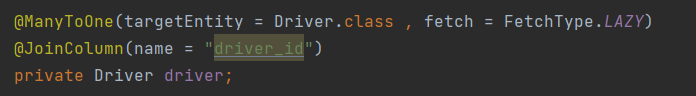
For driver table we have two relationships :

1. One driver can have multiple cars
2. One driver can publish multiple trips

They are both one to many relationship , but how do we handle it in springboot ?



As in the picture , we annotate them with **@OneToMany** annotation ,since it’s one to many relationship, we represent them as a list here. As we actually store driver id in Car and Public trip table , in those classes , we need to do the following :



We annotate driver entity with **@ManyToOne** and specify the target entity in Car Entity class and PublicTrip entity class respectively.

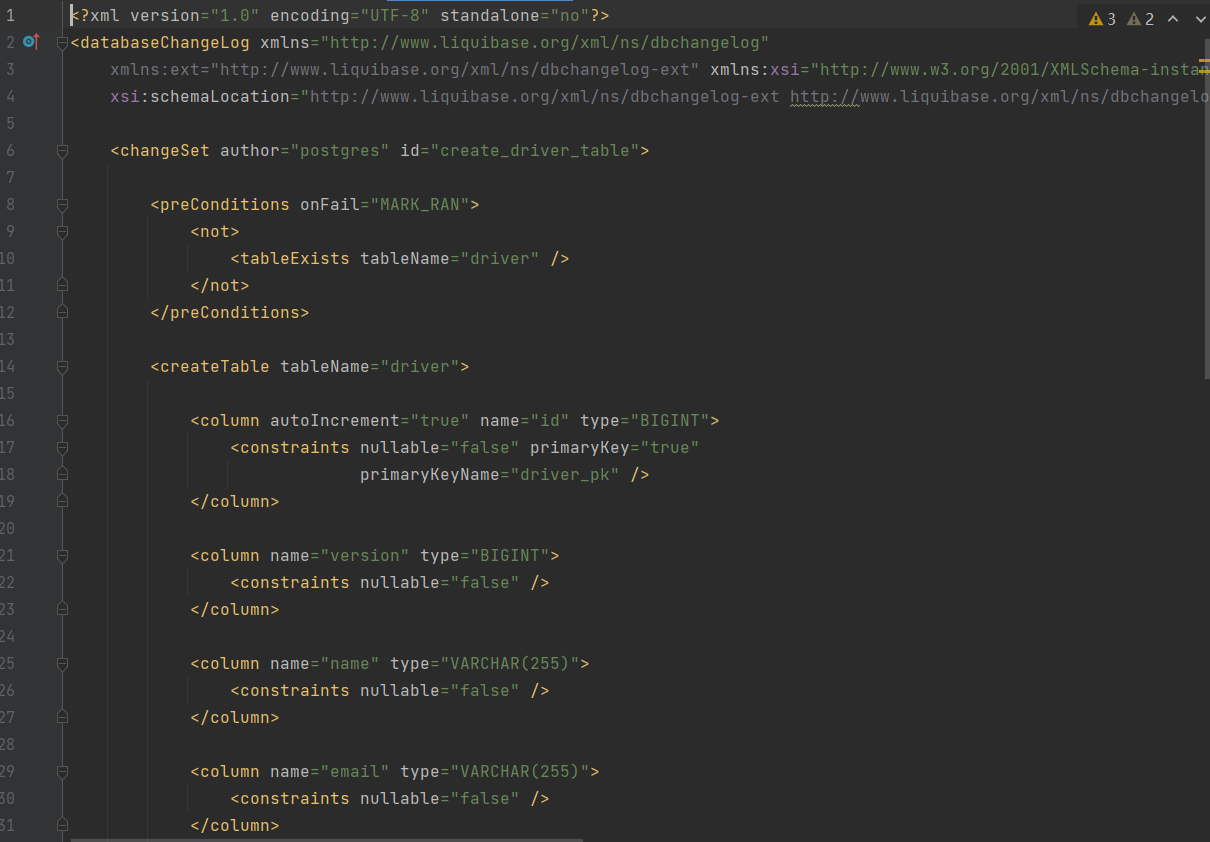
###### Map Entities into database using Liquibase

For now we have our relational model and entities classes ready , but our database is till empty , neither tables nor data are there. We need to map all our designs into database tables . We can surly use SQL statements to build such tables , but each time we flush our database we need to add everything manually . In order to work it around , liquibase is coming to rescue .

Liquibase is an open-source solution for managing revisions of your database schema scripts. It works across various types of databases and supports various file formats for defining the DB structure. The feature that is probably most attractive in Liquibase is its ability to roll changes back and forward from a specific point — saving you from needing to know what was the last change/script you ran on a specific DB instance. [8]

**Building table using liquibase changelog**

We can use liquibase changelog to create our tables , it can be SQL ,XML , JSON or YAML format , for this project we will use XML format :



This is how change logs look like . Inside our change log , we have our change set , it can be multiple change sets , but for each change set we need to specify a id . We use XML tags to “represent ” sql statements , for which liquibase will compile it down to sql statements .

We use **<createTable>** tag to create our table , **<column>** tag to create attributes for our table , we can also add **<constrains>** tag to add some constrains to the column , such as not null , unique etc .

**Dealing with relationships in liquibase**

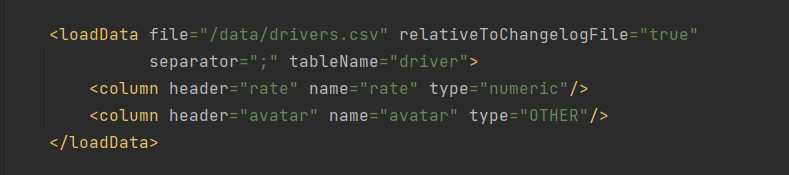
In RDBMS , when dealing with one to many relationship , we get the id from “one” side and put it in “many” side . The same in Liquibase . For one to many relationship between driver and public\_trip table , we will state in liquibase as below :



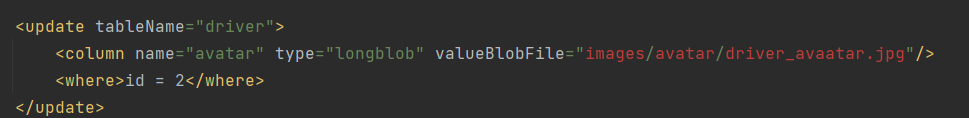
All we need is to specify ForeignKeyName and reference label . In reference , it should be the table name here they are table driver and car .

**Insert data into database using Liquibase**

After creating the database , we can insert some data into database as test data . When inserting data , we will use **<loadData>** tag in a separate change set :



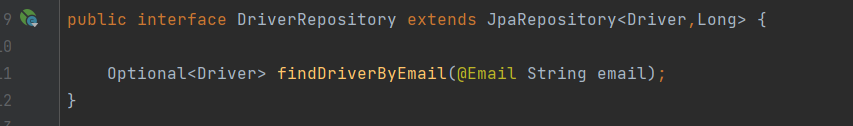
We can put our data into csv files and put the link inside the tag , then Liquibase will do the magic . Note that when inserting pictures into database , using csv is a bit hard to handle that , we can use update statement instead .



After finishing our model layer and mapping everything into database , we are finally be able to store our data . Our next step will be building our repository layer in order to have **CRUD** functionality .

###### Repository layer

The traditional DAO layer usually consists of a lot of boilerplate code that can and should be simplified. Spring data significantly simplified the process and makes it possible to remove the DAO implementations entirely . In order to use the model spring jpa provides us , we need to extend our class with **JPA specific Repository interface**, **JpaRepository**. By extending the interface, we get the most relevant CRUD methods for standard data access available in a standard DAO. [9]



When extending , we need to provide two parameters , first is the entity class that you are having CRUD operation with , the second is the id type , we are using Long as our id , so we will put Long there.

By extending the interface, we’ve got the most useful CURD functionality here . Interface JpaRepository provides useful method including: findAll , getById deleteById ,save etc. With all these methods ,we can build our CURD functionality easily , without writing SQL statements ourselves .

**Custom Access Method and Queries**

By implementing one of the Repository interfaces, we already have some basic CRUD methods (and queries) defined and implemented in JpaRepository . When we need more complex and custom query , we can simply add our custom query by :

1.simply define a new method in the interface , as long as it fits JPA grammar, Spring data will take over and implement it for us to use .

2 .provide the actual JPQL query by using the **@Query** annotation , we can also write original SQL by adding native = true to the annotation.

3 .use the more advanced Specification and Querydsl support in Spring Data.

