**Thesis**



*Wang Yuxuan*

*Debrecen*

*2022*

**University of Debrecen**

**Faculty of Informatics**

**A Ride-share Web Application Based on Java and TypeScript**

Supervisor Candidate

Dr. Anikó Vágner Wang Yuxuan

*Debrecen*

*2022*

Contents

[I. Introduction 4](#_Toc99452520)

[1. Introduction to rideshare 4](#_Toc99452521)

[2. Web application and Web development 4](#_Toc99452522)

[3. Introduction to Implementation Technology Stack 5](#_Toc99452523)

[II. Backend Technologies and Implementation 6](#_Toc99452524)

[1. PostgreSQL 6](#_Toc99452525)

[i. Introduction 6](#_Toc99452526)

[ii. Database Design in Carpooling 7](#_Toc99452527)

[2. Spring Boot 7](#_Toc99452528)

[i. Introduction to spring boot 7](#_Toc99452529)

[ii. Spring boot beans and life circle 8](#_Toc99452530)

[iii. Layers in spring boot 9](#_Toc99452531)

[iv. Building Entity In Spring Boot 10](#_Toc99452532)

[v. Map Entities into the database using Liquibase 13](#_Toc99452533)

[vi. Repository layer 15](#_Toc99452534)

[vii. Service Layer 16](#_Toc99452535)

[viii. Controller Layer 17](#_Toc99452536)

[ix. DTO(Data Transfer Objects) 18](#_Toc99452537)

[x. Mapping with MapStruct 19](#_Toc99452538)

[3. Docker 21](#_Toc99452539)

[i. Introduction 21](#_Toc99452540)

[ii. Images and containers 22](#_Toc99452541)

[iii. Volumes in Docker 24](#_Toc99452542)

[iv. Docker Compose 25](#_Toc99452543)

[III. Frontend Technologies and Implementation 27](#_Toc99452544)

[1. Connect Frontend with Backend 27](#_Toc99452545)

[2. TypeScript 29](#_Toc99452546)

[3. React 30](#_Toc99452547)

[i. Introduction 30](#_Toc99452548)

[ii. Our react structures 32](#_Toc99452549)

[iii. Components 33](#_Toc99452550)

[iv. Store layer with Mobx 34](#_Toc99452551)

[v. Service 36](#_Toc99452552)

[4. Material-UI 39](#_Toc99452553)

[i. Introduction 39](#_Toc99452554)

[ii. Text field 39](#_Toc99452555)

[iii. Grid System 42](#_Toc99452556)

[iv. Table 43](#_Toc99452557)

[IV. Conclusion 47](#_Toc99452558)

[V. Bibliography 48](#_Toc99452559)

# Introduction

## Introduction to rideshare

Ride-share (Carpooling) is a brand-new type of traveling, the main idea of it is to share car journeys to reduce each person’s cost for the trip. Drivers will post their ride on the platform, and passengers will join part of or the whole journey based on their needs, and share the traveling cost. Drivers and passengers will contact each other to make an appointment for pick-up places and may negotiate special needs (such as large luggage) and prices. 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 to reduce air pollution and traffic congestion on the roads.[1]

In order to post or share the carpooling information, we need such a platform. There are many types of platforms for it: website, carpooling agency, pick-up point, carpooling groups, etc. Nowadays, web 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 software that runs on a web server. It is more portable, light than desktop applications. You can open it anywhere on any platform as long as you have a browser. Apart from that, as web apps are cloud-based, it is easier to sync all the data and info between all the devices.

There are several reasons why I am in favor of web applications 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 users 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 server-side, no matter what platform we are using, we can easily access it.

* **Automatic updates**

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

* **Light on Computer Resources**

Web applications take significantly less resources on our computer, since most of the services are provided from the server-side, it takes significantly less memory and processor compared with desktop or mobile applications.[2]

## Introduction to Implementation Technology Stack

For web applications, mostly we have frontend and backend. The backend is more about dealing with the database, handling data, server, and implementing main logic and then providing the APIs to the frontend, while the frontend is more about user interface and interaction, sending or requesting data from the backend.

As for our Technology Stack to implement such a carpooling website, we are going to use PostgreSQL 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.

