



# **Gorju Car Rental Documentation**

## **A CPS2002 Assignment**

### **PREPARED BY**

Julianne Vella - 180103L

Giorgio Grigolo - 041880L

DEC 19, 2022



# Table of Contents

## [1. Task Specification](#)

[Source Control and Delivery Pipeline](#)

## [2. System Architecture](#)

[2.1 Gateway Service](#)

[2.2 Frontend Service](#)

[2.3 Database Service](#)

[2.4 User Service](#)

[2.5 Vehicle Service](#)

[2.6 Booking Service](#)

## [2. System Design](#)

[vehicle-management](#)

[Structure](#)

[Design Patterns](#)

[Singleton](#)

[Iterator](#)

[user-management](#)

[Structure](#)

[Endpoints](#)

[Design Patterns](#)

[Singleton](#)

[booking-management](#)

[Structure](#)

[Endpoints](#)

[Design Patterns](#)

[Façade](#)

## [3. Code Coverage](#)

[vehicle-management](#)

[user-management](#)

[Booking-management](#)

## [4. Self-Evaluation](#)

[References](#)

# 1. Task Specification

The car rental company domain was selected. A resource rental solution was designed and specialised to the relevant domain. This domain involved designing a solution for a car rental company, namely Gorju Car Rental, which rents out different types of Vehicles: namely family cars, motorcycles and commercial vehicles.

The solution was designed with the following features: resource management, customer management, timetabling as well as user interface.

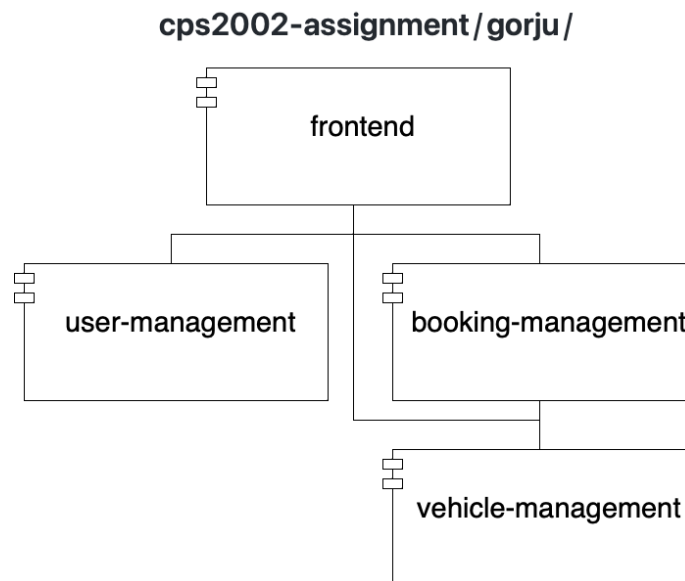
Resource management pertains to creating, reading, updating and deleting vehicles. Vehicles can be queried by colour and availability attributes.

Customer management was implemented through the use of creating, reading, updating and deleting users from the system.

A console application was developed as a user interface to expose the system functionality.

The above was taken into consideration to develop the specification through multiple group discussions. The specification involves 3 microservices, namely the user-management, booking-management and vehicle-management, of which its general form can be seen below.

vehicle-management was designed and developed by Julianne, user-management was designed and developed by Giorgio and booking-management was worked on jointly.



# Source Control and Delivery Pipeline

Github was selected as the version control system provider for this project. The project was created and developed in the following GitHub repository: <https://github.com/giorgio/cps2002-assignment>.

Within the development, different branches were used to ensure features were developed, bugs were fixed and safe experimentation with new ideas took place in a contained area of our repository. Eventually, the branches were merged into the 'main' branch and the project was finalised.

Code commits took place regularly and particular attention was given to the commit message to provide as much of a clear and concise description of the changes committed. It can also be noted that code was pushed and pulled very frequently.

Using a version control system enabled us to work efficiently as a team, track changes we made to our files over time and the ability to go back to a previous version in the case of any issues. Furthermore, peace of mind was instilled as throughout development, the code was stored on a cloud-based service rather than locally which proved useful when writing code on different devices.