###### Service Layer

In service layer we will call repository layer and use the method that it provides to handle data . As service layer serves as a transactional barrier and houses both application and infrastructure services. Furthermore, the public API of the service layer is provided by the application services. They often serve as a transaction boundary and are in charge of authorizing transactions. Infrastructure services provide the “plumbing code” that connects to external tools including file systems, databases, and email servers. These approaches are often used by several application services. **[10]**

Let’s take CustomerService for example :



For service class we need to annotate it with **@Service** annotation , it indicate that current class belong to service layer , It is a specialization of **@Component** Annotation as well .

Inside service layer we will handle transactions with our repository layer and provide the data to controller layer . Here we have the basic CURD functionalities to create , get , delete , update a customer and get all customer information . The data filtering and preprocessing is also inside service layer , we can use stream to filter our data and only send the necessary data to controller , then to the frontend .

###### Controller Layer

A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer and was created by computer scientist Roy Fielding . [11]

Controller layer is mainly responsible for processing incoming REST API requests , process the data and pass them down to the service layer . And returning the response to the frontend . Let’s take driver Controller as an example :



The controller class needs to be annotated with **@Controller** annotation or **@RestController** . The **@RequestMapping** annotation is used to map web requests to specific handler class or methods . All the method in driver class will be after “api/driver” path . we can annotate the methods depends on the type of request sent from frontend , the most commonly used are :**@GetMapping** , **@PutMapping** , **@PostMapping** ,**@DeleteMapping** , they will map HTTP Get /Put /Post /Delete requests onto corresponding handler methods respectively.

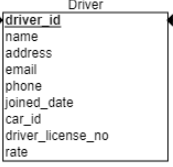
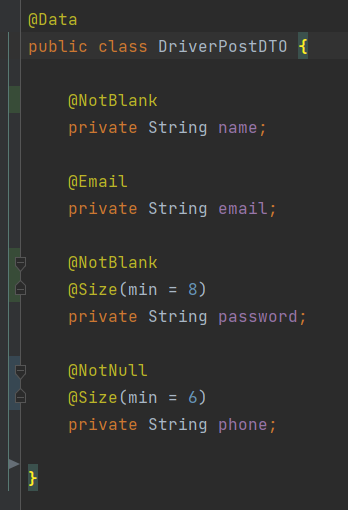
Path variable is the variable that you can pass within the url , usually quite useful when you want to access to certain item in a list based on id . When annotated with **@PathVariable** , the template variable in url will be mapped and set as method parameters .

###### DTO(Data Transfer Objects)

Here we notice that the objects that we returning to the frontend side are actually DTOs instead of entities. A **DTO**(data transfer object) is an object that is used to encapsulate data before sending to the frontend .

When the frontend is requesting or posting data , it won’t always be all the data in an entity , it can be some necessary attributes in an entity or multi entities , since now return the whole entity or all the required entity to frontend is not wise at all , that’s why we need to encapsulate them into a DTO , no matter request or response .It can also be useful when a method takes more than five parameters , it’s better to collect them into DTO .

Let’s take DriverPostDTO for an example :



DriverPostDTO is used for driver registration . There are quite many attributes in driver table , but we surely don not want to provide them all when registering , maybe just some necessary info , and we will make our profile better later . That when we need our DTO , since we only need some necessary attributes , we can put them into the post DTO , to tell frontend what to send .

**Spring Data validation in DTO**

Here we annotated the field with some data validation annotations .

**@NotBlank** : indicates that name must not be null and must contain at least one non-whitespace character

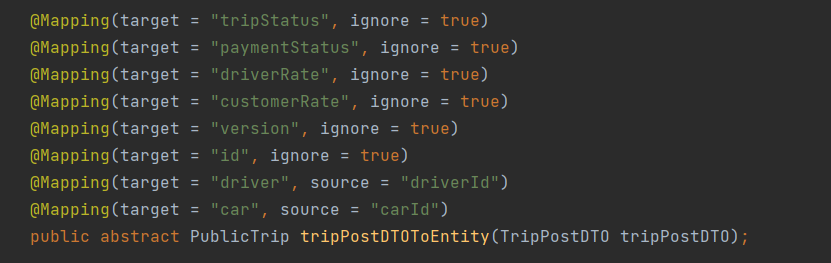
**@NotNull** : indicates that the element should not be null

**@Size** : indicates that the password should have at least 8 characters and phone number should have at least 6 characters.

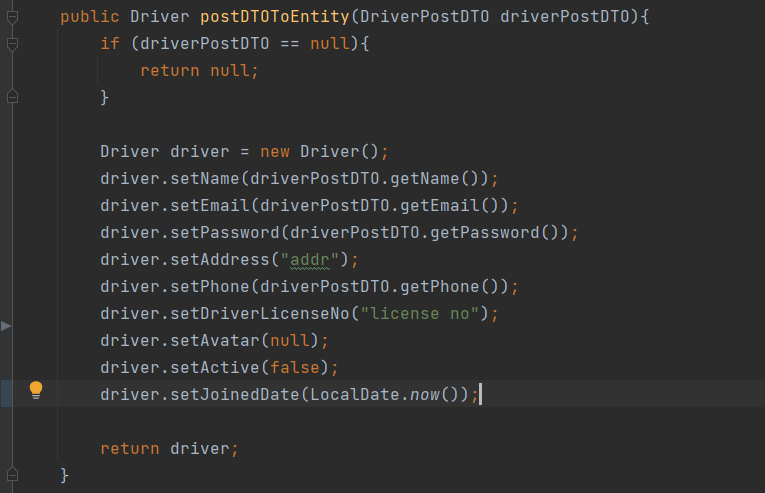
###### Mapping with MapStruct

After we have our DTO , another problem is mapping between dto and entity . Mapstruct is a good tool for it .MapStruct is a code generator that greatly simplifies the implementation of mappings between Java bean types based on a convention over configuration approach.The generated mapping code uses plain method invocations and thus is fast, type-safe and easy to understand. [12]

In order to map between dto and entity , we need a mapper .In mapStruct , it can be an abstract method in abstract class or Interface, and the mapping code will be auto generated .When there’s naming dismatch and mapsturct cannot find corresponding variable , we need to annotate the method with **@Mapping** and tell mapstruct which variable it is by adding it in source tag , or simply add ingore = true to ignore it .



If we would like to add some customize logic while mapping , we can simply implement the mapping method . For postDTO we set the corresponding field to entity , for some field is auto generated : The joined date should be decide by local date , the active field should be false since driver just registered and certainly haven’t confirm his or her email yet . We will set the active back as soon as the email is confirmed .



After mapping , we can get our entity object and ready to save it into database .After we have our controllers , DTOs and relevent logic , our backend is almost ready for requests and responses from frontend .

#### Introduction to Docker

###### What is Docker

Docker is an open source containerization platform. It enables developers to package applications into containers—standardized executable components combining application source code with the operating system (OS) libraries and dependencies required to run that code in any environment. Containers simplify delivery of distributed applications, and have become increasingly popular as organizations shift to cloud-native development and hybrid multicloud environments.

Developers can create containers without Docker, but the platform makes it easier, simpler, and safer to build, deploy and manage containers. Docker is essentially a toolkit that enables developers to build, deploy, run, update, and stop containers using simple commands and work-saving automation through a single API.[13]

Why are we using docker

Docker can make developing application either on your own or team much easier to manage . Let’s say we need a specific Java version (JDK17.0.1) and postgres version (14.1)for our current application , in order to let my application to run on the other computers properly , the other computers needs to have the same Java and postgres version installed , which made a huge work just for running the application . And different application may require different version as well . That is why we need docker and containers .

Imagine docker container as a box , it contains everything our application needs to run . Source code , dependencies , version and runtime environments are all in it . So this application can run on our computer in isolation , independent from all other processes on our computer , which makes it easier for me and any other people who want to run the application on their computer . It also helps to deploy our product to sever , we don’t need to configure that much on our sever since all the configuration is set up in docker container already .

**Docker vs virtual machines**

Since virtual machines can solve the similar problem , why should we use docker instead of virtual machines ? The reason is that they are having different mechanisms .

Virtual machines has it’s own full operating system , which mean typically slower to start and run , while docker containers share the host’s operating system and runs or starts quicker . Docker containers are much more lightweight and faster in this occasion . But for some occasion you will prefer virtual machines instead of docker , both of them have their advantages.