1. Backend Technologies and Implementation

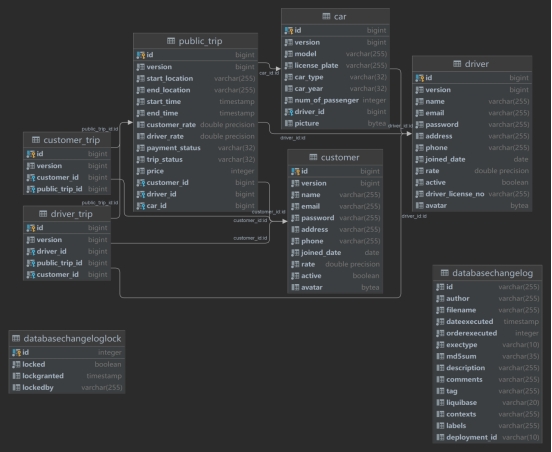
## PostgreSQL

### Introduction

PostgreSQL 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 Design in Carpooling

We will store data in the database. The database design is shown below:

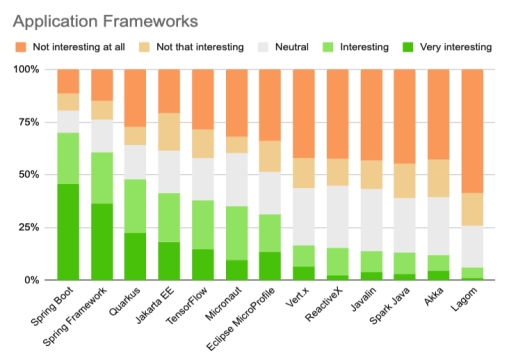


*Figure II-1 : Database Design in Relation Table*

## Spring Boot

### Introduction to spring boot

Spring boot is one of the most famous and popular frameworks, it is open-source, microservice-based, and makes building backend or even full-stack development using Java or kotlin simple and easy, it gives us everything we need to build applications.



*Figure II-2 :* Spring Boot is Dominating in Java [4]

### Spring Boot Bean and Life Circle

**What is spring boot bean**

By official document:

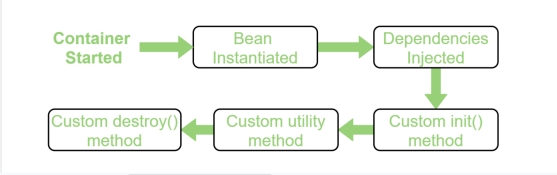
**In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.**[5]

In short, the Spring boot bean is still a Java object but managed by the 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. The 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.

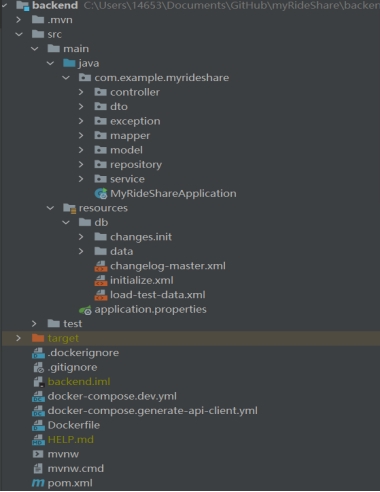
By writing code in the custom init() and destroy() method, we can execute our custom on the bean instantiation and destruction.

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

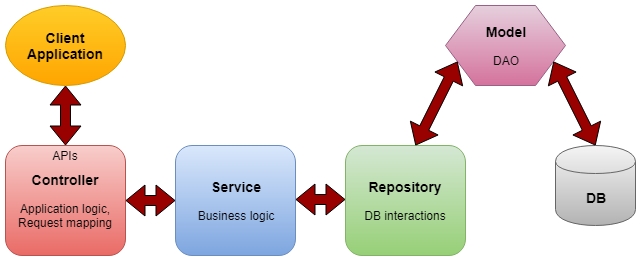


*Figure II-3 : Spring Boot Bean Life Circle*

### Layers in spring boot



*Figure II-4 :* Structure in Backend



*Figure II-5 :* Layers in Spring Boot

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

**Service layer:** The service layer is in the middle of the controller and repository layer, which performs the business logic and validation logic. It manipulates the data that we get from the repository layer before sending it to controllers. And also form the user input data before we interact with the database when calling the 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 that are acting as the DTO (Interact with application-level data transfer) or DAO (Interaction with database operations) [7]

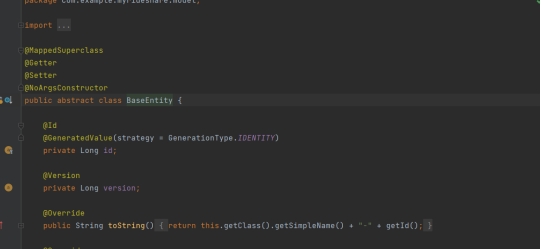
**Database Layer**: In the database layer, CRUD operations are performed and this is the layer where we store our data.