To build the Continuous Integration (CI) pipeline, GitHub Actions and the respective configuration files were used. This ensured that upon every push that took place, the test suite of each microservice would be triggered. This made sure that no change made in any of the pushed commits would break the build, or if it did it would be drawn to our attention and it would be fixed in a timely manner.

The aforementioned pipeline can be accessed on: <https://github.com/giorgio/cps2002-assignment/actions>

CI methodology acted towards our line of defence against code with issues. Thus, we were able to appreciate its importance in larger projects involving numerous teams making use of environments beyond the integration environment.

## 2. System Architecture

For this project, we opted to use Docker and Docker Compose for containerization. We chose Docker because it is a popular containerization tool that is easy to use and is well documented.

A brief description of the docker-compose.yml file shall reveal everything about the system: from how the services are built to their allowed interactions.

### 2.1 Gateway Service

We achieve a gateway by deploying nginx as a reverse proxy. The gateway is responsible for routing requests to the appropriate service. It is also responsible for authorization through Cross-Origin Resource Sharing (CORS). CORS is a mechanism that uses additional HTTP headers to services where to accept requests from. In this way, services should only accept requests through the gateway.

#### docker-compose.yml

```
proxy:
  build: ./proxy
  ports:
    - "80:80"
  networks:
    - frontend
    - backend
  depends_on:
    - vehicleapp
    - user_app
    - booking_app
```

The gateway service is connected to the frontend and backend networks, which allows it to communicate with the frontend and backend services making them accessible to the outside world.

Also notice how this service depends on the vehicleapp, user\_app, and booking\_app services. This means that the gateway service will not start until all the API services are running.

## 2.2 Frontend Service

The frontend service is a Python CLI that is responsible for the user interface that uses the widely used requests package.

It is strictly only connected to the frontend network (symbolising the public internet network), which allows it to communicate only with the authorised endpoint, the gateway service.

### docker-compose.yml

```
frontend:
  build: ./gorju/frontend
  tty: true
  stdin_open: true
  environment:
    - PYTHON_ENV=container
  networks:
    - frontend
  depends_on:
    - proxy
```

An environment variable is supplied so that if one attaches to this container, the python script knows how to refer to the proxy service, where the hostname becomes the container name.

It is worth noting that this service depends on the proxy service meaning that the frontend service will not start until the proxy service is running. This is due to the fact that the frontend must make sure that the proxy server is actually running before starting to make any requests!

## 2.3 Database Service

The database service is a MongoDB container. It is a document-oriented NoSQL database that stores data in JSON-like documents.

### docker-compose.yml

```
mongodb:
  image: mongo:latest
  ports:
    - "27017:27017"
  networks:
    - backend
volumes:
  db-data:
```

The MongoDB container is connected to the backend network, which allows the backend serviceS to communicate with the database. It also has a volume attached to it, which allows the database to persist data even if the container is stopped or removed.

The only services that interact with MongoDB, as we will see, are the user service and booking service. This is done through mongoose<sup>1</sup>, an Object Data Modelling library for Node.js that aims to reduce deployment/code complexity, as well as development times by implementing commonly used functions as its builtins.

*"Let's face it, writing MongoDB validation, casting and business logic boilerplate is a drag.  
That's why we wrote Mongoose."*

*Mongoose developers.*

---

<sup>1</sup> <https://mongoosejs.com/>

## 2.4 User Service

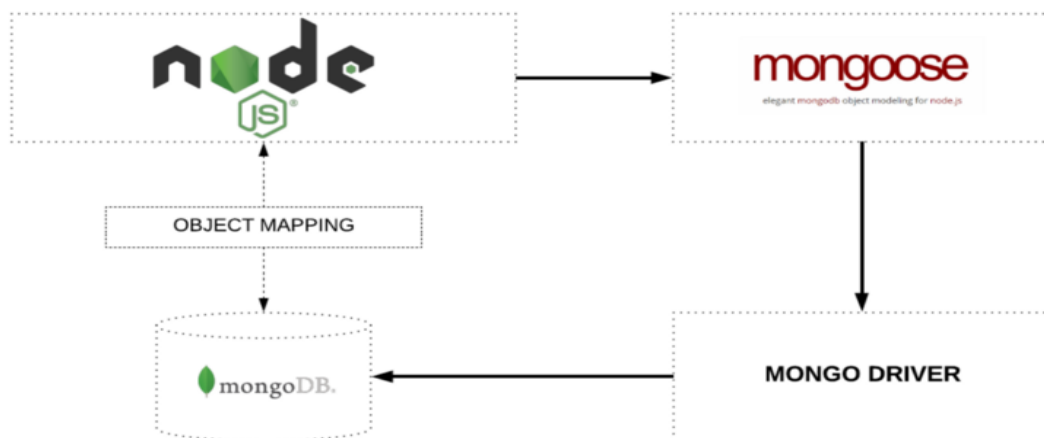
The user service is a Node.js application running on the Express.js framework.