###### How docker works

**Images and containers**

Docker images are like the blue prints of the container . They have the things store in them , including :

**Runtime environment** : versions like JDK 17 and postgres 14.1 and so on

**Application code** : the code itself

**Dependencies** : like Maven dependencies

**Extra Configuration** : environment variables and so on

**Commands** : command to run the app or make it to work

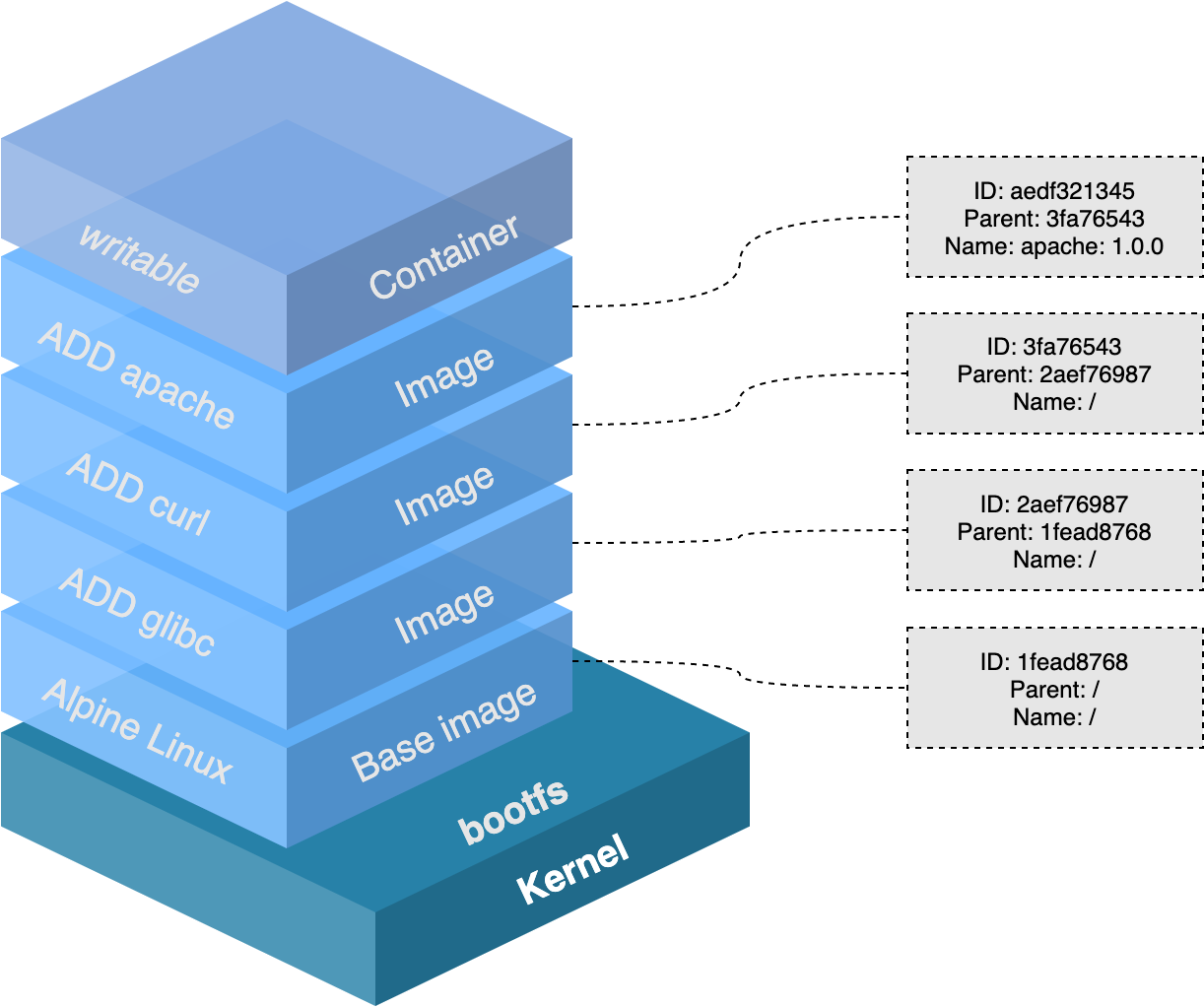
Images have them store inside but now running them . They are a independent file system from your computer and read-only , which means once an image is created it cannot be changed . If you need to apply some new changes you need to create a new brand new image .

Containers are runnable instances of images , it is a process exactly outlined as the image we created . Container is also an independent process , meaning they run independently from any other processes in our computer . The whole thing is like a box , our application is running in it isolated , packaged with everything our application need to run with .

Since everything is isolated and independent , we can just share our image if the others would like to run the process on their computer . They don’t have to set up the environment or have specific Java version installed , since everything is stored in our blueprint --- image .And the application will run exactly the same way on my computer .

**How images are made**

Docker images are made up from different layers . where each layer add something else to the image incrementally , the order of the images does mater .



Docker images[14]

In image we start with our base image or also as known as parent image , which is the first layer in the image . It describes the operating system and runtime environment that we want . And we build our image by adding other layers on it such as our source code , install dependencies and so on . Usually we can get our pre-made first layer from docker hub .

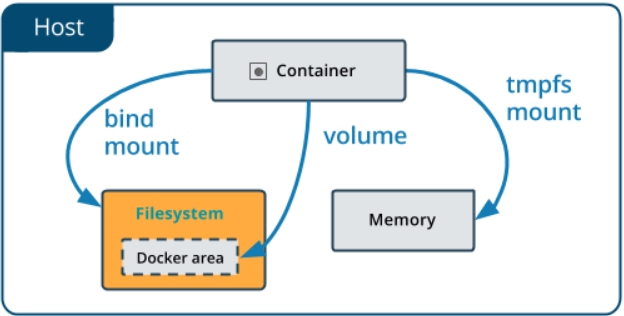
**Layer Caching**

Since image is build by layers being stacked on top of each other , each time we adding layers we are essentially changing the image , and we are giving docker some work to do , to add something to the image when we try to build it , and it takes some time .

Every time we change something , like our source code , we need to build a new image to pick it up , since our old image is based on old code and read-only . In order to rebuild the image docker is going rebuild by adding those layers in turn , which might be quite time-consuming when building the layer from base layer . The actual is , docker will caches our image layers when after building it , which ends up with a cached version image with every layers . Every time docker tries to build an image and work through the layers , it will look into the caches and try to find a image that it can use for the new image that we are creating in order to reduce the workload and building time .

###### Volumes in Docker

Since our image is read-only and our container is based on our image . Imagine such a case , we made some code changes in our app , and we would like to pick up this change . We can stop our container and rerun it , but it does not work , since the container is based on our image and our image does not change until we build a new one . So in this case we have to build a new image and a new container based on it to pick up our change , even we have layer caching to save our time but still quite a complex process . But there is indeed a solution for it , that is when volumes come to play .



Docker volume[15]

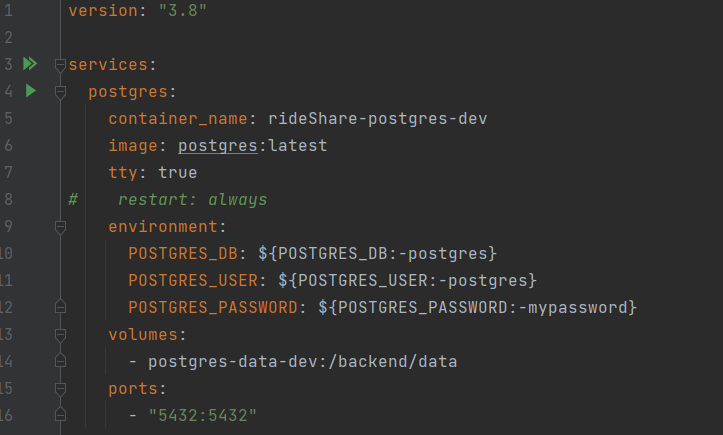
Volumes is a feature of docker that allow us to specify folders on our computer that are available to run containers , and we can map those folder to specific folders inside the docker container , so that if something changed in our computer in that folder , the change can be reflected in the folder and mapped to container . With volumes we can pick up our code changes easily.

Volumes provides us a way that we can make changes and preview them without rebuilding the images all of the time . But one thing is important , the image will not change . Volume is just mapping the changes into container and our image remains as it was . If we want to share the latest changes to the others we have to rebuild the image . But it is quite useful when we are building and testing our application .