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

### Building Entity in Spring Boot

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

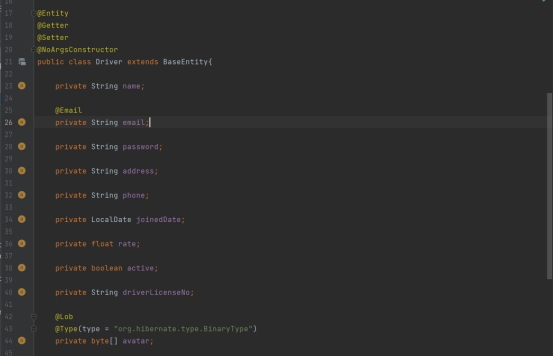
***Base Entity***



*Figure II-6 : Part of Base Entity Class*

For every entity in our database, they do need 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 an abstract class and let the other entity classes extend it. So that they will have an 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:



*Figure II-7 : Driver Entity Class*

Firstly, we need to extend our Base Entity class and mark **@Entity** annotation to tell Spring Boot it’s a database entity.

Attributes

For attributes we have:

name: driver’s name,

email: with **@Email** annotation 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 the driver registered himself with an email, the active field will be set to false since the email has not confirmed the email yet, and the driver cannot log in 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 the database are in a 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 the 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 relationships, but how do we handle them in Spring Boot?



*Figure II-8 : One-to-many Relationship in Driver Table*

As in Figure II-8, we annotate them with **@OneToMany** annotation, since it’s a one-to-many relationship, we represent them as a list here. As we store driver id in the Car and Public trip table, in those classes, we need to do the following:



*Figure II-9 : Many-to-one Relationship in PublicTrip Table*

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

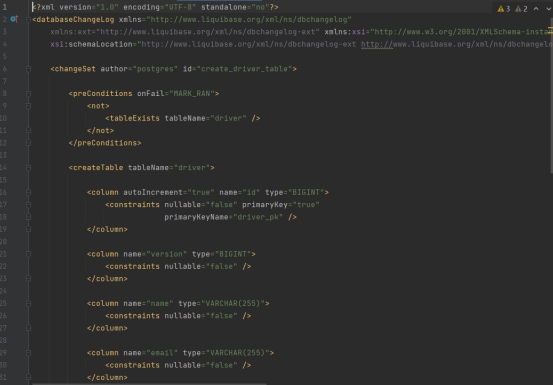
### Map Entities into database using Liquibase

For now, we have our relational model and entities classes ready, but our database is still empty, neither tables nor data are there. We need to map all our designs into database tables. We definitely can 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:



*Figure II-10 : Liquibase Changelog in Building Driver Table*

In Figure II-10 we can see what changelogs look like. Inside our changelog, we have our changeset, it can be multiple changesets, but for each changeset, we need to specify an 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 constraints to the column, such as not null, unique, etc.

Dealing with relationships in Liquibase

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

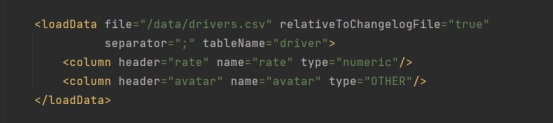


*Figure II-11 : Relationships in Liquibase*

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 the database using Liquibase

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



*Figure II-12 : Load Data Using Liquibase*

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 the database, using CSV is a bit hard to handle that, we can use an update statement instead.