*“Express.js is a small framework that works on top of Node.js web server functionality to simplify its APIs and add helpful new features. It makes it easier to organise your application’s functionality with middleware and routing. It adds helpful utilities to Node.js HTTP objects and facilitates the rendering of dynamic HTTP objects.”<sup>2</sup>*

### docker-compose.yml

```
user_app:
  build: ./gorju/user-management
  environment:
    - SERVICE_PORT=3000
    - DATABASE_URL=mongodb://mongodb:27017/gorju-cars
  networks:
    - backend
  depends_on:
    - mongodb
```

The user service is connected to the backend network, which allows it to communicate with the database service. This service also depends on the database service, meaning that the user service will not start until the database service is running, which again is the logical order of deployment.



<sup>2</sup> <https://www.geeksforgeeks.org/express-js/>



## 2.5 Vehicle Service

The vehicle service is a Java application using the Spring Boot framework. It is managed using the maven build tool. By compiling a standalone jar file, we can run the application using the `java -jar` command inside the container.

### docker-compose.yml

```
vehicleapp:
  build: ./gorju/vehicle-management
  networks:
    - backend
```

### Dockerfile

```
FROM maven:3.6.3-jdk-11-slim
WORKDIR /app/vehicle/
COPY pom.xml ./
RUN mvn dependency:go-offline -B
COPY ./ ./build/
RUN cd build && mvn clean package spring-boot:repackage && mv
target/vehicle-management-1.0-SNAPSHOT.jar ../ && cd ..
CMD ["java", "-jar", "vehicle-management-1.0-SNAPSHOT.jar", "$ARGS"]
```

This service takes control of the main resources of the application. It is responsible for the CRUD operations of the vehicles. It is connected to the backend network, which allows it to communicate with the other services.

## 2.6 Booking Service

Similarly to the user service, the booking service is a Node.js application running on the Express.js framework.