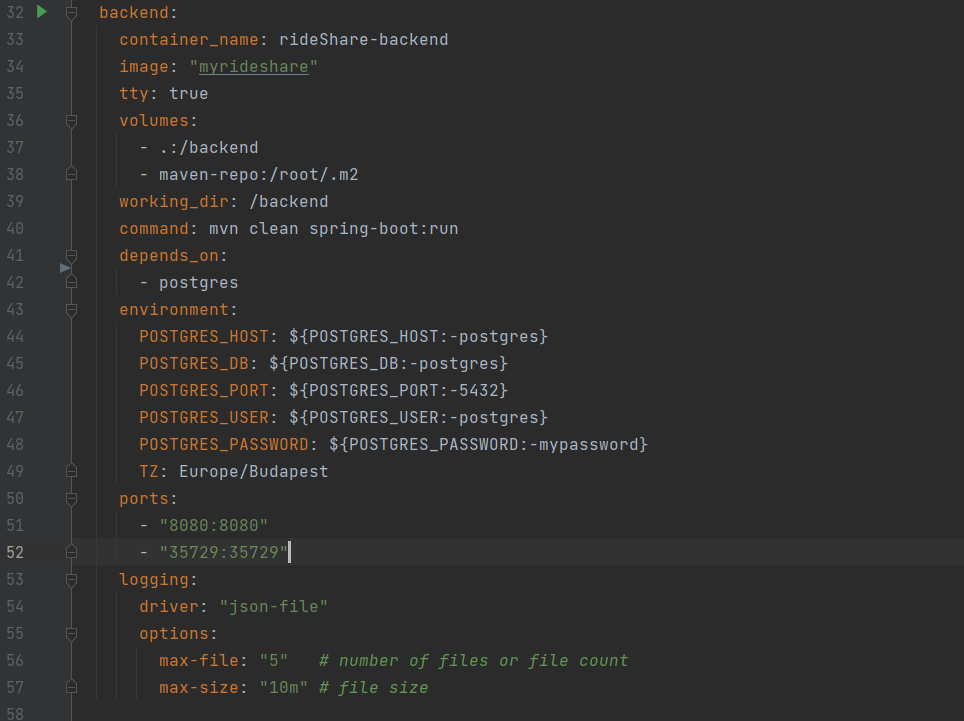
###### Docker Compose

So far we have discussed the images , volumes and containers . Every time we would like to dockerize something it would be quite a long process , with many lines of command and we need to type them in terminal one by one . And sometimes we may have multiple project and we would like to run those containers at once , since one may depend on another , like our frontend is depend on backend and backend is depend on database . We can run those containers one by one by typing the commands , but it can be messy and complicated . That is why we need to introduce a easier way to manage our containers which is something called docker compose .

Docker compose is a tool which give us a way to make a single docker compose file that contains all the container configuration and dependencies . Let’s take the database as example :



###### Dockerize our spingboot application



Inside docker compose file , we specify :

container\_name : the container name .

image : the base image that we need to use .

volumes: The list of volumes that we need , use relative path .

working\_dir : specify the working directory in our docker container.

command : the command that we need to run our app properly

depends\_on : indicates that our backend container depends on postgres container

environment : the environment variables , here is mostly the database configure and authrization

ports : port mapping , we map 8080 port in docker container to port 8080 in our machine .

logging : logging configures

After configured everything in our docker compose file , we just need to simply type :



To run our application , which is much simpler than creating images volumes and containers step by step .

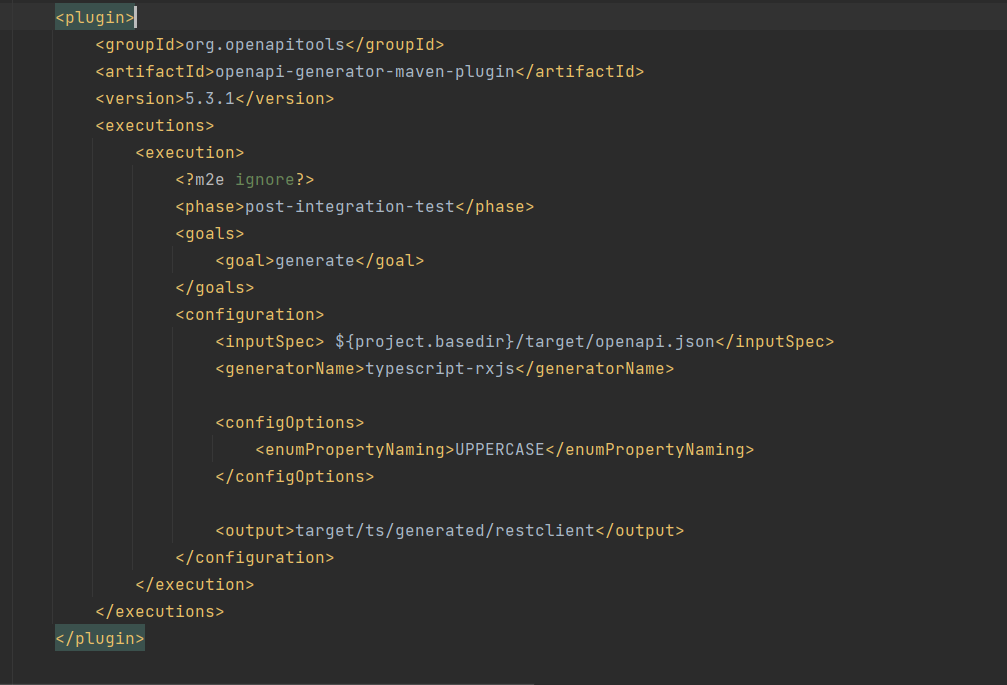
# Frontend part

#### Connect Frontend with Backend

When we have our backend ready and some frontend features , we are having a problem to send or get data from backend . The backend provides api for frontend and usually they are in the controller , but how could we connect two sides ? Surely we can make HTTP request all the time by our own , but it better and easier to invoke if we have our dtos and restAPIs on our frontend side . That’s when openAPI generator come to play .

OpenAPI Generator is a tool designed to create API client libraries, server stubs, configurations, and documentation from OpenAPI 2.0 and 3.x documents. OpenAPI Generator focuses on ease of use; it positions itself as being a tool for reducing the burden on new development and technologies through the integration and leveraging of OpenAPI documents. **[16]**

To generate the apis , first we need our Openapi plugin in our pom.xml



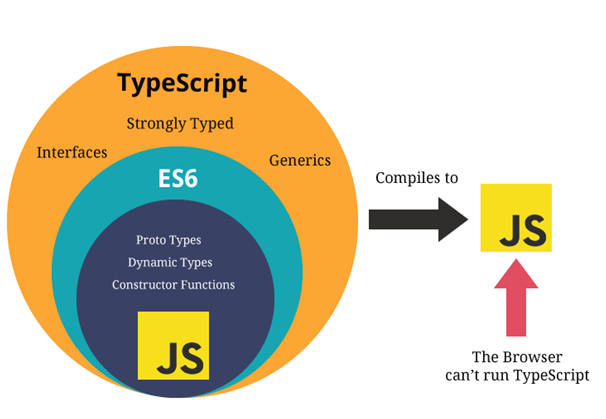
After dockerizing our api generator , we can generate our api by typing the following line in terminal .



Then we will have have our api and dto generated and ready to use on our frontend .

#### TypeScript

For our application we will use typescript as frontend language. Typescript is a super set of JavaScript and can be used as an alternative to JavaScript .It extends JavaScript with new features and syntax it can do everything JavaScript can do .Typescript adds additional syntax to JavaScript to support a tighter integration with our editor , which helps a lot to catch errors early in our code editor. **[17]**



Typescript and JavaScript[18]

Since typescript is not the same as JavaScript and browsers understand JavaScript instead of typescript , we have to compile it down to JavaScript in order to let browsers to understand it . But don not worry about it , it is quite easy to compile and won’t take long . This makes typescript runs everywhere JavaScript runs.

Typescript allows us to use strict types . For example we declare a variable as a number , then we cannot change the type later or assign a string to it , which can be done in JavaScript . JavaScript uses dynamic types ,we can change variable types at any point , but that may lead to more potential errors . The strict types in typescript makes our code becomes easier to debug .

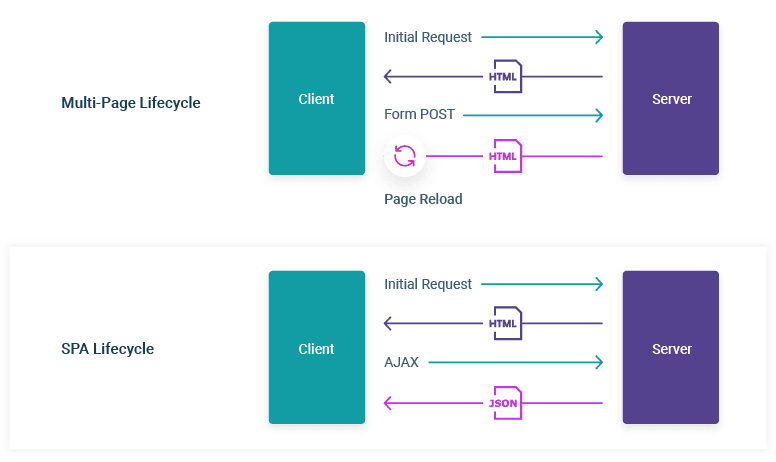
Typescript not only includes modern JavaScript features like arrow functions ,let , const , but also have many more features that JavaScript do not have : interfaces , generic , enums ,tuples etc . These features will helps us a lot in web developing .