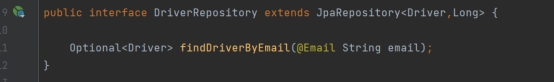


*Figure II-13 : Inserting Pictures using Liquibase*

After finishing our model layer and mapping everything into the database, we are finally able to store our data. Our next step will be building our repository layer 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. 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]



*Figure II-14 : DriverRepository Class*

When extending, we need to provide two parameters, the 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 some more complex and custom queries, 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 the service layer, we will call the 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:



*Figure II-15 : Part of CustomerService Class*

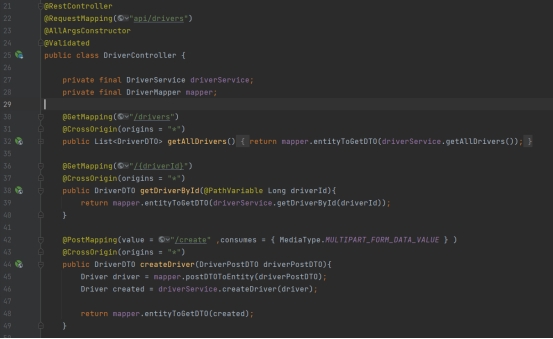
For the service class, we need to annotate it with **@Service** annotation, it indicates that the current class belongs to the service layer, it is a specialization of **@Component** Annotation as well.

Inside the service layer, we will handle transactions with our repository layer and provide the data to the 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 are also inside the service layer, we can use streams to filter our data and only send the necessary data to the 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]

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



*Figure II-16 : Part of DriverController class*

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

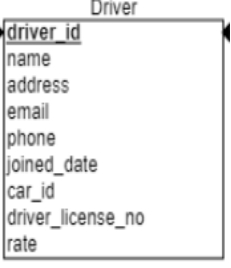
The path variable is the variable that you can pass within the URL, usually quite useful when you want to access a certain item in a list based on id. When annotated with **@PathVariable**, the template variable in the URL will be mapped and set as method parameters.

### DTO (Data Transfer Objects)

Here we notice that the objects that we returning to the frontend side are DTOs instead of entities. A **DTO** (data transfer object) is an object that is used to encapsulate data before sending it 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 returning the whole entity or all the required entity to the 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 as an example:



*Figure II-17 : DriverPostDTO and Driver Table*

DriverPostDTO is used for driver registration. There are quite many attributes in the driver table, but we surely do 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 postDTO, to tell the 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

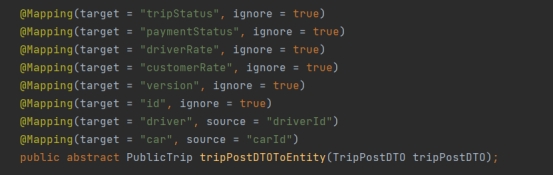
**@NotNull**: indicates that the element should not be null

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

### Mapping with MapStruct

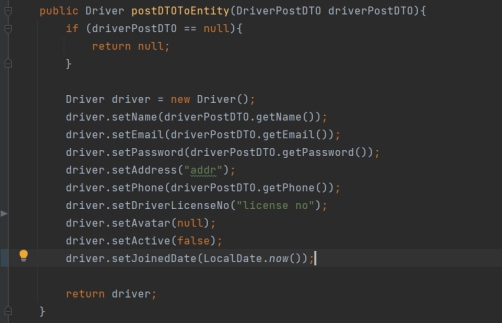
After we have our DTO, another problem is the mapping between DTO and entity. MapStruct is a good tool for it. MapStruct is a code generator that greatly simplifies the implementation of mappings between Java bean types based on a convention over configuration approach. The generated mapping code uses 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 an abstract class or interface, and the mapping code will be auto-generated. When there’s a naming mismatch and MapSturct cannot find the corresponding variable, we need to annotate the method with **@Mapping** and tell MapStruct which variable it is by adding it in the source tag, or simply adding ignore = true to ignore it.



*Figure II-18 : Mapping TripPostDTO to Entity Using MapStruct*

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



*Figure II-19 : Customized Mapper*

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

## Docker

### Introduction

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 distributed application delivery and have become increasingly popular as organizations shift to cloud-native development and hybrid multi-cloud 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 applications 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, to let my application run on the other computers properly, the other computers need to have the same Java and Postgres version installed, which made a huge work just for running the application. And different applications may require different versions 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 server since all the configuration is set up in docker container already.

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

### Images and containers

Docker images are like the blueprints of the container. They have the things stored 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 work

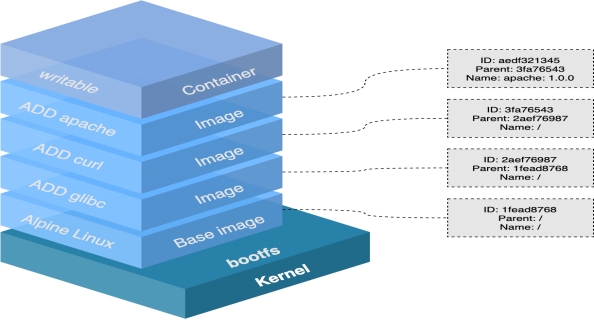
Images have them stored inside but now running them. They are an independent file system from your computer and read-only, which means once an image is created it cannot be changed. If you need to apply some new changes you need to create a new brand-new image.