### docker-compose.yml

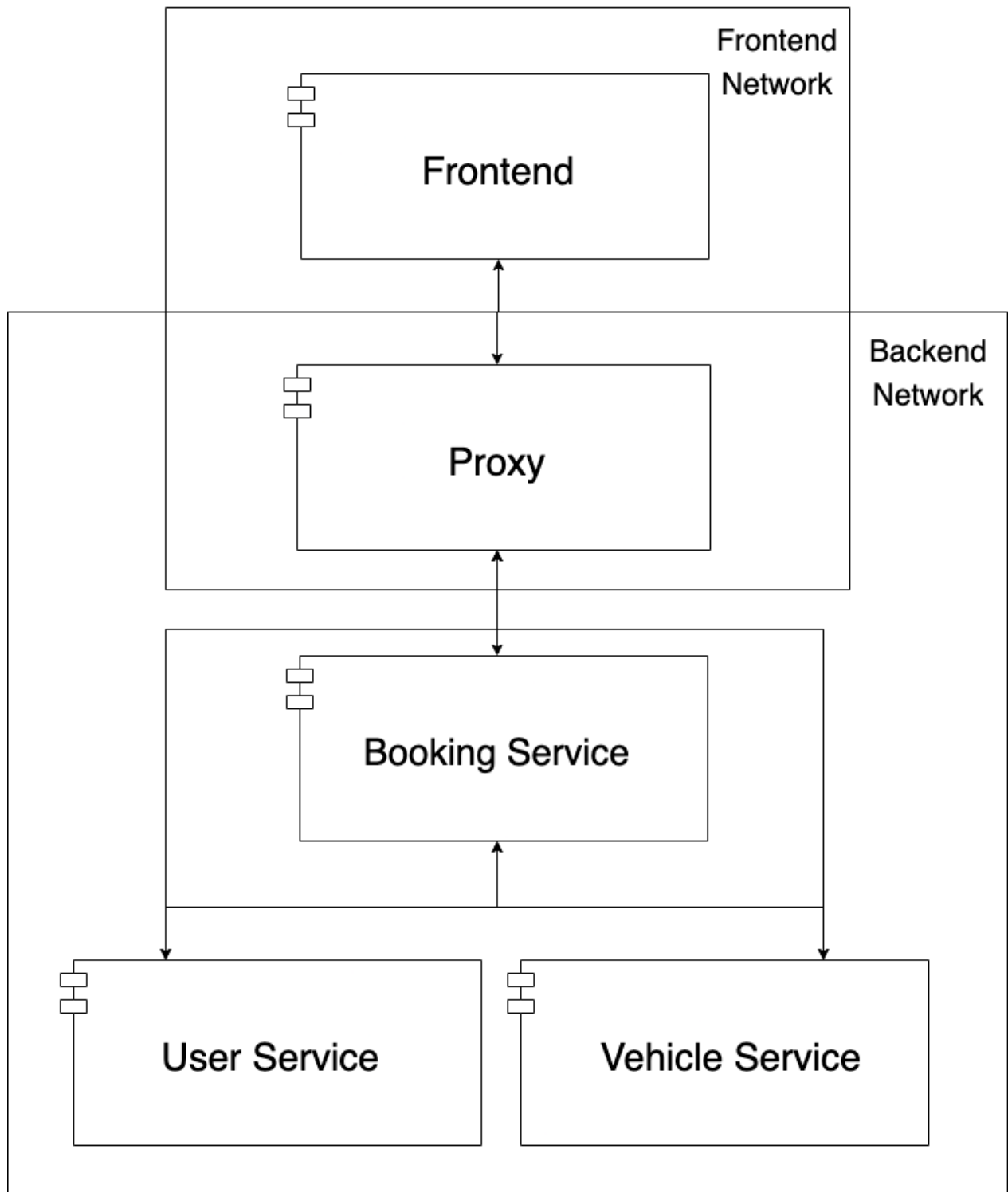
```
booking_app:
  build: ./gorju/booking-management
  environment:
    - SERVICE_PORT=${SERVICE_PORT}
    - DATABASE_URL=mongodb://mongodb:27017/gorju-cars
  networks:
    - backend
  depends_on:
    - mongodb
```

This service serves as a façade for the whole system. Once the independent resources, namely the users and vehicles, are created, the booking service is responsible for the creation of the bookings.

By connecting to the rest of the services, it's also possible to query properties such as vehicle price and availability and user information using only their IDs. This is possible due to it being connected to the backend network, and serving as its glue.

The following graphic summarises all of the above.