#### React

###### Introduction

React is a JavaScript library that we can use to create interactive websites and it allow us to easily create single page applications .

**multi-page applications and single page applications**



**Multi Page Application**

A Multi-page Application is a web application consisting of a large number of pages completely refreshed every time when data changes on them. Whenever there is a data change , the browser will have to send a new request to the server , get back the data and display the new page .

This is a classic approach to web app development, programmers need to do great effort on performance and speed in order to improve user experience .Examples are Amazon eBay and blogs, forums, directories, online publishing websites are more commonly to be multi-page web applications.

**Single Page Applications**

For single page applications (short for spa ) , the sever only needs to send a single html page to the browser to run fully . After we interact with the web page , only the necessary component will be injected .It is loaded from one page, and all user interaction with this service is carried out, using one screen (page).

On demand, a single-page app reloads only the when necessary. In the case of a multi-page app, the entire web page content is refreshed when there is a data change. When the user launches a single page applications, the server loads the entire page. Later on,when responding to the request, and user interaction, only the necessary data is transferred in the form of JSON files .

Examples of single-page applications include Twitter, Gmail, Evernote, Pinterest, and many other web apps that use the technology of a single-page application to provide flexible and scalable user experiences. [19]

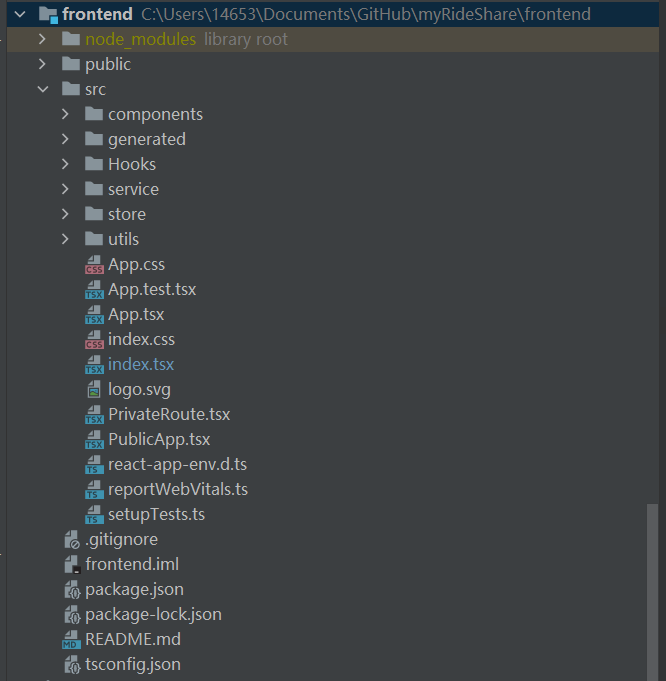
**React for SPA**

Reacts takes over and managing whole website including any kinds of data and user interactivity such as click event and routing from page to page . Instead of sending a new request to the sever when user navigate from page to page , react will change the content of browser depend on the url that user clicked and inject into the page . By contrast , in the treditional way , for every link the user click the browser is going to send a request to the sever for a new html page . This makes react applications usually load really quickly and results in a very speedy user experience .

To create a new react app , we simply type :



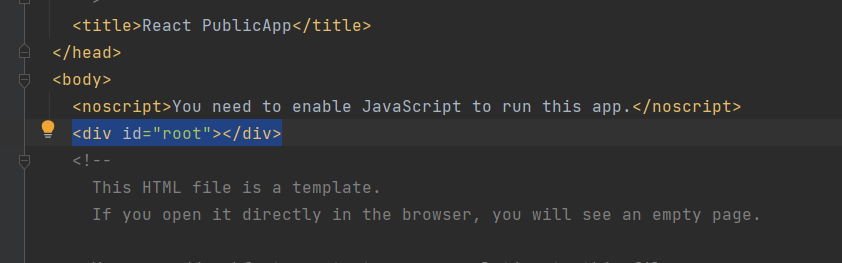
###### Our react structures



Node\_modules : this is where all our dependencies and react library lives ,

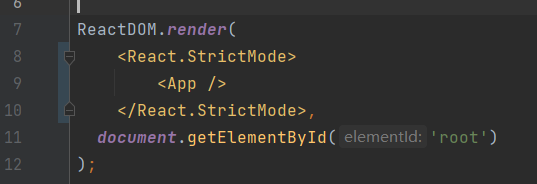
Public folder : Including all the public files to browser .

Index.html : inside public folder we have this index.html file , it is the one html file that is served to browser and all of our react code will be injected into this div with id of root .



Src folder : source folder , all of our react component and logic that we write are going to live inside there .

Index.js : this file is responsible for taking all of our react components and mount them to the dom . In our index.js file we are rendering our App component to the dom .



**Components**

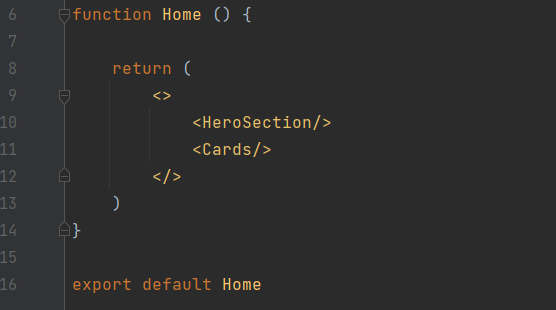
Components is the uppermost layer , usually return html elements and tell react what should be rendered into dom . Components is what we see in the web page . In React, we mainly have two types of components:

**Functional Components :**

Functional components are simply Typescript functions. We can create a functional component in React by either classic function declaration or arrow functions . They can also have parameters as input .

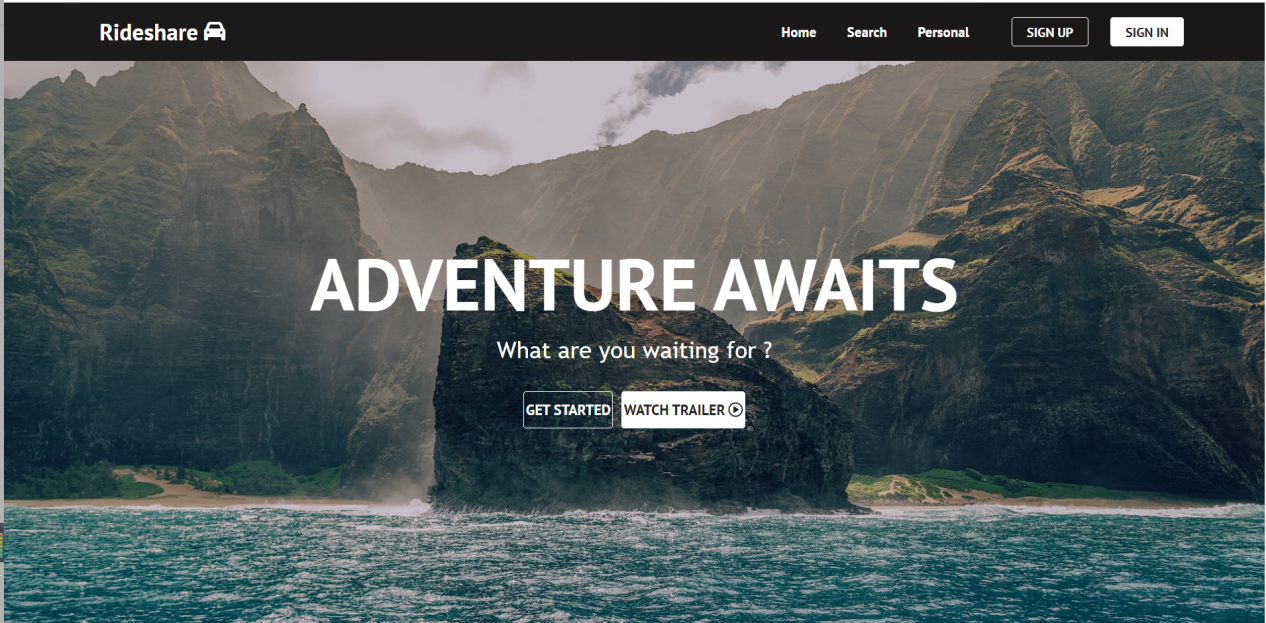
**Class Components :**

The functional components are not aware of the other components in the program whereas the class components can work with each other. We can pass data from one class component to other class components. **[20]** .



Home component in Functional and class type

These two types of components are equivalent , we will use more function component since it is simpler and more convenient.



Our home page

**Generated**

In generated layer we have the rest apis and DTO that generated by open api generator , the DTOs represent the backend status .