Containers are runnable instances of images, it is a process exactly outlined as the image we created. A 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 needs 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 a 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 of different layers. where each layer adds something else to the image incrementally, the order of the images does matter.



*Figure II-20 : Docker images [14]*

As shown in Figure II-20, we start with our base image or also as known as the 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 to it such as our source code, install dependencies, and so on. Usually, we can get our pre-made first layer from the docker hub.

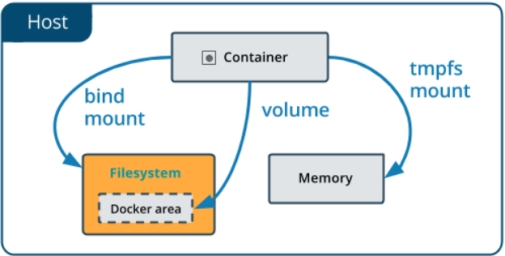
Layer Caching

Since the image is built by layers being stacked on top of each other, each time we add 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 to add those layers in turn, which might be quite time-consuming when building the layer from the base layer. The actual is, docker will cache our image layers when after building it, which ends up with a cached version image with every layer. Every time docker tries to build an image and work through the layers, it will look into the caches and try to find an 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.



*Figure II-21 : Docker volume [15]*

Volume 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 folders 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.

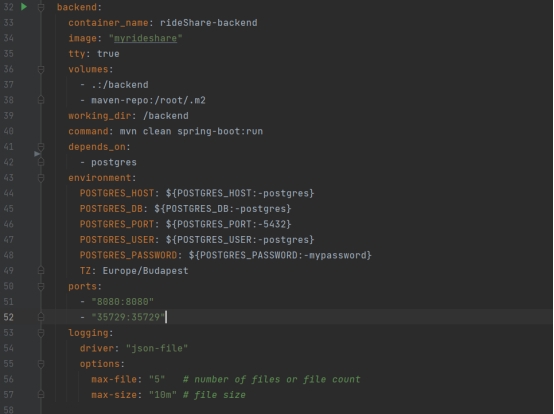
Volumes provide us a way that we can make changes and preview them without rebuilding the images all of the time. But one thing is important, the image will not change. Volume is just mapping the changes into containers and our image remains as it was. If we want to share the latest changes with the 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 the terminal one by one. And sometimes we may have multiple projects and we would like to run those containers at once, since one may depend on another as our frontend depends on the backend and the backend is dependent on the 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 an easier way to manage our containers which is something called docker compose.

Docker compose is a tool that gives us a way to make a single docker compose file that contains all the container configuration and dependencies.

Dockerize our Spring Boot application



*Figure II-22 :* *Dockerize Spring Boot application*

As shown in Figure II-22 , we will specify the following attributes in our docker compose file:

**container\_name**: the container name.

**image**: the base image that we need to use.

**volumes**: The list of volumes that we need, use a 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 the Postgres container

**environment**: the environment variables, here is mostly the database configure and authorization

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

**logging**: logging configures

After configuring everything in our docker compose file, we just need to simply type the following command to run our application, which is much simpler than creating images volumes, and containers step by step.



*Figure II-23 : Docker Compose Command*

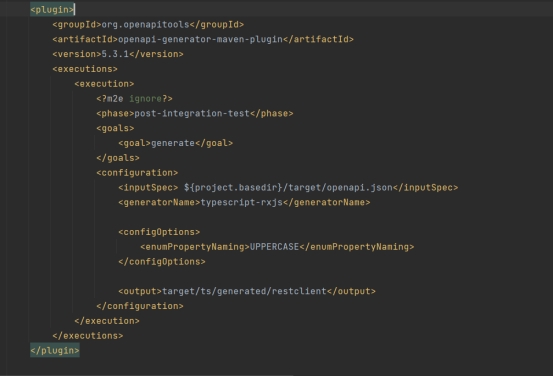
# Frontend Technologies and Implementation