## cps2002-assignment/gorju/



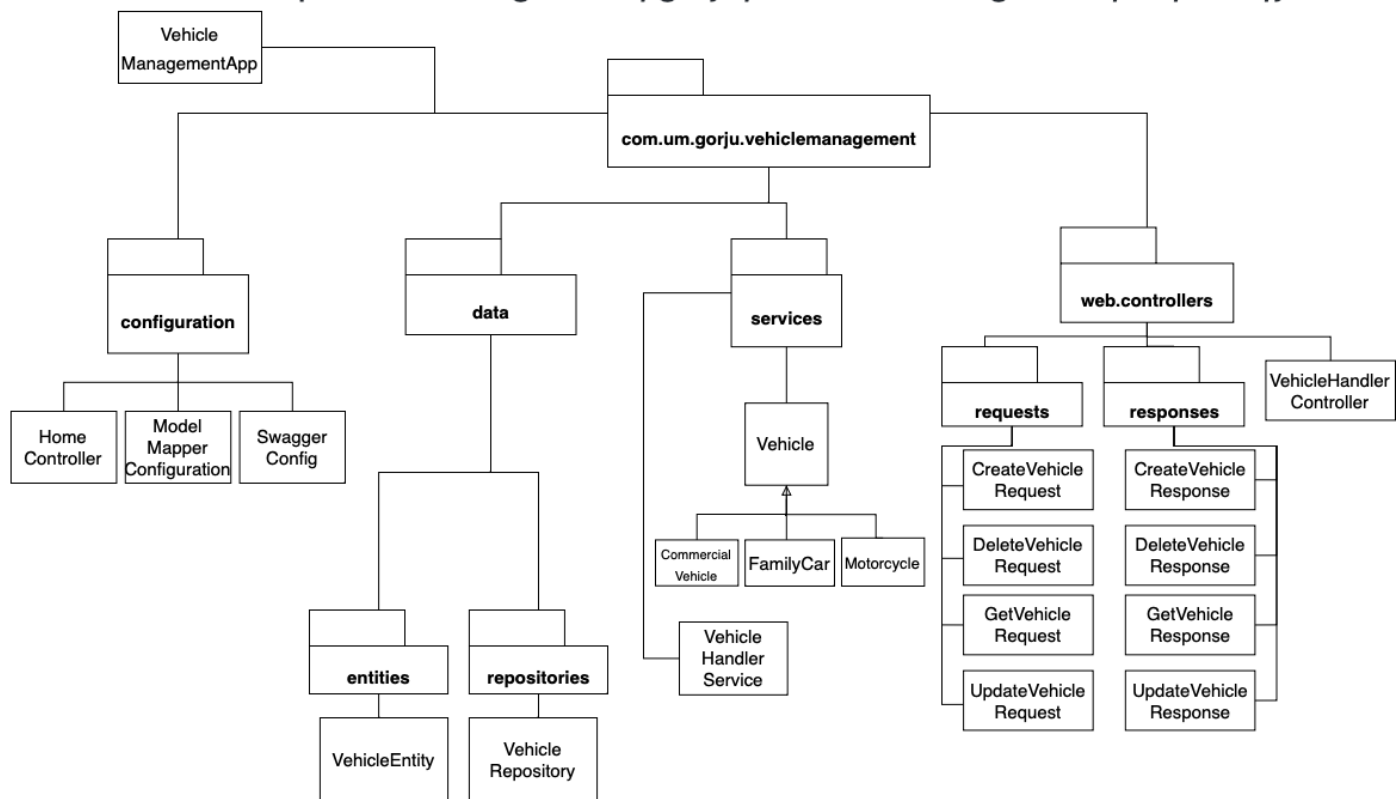
## 2. System Design

### vehicle-management

#### Structure

This microservice was developed using Java. A diagram depicting its layers, namely the data, services and web layers, can be seen below. Each layer resides in a separate package, within `com.um.gorju.vehiclemanagement`.

`cps2002-assignment / gorju / vehicle-management / src / main / java`



The web controller receives and accepts requests from the microservice client validating schema and data. The `VehicleHandlerController` adheres to REST standards by implementing GET, POST, PUT, and DELETE operations. It exposes functionality to create vehicles, update vehicle fields, delete vehicle objects and retrieve all vehicle objects with the ability to retrieve by number plate.

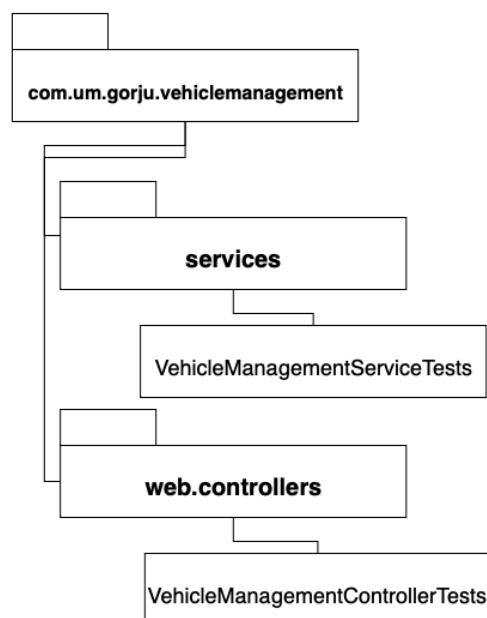
The service layer contains the business logic pertaining to managing vehicles of different types within a car rental company. It interacts both with the data and web layer to ensure communication. It contains a Vehicle parent class, whose children are CommercialVehicle, FamilyCar and Motorcycle and a VehicleHandlerService class that contains the necessary methods to provide operations on vehicles.