**Hooks**

In hook folders we have our custom hooks . Note that custom hooks should starts with “use” .

###### Store layer with Mobx

Store layer is in the middle of components and service , it can get data from service and send to components , then display to the users after everything on mount or vice versa .

Let’s take trip result store as an example , it is used when we perform a trip search .



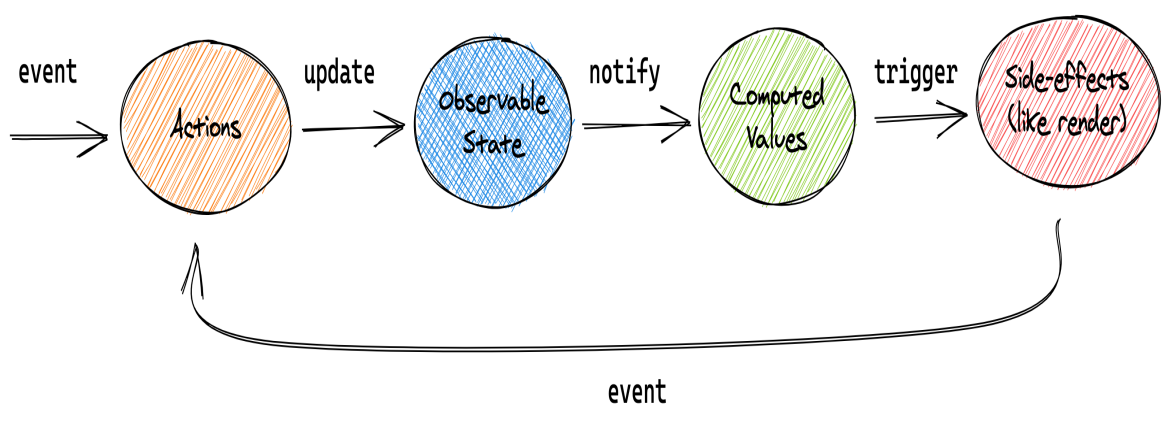
When we perform a trip search , we will send a get request to backend , and if everything goes right , we will get back the data . But the problem is where would we store it .If we inject it directly to component it would may duplicate when there are multiple component that need all or part of the data. It would be more wisely to store them into a middle layer , we can filter our data there can call it from component whenever we need it . We call backend api to get the data and then put them into store .For any component that need this data they call it from store . Here we need the trip information corresponding to our user’s search , once user press search button , the service will get data from backend and inject into tripDTOs in our store , then we can call the dto and inject them into table and display it to user .

**MOBX**

MobX is a battle tested library that makes state management simple and scalable by transparently applying functional reactive programming (TFRP).

We can wrap our React component with observer wrapper , then mobx will take care of re-rendering the component when precisely that field is changed updated in the future. [21] .

In our store we are annotating some of our variables with **@observable** annotation . this tells mobx that for Every event invokes an action which updates observable state , changes in the observable state are propagated precisely to all computations and side effects that depend on the changes being made. [22]



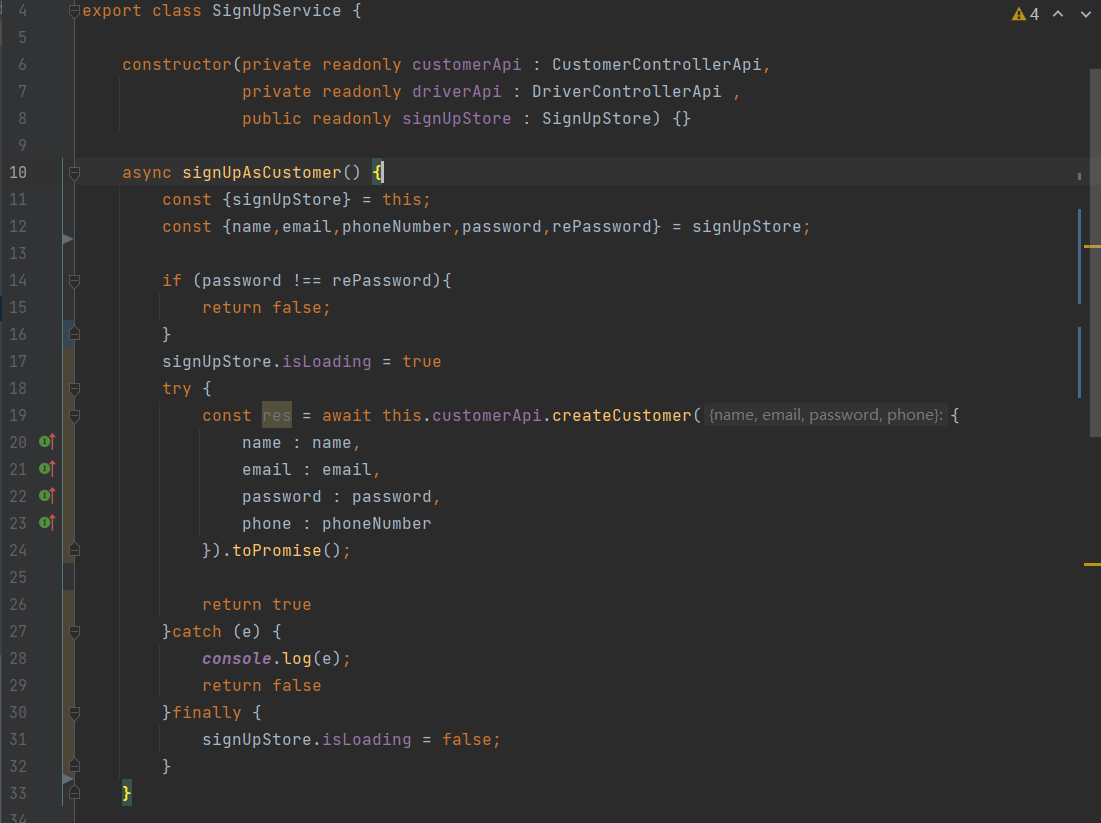
MobX unidirectional flow

This conceptual picture can be applied to the above example, or any other application using MobX.

###### Service

In service layer we mainly deal with the connections with backend , it can be seen as a custom fetch api .When posting , we get the data from store , process and form the data before sending the post request . When getting data from backend , we can inject the data into corresponding store , filter and form the data before injecting them into component .

We will call the generated api from service layer , communicate with backend controllers . Let’s take sign up as an example :



For signUpAsCustomer method, we are getting the necessary data from signUpStore , as it is injected to store when user submit the sign up form . We do the last password check , as we have already done in components , but we will check again before sending to backend . Then we put our api calling code into try block and catch relative errors. Inside try block we call the corresponding api method and pass down all the data to accomplish a post request .

**Async and Await**

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically. The await keyword works only inside async functions,The keyword await makes JavaScript wait until that promise settles and returns its result. [23]

The function execution “pauses” at the line 19 and resumes when the promise settles, when we get the response from backend , the res become a result. Which means our post request runs successfully and our sign up action is success . Till then we can resume the process and return the true value .

The return value is actually quite useful , as we can use it to check if the process runs properly , for example we can redirect the user to main or personal page when register successful or keep them at sign up page when register not success .Using async and await keyword is a more elegant syntax of getting the promise result than “promise.then.” And, it’s easier to read and write.

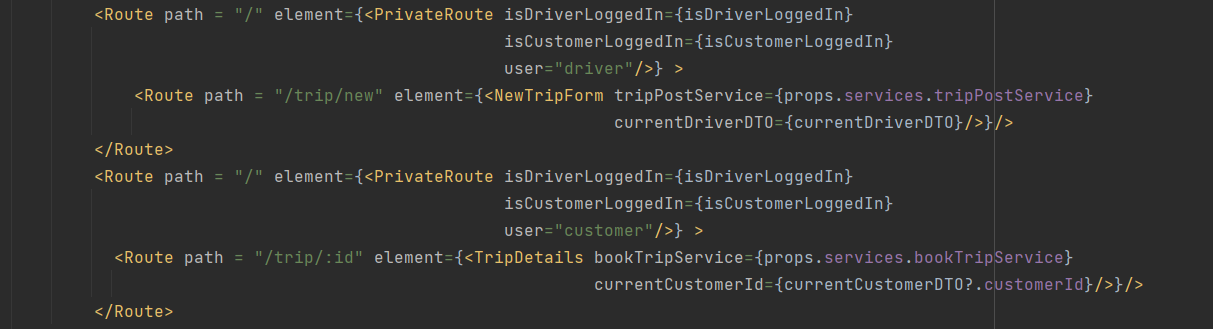
###### Route Authentication

In react , we are accessing different pages by routes , some routes are public , they don’t user login to access to those pages . Such as our landing page , trip search page etc . But some page do requires a login status , like personal page , we would like user to login before accessing their personal details . Same for trip details page , when user want to know the details and driver info of a trip ,it’s better to require a login status as well . Since our pages are managed by routes , we can do the authentication by deciding to route the page or not , depend on the user login status . We can have a separate class to do the job , let’s call it private route :



Inside the PrivateRoute class , we pass down the login status , and check if they are login . We are using Outlet component : it renders the child route's element, if there is one.