## Connect Frontend with Backend

When we have our backend ready and some frontend features, we are having a problem sending or getting data from the backend. The backend provides API for the frontend and usually, they are in the controller, but how could we connect the two sides? Surely, we can make HTTP requests all the time on our own, but it is better and easier to invoke if we have our DTOs and rest APIs on our frontend side. That’s when the open API generator comes 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



*Figure III-1 : Open API Plugin*

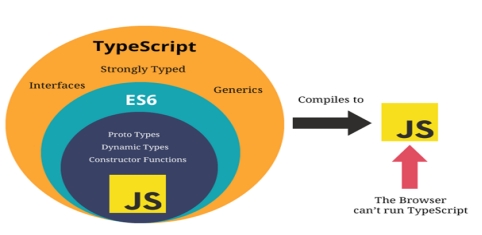
After dockerizing our API generator, we can generate our API by typing the following line in the terminal. Then we will have our API and DTO generated and ready to use on our frontend.



*Figure III-2 : Docker Compose Command to Run Open API Generator*

## TypeScript

TypeScript is the programming language in frontend. TypeScript is a superset 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]**



*Figure III-3 : 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 understand it. But do 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 make our code becomes easier to debug.

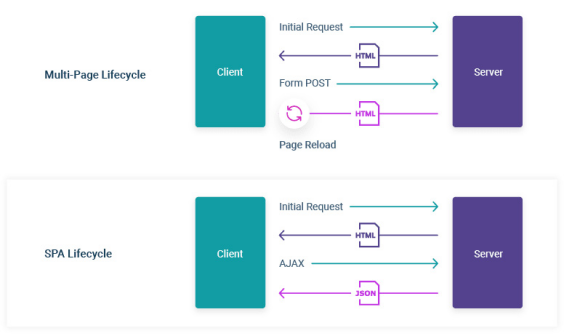
TypeScript not only includes modern JavaScript features like arrow functions, let, const, but also has many more features that JavaScript does not have: interfaces, generic, enums, tuples, etc. These features will help us a lot in web development.

## React

### Introduction

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

Multi-Page Applications and Single-Page Applications



*Figure III-4 : MPA and SPA*

Multi-Page Application

A Multi-page Application is a web application consisting of a large number of pages completely refreshed whenever 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 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 server 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 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 application, 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

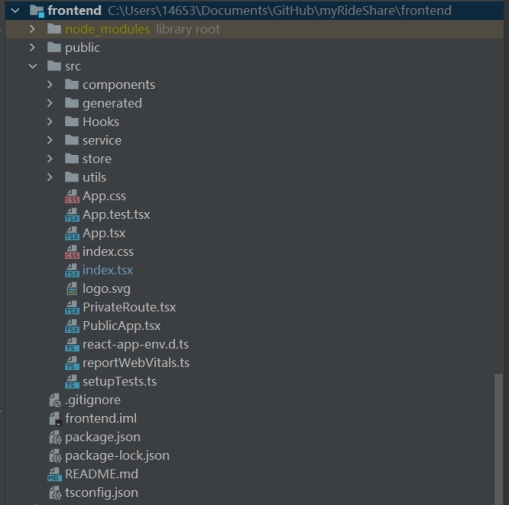
Reacts takes over and manages the whole website including any kinds of data and user interactivity such as click events and routing from page to page. Instead of sending a new request to the server when the user navigates from page to page, react will change the content of the browser depending on the URL that the user clicked and inject into the page. By contrast, traditionally, for every link, the user clicks the browser is going to send a request to the server for a new HTML page. This makes react applications usually load quickly and results in a very speedy user experience.

To create a new react app, we simply type:



*Figure III-5 : Create React App*

### Our react structures

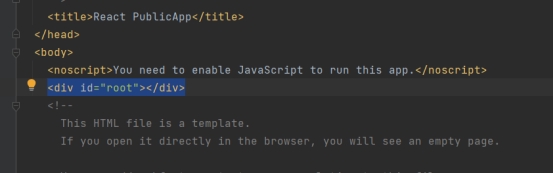


*Figure III-6 : Frontend React Structure*

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

Public folder: Including all the public files to the browser.

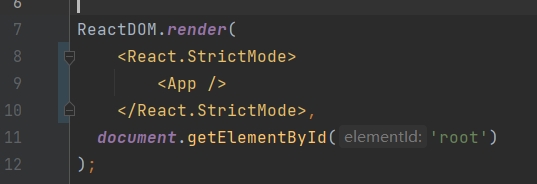
Index.html: As shown in Figure III-7 inside the public folder we have this index.html file, it is the one HTML file that is served to the browser and all of our react code will be injected into this div with the id of root.



*Figure III-7 : Index.html*

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

Index.tsx: This file is responsible for taking all of our react components and mounting them to the dom. In our index.tsx file, we are rendering our App component to the dom.



*Figure III-7 : Index.tsx*

### Components

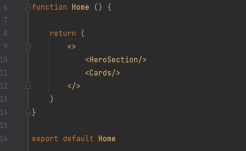
Components are the uppermost layer, they usually, return HTML elements and tell react what should be rendered into dom. Components are what we see on the web page. In React, we mainly have two types of components:

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.

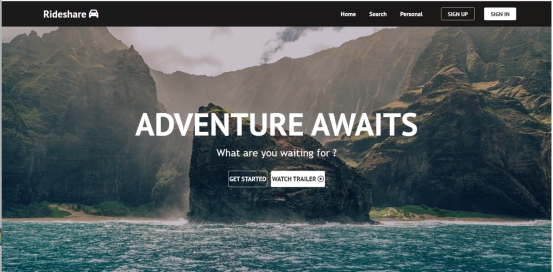
Class Components:

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].**



*Figure III-8 : Home component in Functional and class type*

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



*Figure III-9 : Our home page*

### Store layer with Mobx

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

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



*Figure III-10 : TripResultStore.tsx*

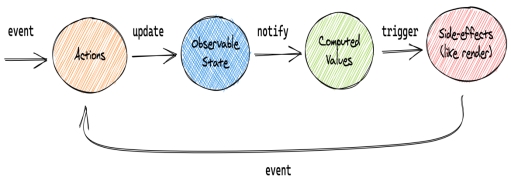
When we perform a trip search, we will send a get request to the backend, and if everything goes right, we will get back the data. But the problem is where would we store it. If we inject it directly into components it would duplicate when multiple components need all or part of the data. It would be wiser to store them in 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 the store. For any component that needs this data, they call it from the store. Here we need the trip information corresponding to our user’s search, once a user presses the search button, the service will get data from the backend and inject it into tripDTOs in our store, then we can call the DTO and inject them into the table and display it to the 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 an observer wrapper, then mobs 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 that updates the observable state, changes in the observable state are propagated precisely to all computations and side effects that depend on the changes being made. [22]



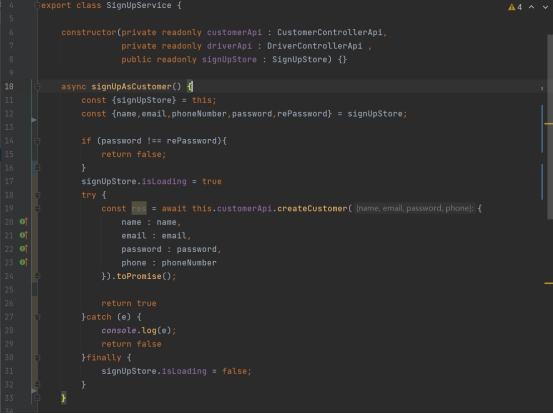
*Figure III-11 : MobX unidirectional flow*

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

### Service

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

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



*Figure III-12 : Sign Up Method in Service Layer*

For signUpAsCustomer method, we are getting the necessary data from signUpStore, as it is injected into the store when the user submits the signup form. We do the last password check, as we have already done in components, but we will check again before sending it to the backend. Then we put our API calling code into the try block and catch relative errors. Inside the 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 line 19 and resumes when the promise settles, when we get the response from the backend, the res becomes a result. This means our post request runs successfully and our sign-up action is a success. Till then we can resume the process and return the true value.

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

Route Authentication

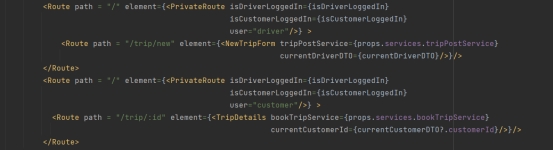
In react, we are accessing different pages by routes, some routes are public, they don’t user login to access those pages. Such as our landing page, trip search page, etc. But some page does require a login status, like a personal page, we would like the user to log in before accessing their details. Same for the trip details page, when users 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, depending on the user login status. We can have a separate class to do the job, let’s call it a private route:



*Figure III-13 : Route Authentication-1*

Inside the PrivateRoute class, we pass down the login status and check if they are login. We are using the 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 an element that will navigate to the sign-in page.