The whole logic now works , if logged in (authorized),we will return an outlet that will render child elements , while if not logged in yet (unauthorized),we return element that will navigate to sign in page.



Back to our PublicApp class , where our routes at , in order to maake authentication work , we surround our “private ” routes with our authentication class , it will render the child component , which is our “private ” component when login status is true , or render the sign in page vice versa .

###### Material-UI

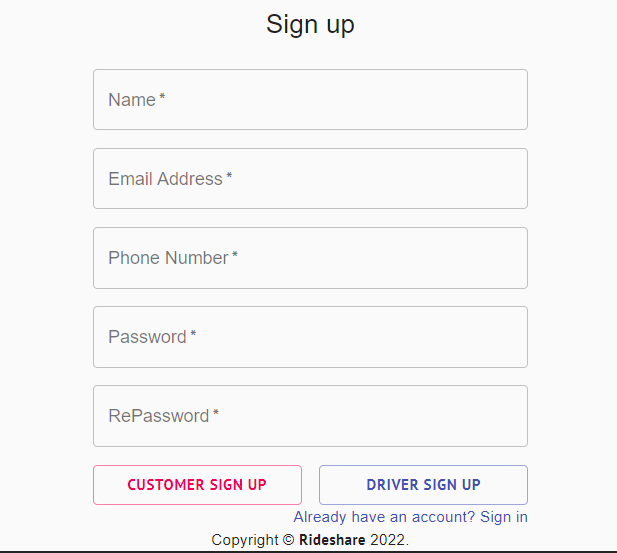
**Introduction**

Material-UI is simply a library that allows us to import and use different components to create a user interface in our React applications. This saves a significant amount of time since the developers do not have to write everything from scratch.[24]

Material-UI is a UI component library for react . It contains quite a lot of ready-to-use components like buttons , navbars , tables , grid systems etc. One good thing about Material-UI is that it’s very customizable . We can customize the Material-UI components simply by using props or override the Material-UI components system by create our own CSS code or useStype hook that Material-UI gives us .

**Text field**

Material-UI provides text field library to let users enter and edit text.



In sign in form we used Material-UI text field to get user login in input , let us take password text field as an example ,here’s how it is implemented :

1. <TextField
2. onChange={(e) => {
3. signUpStore.rePassword = e.target.value;
4. }}
5. variant="outlined"
6. color="primary"
7. required
8. fullWidth
9. name="password"
10. label="RePassword"
11. type="password"
12. id="re-password"
13. autoComplete="new-password"
14. error={rePasswordError}
15. />

As we can see , we can customized the the whole text field by passing down props , and store the value by adding on change event . In on change event , we take user inputs in the password field and store it into store .Fullwidth property give the text field full width in current grid . Type = password indicates that the text shows in text field is black dots instead of password text . AutoComplete tells our browser what to auto completed , if user has logged in before and saved the password , it will help to improve the user expirence .

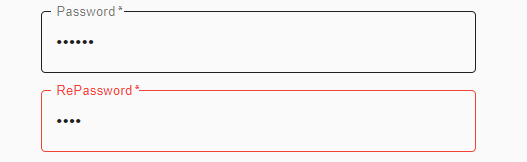
**Validation**

Before we submit the form , it is necessary to validate the data and tell user if there’s anything inappropriate . We can do validation using useState hook together with MUI built in error state .

We have a error prop in Material-UI text field , it receives a boolean , and turn to error statues to warn user when the boolean value is true , we can use this to do our validation . We can declear a boolean variable : rePasswordError

1. **const** [rePasswordError , setRePasswordError] = useState(**false**);
2. **if**(signUpStore.password !== signUpStore.rePassword){
3. setRePasswordError(**true**);
4. }**else** setRePasswordError(**false**);

We check if the password and repassword are the same , if they are not the same , we set the error state to true .



Error state

**Grid System**

When it comes to layouts using material ui the main component that we will use is the grid components. Material Ui offers us a 12 column grid system based on flexbox . In order to use grid system , first thing we are going to use is a grid container , it is like a wrapper and wraps around all the elements inside the grid

1. <Grid container spacing={2}>
2. ...//code inside
3. </Grid >

The we put the grid items inside the container :

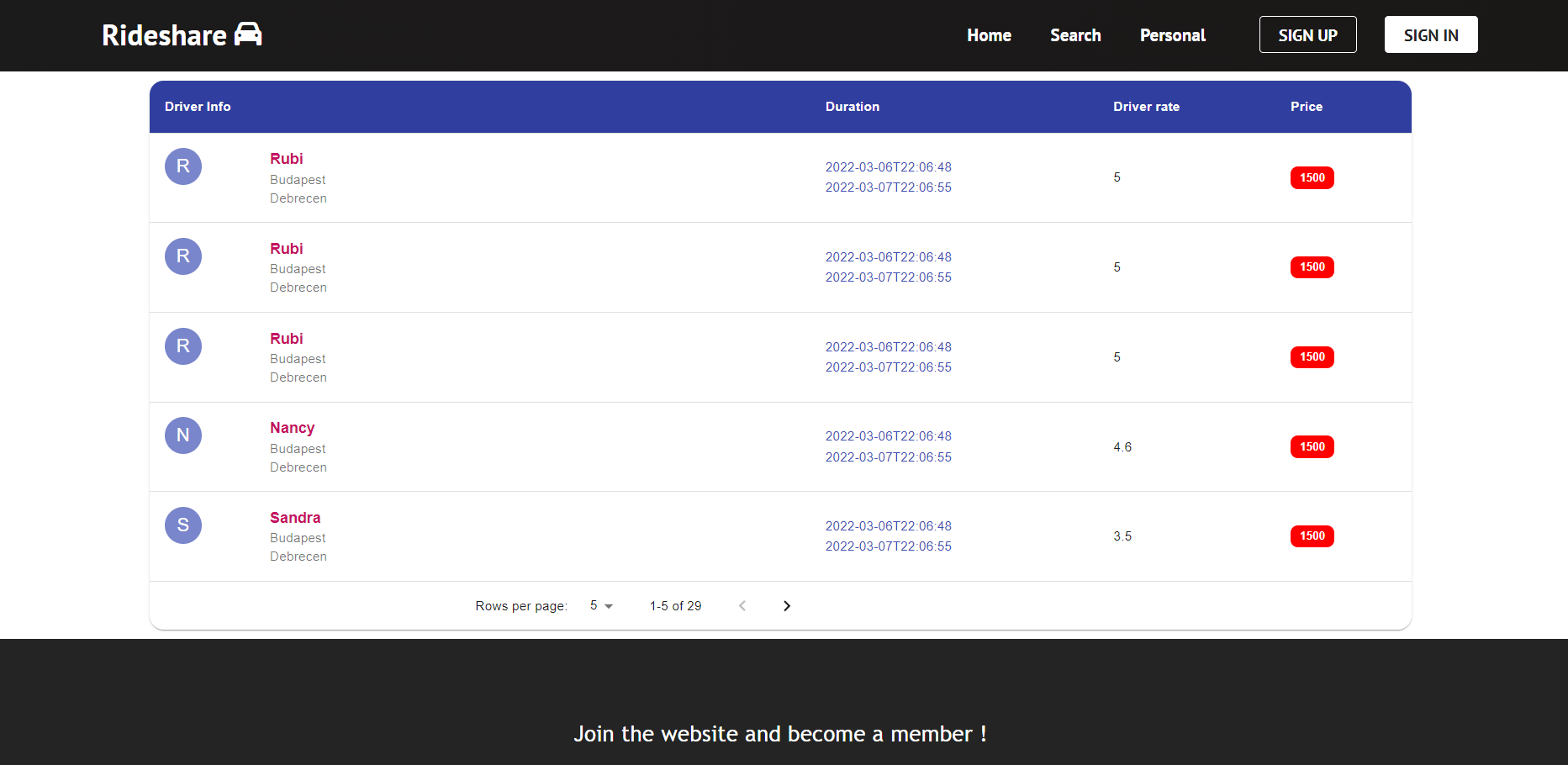
1. <Grid item xs={12} sm={6}>
2. <Button
3. type="submit"
4. fullWidth
5. color="secondary"
6. variant="outlined"
7. onClick={async (e) => {
8. e.preventDefault();
9. **if** (isSyntaxOk()){
10. signUpService.signUpAsCustomer();
11. }
12. }}
13. >
14. Customer Sign up
15. </Button>
16. </Grid>
17. <Grid item xs={12} sm={6}>
18. <Button
19. type="submit"
20. fullWidth
21. color="primary"
22. variant="outlined"
23. onClick={async (e) => {
24. e.preventDefault();
25. **if** (isSyntaxOk()){
26. signUpService.signUpAsDriver();
27. }
28. }}
29. >
30. Driver Sign up
31. </Button>
32. </Grid>

We pass down the item prop to indicate that it is a grid item , xs={12} represents the 12 column in our grid system and sm={6} indicates that we are only going to take 6 of them .Since each button takes 6 if them , two buttons are going to share the half of the grid area , as we want them to .



**Table**

Material-UI also provide table to display sets of data, which can be fully customized as well . For our trip searching , table is a good option to display the result set of data .



Here’s how it is implemented using Material-UI :

1. <TableContainer component={Paper} className = {styleClasses.tableContainer}>
2. <Table className = {styleClasses.table} aria-label="simple table">
3. <TableHead>
4. <TableRow>
5. <TableCell className = {styleClasses.tableHeaderCell}>Driver Info</TableCell>
6. <TableCell className = {styleClasses.tableHeaderCell}>Duration</TableCell>
7. <TableCell className = {styleClasses.tableHeaderCell}>Driver rate</TableCell>
8. <TableCell className = {styleClasses.tableHeaderCell}>Price</TableCell>
9. </TableRow>
10. </TableHead>
11. <TableBody>
12. {tripTable
13. .slice(page \* rowsPerPage, page \* rowsPerPage + rowsPerPage)
14. .map((row) => (
15. <TableRow
16. key={row.id}
17. hover = {**true**}
18. onClick={(e) => {
19. e.preventDefault();
20. navigate(`/trip/${row.id}`);
21. }}
22. >
23. <TableCell component="th" scope="row">
24. <Grid container>
25. //grid items inside to display data in table cells
26. </Grid>
27. </TableCell>
28. //Table cells in table row
29. </TableRow>
30. //table rows in table body
31. </TableBody>
32. //table footer
33. </Table>

Once the user perform a search and we get the data set from backend , we can map them into table using map function . In Material-UI table we have :

table head , which is the header of table , we can define and customize the header by adding table cells in header and styling them .

Table body : the main body of the table . In table body we can map each trip info into each table row , and display different data in different table cells . Inside table cell we can using our Grid system to style them .

Table footer : the bottom part of the table , we can add our paginator here :

1. <TableFooter>
2. <TablePagination
3. rowsPerPageOptions={[5,10,15]}
4. component="div"
5. count={tripTable.length}
6. rowsPerPage={rowsPerPage}
7. page={page}
8. onPageChange={handleChangePage}
9. onRowsPerPageChange={handleChangeRowsPerPage}
10. />
11. </TableFooter>

By passing rowsPerPageOptions prop we can control how many trips we would like to display for each page . The can also help to improve user experience and application performance . When we have a large amount of trip data , this will be quite helpful since it can significantly reduce the amount of requested data , which can protect our sever and make the app react faster a well .

# Conclusion

Throughout the development of this web application . It strengthen my spring boot and react knowledge. It is my first time to experience full stack development and would love to learn more .

In this software development , several important technical stack are used : creating and manipulating postgresSQL database , mapping entities into database and adding test data using liquibase , using mapstruct to map between entities and dtos , dockerizing the database and application ,connecting two side using openApi generator , DOM manipulation in typescript , authentication in react .

For the current application , although it is ready to use and have core functionalities , there are still quite many place to improve with :

1. For now the register and login is control is based on user email and password only , while modern web application usually provide various login or register options such as using google or other social account . Connect the application to firebase to various authentication method will be a big improvement .
2. In trip searching bar , users can type departure place and destination . Even it is implemented to be case-insensitive , user still need to spell the name correctly in order to perform a successful search . We can improve the search bar to make it more functional .
3. Now customer can book trip from driver after search , it would be better if the driver and customer can rate each other and have a short comment after the trip , we can even create a community from it .
4. Customer should be able to cancel the trip that they booked for certain reason . And furthermore it would be good if we can support payment system in the website .

# Bibliography

[1] Wikipedia Contributors. “Carpool.” Wikipedia, Wikimedia Foundation, 4 Aug. 2019, en.wikipedia.org/wiki/Carpool.

[2] “Desktop App vs Web App: Comparative Analysis.” Digital Skynet, 3 June 2020, digitalskynet.com/blog/Desktop-App-vs-Web-App-Comparative-Analysis.

[3] Hristozov, Krasimir. “MySQL vs PostgreSQL -- Choose the Right Database for Your Project.” Okta Developer, 19 July 2019, developer.okta.com/blog/2019/07/19/mysql-vs-postgres.

1. Schlosser, Hartmut. “Java Trends: Top 10 Frameworks in 2020.” JAXenter, 25 Feb. 2020, jaxenter.com/java-trends-top-10-frameworks-2020-168867.html.
2. “Spring Framework Documentation.” *Docs.spring.io*, docs.spring.io/spring-framework/docs/current/reference/html/.

[6] kashiffiroze95. “Bean Life Cycle in Java Spring.” GeeksforGeeks, 21 May 2020, www.geeksforgeeks.org/bean-life-cycle-in-java-spring/. Accessed 27 Mar. 2022.

[7] randika. “RANDIKA’S TECH BLAST.” Randikatech.blogspot.com, 19 Sept. 2019, randikatech.blogspot.com/. Accessed 27 Mar. 2022.

[8] Shmeltzer, Shay. “Introduction to Liquibase and Managing Your Database Source Code - DZone Database.” Dzone.com, 23 Oct. 2017, dzone.com/articles/introduction-to-liquibase-and-managing-your-databa. [9] Paraschiv, Eugen. “Introduction to Spring Data JPA | Baeldung.” Www.baeldung.com, 24 May 2014, www.baeldung.com/the-persistence-layer-with-spring-data-jpa.

[10] baeldung. “Spring Validation in the Service Layer | Baeldung.” Www.baeldung.com, 7 June 2021, www.baeldung.com/spring-service-layer-validation. Accessed 12 Mar. 2022.

[11] RedHat. “What Is a REST API?” Www.redhat.com, 8 May 2020, www.redhat.com/en/topics/api/what-is-a-rest-api.

1. “MapStruct – Java Bean Mappings, the Easy Way!” Mapstruct.org, mapstruct.org/. Accessed 27 Mar. 2022.
2. IBM Cloud Education. “What Is Docker?” Www.ibm.com, 13 June 2021, www.ibm.com/cloud/learn/docker.
3. Flade, Joerg. “Docker — What It Is, How Images Are Structured, Docker vs. vm and Some Tips (Part 1).” Medium, 12 July 2020, ragin.medium.com/docker-what-it-is-how-images-are-structured-docker-vs-vm-and-some-tips-part-1-d9686303590f. Accessed 15 Mar. 2022.
4. Waikar, Prakash. “Docker Volume.” Medium, 16 July 2019, prakashkumar0301.medium.com/docker-volume-ba830fbbdde5.
5. Sandoval, Kristopher. “Introduction to OpenAPI Generator | Nordic APIs |.” Nordic APIs, 22 Apr. 2020, nordicapis.com/introduction-to-openapi-generator/. Accessed 17 Mar. 2022.
6. Microsoft. “TypeScript - JavaScript That Scales.” Typescriptlang.org, 2015, www.typescriptlang.org/.
7. Vishnupriya. “What Is Typescript ?” Medium, 15 Dec. 2018, medium.com/@vishnupriya\_web/what-is-typescript-faa0890b2baf.
8. Lvivity Team. “Single-Page Application vs Multi-Page Application: Pros, Cons, and Which Is Better?” Lvivity, 13 Aug. 2020, lvivity.com/single-page-app-vs-multi-page-app.
9. harsh.agarwal0. “ReactJS | Components.” GeeksforGeeks, 10 Mar. 2018, www.geeksforgeeks.org/reactjs-components/.
10. “README · MobX.” Mobx.js.org, mobx.js.org/README.html.
11. Absolon, Peter. “MobX State Management in React - NearForm.” NearForm, 4 May 2021, www.nearform.com/blog/mobx-state-management-in-react/. Accessed 20 Mar. 2022.
12. “Async/Await.” Javascript.info, 2020, javascript.info/async-await.
13. Barasa, Michael. “How to Implement Material-UI in React.” Engineering Education (EngEd) Program | Section, 14 Feb. 2021, www.section.io/engineering-education/how-to-implement-material-ui-in-react/