*Figure III-14 : Route Authentication-2*

Back to our PublicApp class, where our routes are at, in order to make 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

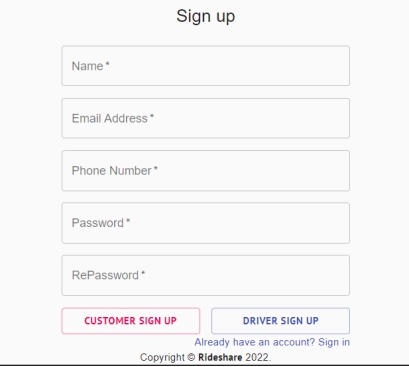
### Introduction

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 creating our own CSS code or useStype hook that Material-UI gives us.

### Text field

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



*Figure III-15 : Our Sign Up Form Using* Material-UI

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. />

*Figure III-16 : Password Text Field*

As we can see, we can customize the whole text field by passing down props, and storing the value by adding on a change event. In on change event, we take user inputs in the password field and store them in store. The fullwidth property gives the text field full width in the current grid. Type = password indicates that the text shown in the text field is black dots instead of password text. AutoComplete tells our browser what to auto completed, if the user has logged in before and saved the password, it will help to improve the user experience.

Validation

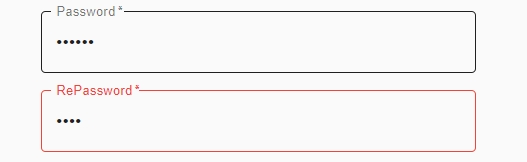
Before we submit the form, it is necessary to validate the data and tell the user if there’s anything inappropriate. We can do the validation using the useState hook together with MUI built-in error state.

We have an 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 declare a Boolean variable: rePasswordError

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

*Figure III-17 : Validation in Password Text Field*

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



*Figure III-18 : 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 columns grid system based on flexbox. In order to use the grid system, the 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 >

Then 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>

*Figure III-19 : Gird System In* Material-UI

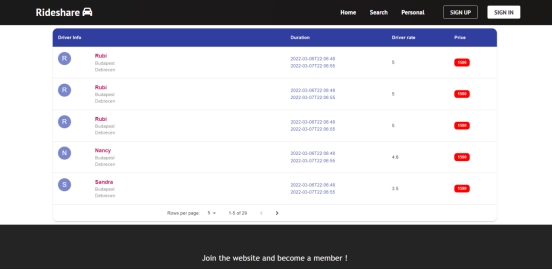
We pass down the item prop to indicate that it is a grid item, xs= {12} represents the 12 columns in our grid system and sm= {6} indicates that we are only going to take 6 of them. Since each button takes 6 of them, two buttons are going to share half of the grid area, as we want them to.



*Figure III-20 : Buttons Using Gird System*

### Table

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



*Figure III-21 : Our Trip Search Result Table*

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>

*Figure III-22 : Table Implementation using* Material-UI

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

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

Table body: the main body of the table. In the table body, we can map each trip info into each table row, and display different data in different table cells. Inside the table cell, we can use 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>

*Figure III-23 : Paginator In Table Footer*

By passing the rowsPerPageOptions prop we can control how many trips we would like to display for each page. They 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 server and make the app react faster as well.

# Conclusion

Throughout the development of this web application. It strengthens my knowledge of Spring Boot and React. It is my first time experiencing full-stack development and would love to learn more.

In this software development, several important technical stacks are used: creating and manipulating PostgreSQL database, mapping entities into the database and adding test data using Liquibase, using MapStruct to map between entities and DTOs, dockerizing the database and application, connecting two sides using open API 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 places to improve with:

1. For now, the register and login control are based on user email and password only, while modern web applications usually provide various login or register options such as using google or other social accounts. Connecting the application to firebase to various authentication methods will be a big improvement.
2. In the trip searching bar, users can type departure place and destination. Even it is implemented to be case-insensitive, the user still needs to spell the name correctly to perform a successful search. We can improve the search bar to make it more functional.
3. Now customers can book trips from the driver after a 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. Customers should be able to cancel the trip that they booked for a certain reason. Furthermore, it would be good if we can support the payment system on the website.

1. 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-database. [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/