The data layer makes data persistence possible through the VehicleRepository and VehicleEntity class that create a form of 'code representation' in the database. The JPA repository gives us the ability to save information in memory as the system is running. In the case of a change in database, the data layer enables this change to take place in a straightforward manner by changing the pom file.

Throughout the development of this microservice, readability and maintainability were kept in mind as attributes for software quality. Although similar models were present across layers, these were kept separate to ensure the software built retains quality over time.

This microservice was developed using test-driven development. Specific attention was paid so that the tests were first written and then code development followed so that the tests passed. Thus, it contains a test suite with 36 tests spread across 2 different classes, namely VehicleManagementServiceTests and VehicleHandlerServiceTests as can be seen in the figure below.

### **cps2002-assignment / gorju / vehicle-management/src/test**



## Design Patterns

### Singleton

The interface `VehicleRepository` extends `JpaRepository`, which by default is Singleton scoped. Furthermore, the use of `@Repository` inside of the aforementioned interface ensured that there is one instance of the database at any time. This can be seen in the code snippet below.

```
@Repository
public interface VehicleRepository extends JpaRepository<VehicleEntity, String>{

    @Query("SELECT vehicle FROM VehicleEntity vehicle WHERE vehicle.numberPlate = ?1")
    VehicleEntity findByNumberPlate(String numberPlate);
}
```

The pattern was selected to control access to a sole instance of the repository as well as preventing unnecessary use of resources as the singleton object is initialised once and does not allow multiple instances.

### Iterator

The `VehicleHandlerService` class makes use of the Iterator design pattern through the interface `java.util.Iterator`. An example of where this design pattern was implemented can be seen in the following code snippet. An instance of `Iterator<VehicleEntity>` was created and `.hasNext()` and `.next()` were used from the interface, implementing the design pattern.

```
public List<Vehicle> getVehicleByColour(String colour){
    List<VehicleEntity> vehicleEntityList = repository.findAll();
    Iterator<VehicleEntity> iterator = vehicleEntityList.listIterator();
    List<Vehicle> matchingVehicles = new ArrayList<>();

    while(iterator.hasNext()){
        VehicleEntity vehicleEntity = iterator.next();
        if(vehicleEntity.getColour().equals(colour)){
            matchingVehicles.add(mapper.map(vehicleEntity, Vehicle.class));
        }
    }
    return matchingVehicles;
}
```

The pattern was selected to be able to traverse a List Collection and process elements inside of it, in a uniform and sequential manner. Using this pattern enabled the code to uphold the Single Responsibility Principle. This is due to the fact that the code was simplified through the use of the algorithms defined in the imported library.

# user-management

## Structure

The user management services encompasses the CRUD operations on the user resource, together with some more features.

With the help of mongoose, a user object is defined as follows:

```
const userSchema = new mongoose.Schema({
  email: {
    type: String,
    unique: true,
    required: true
  },
  name: {
    type: String,
    required: true,
  },
  balance: {
    type: Number,
    default: 100
  }
})
```

## Endpoints

- POST /api/users
  - Expects a JSON body containing name and email. On user creation, balance is automatically set to 100.
  - Returns a response containing the relative creative users' inbuilt id that's generated by mongoose.
- GET /api/users
  - Returns an array of all the user JSON objects currently in the database.x
- GET /api/users/{id}
  - Return the JSON object of the queried user, together with a status code of 200. If the user is not found, a status code of 404 is returned, together with a JSON object with an error message.

- GET /api/users/email/{email}
  - Expects an url query parameter.
  - Returns the user JSON object and a status code of 200 if found or status code 400 if not found.
- PATCH /api/users/{id}
  - Expects a JSON body containing the properties of the user to modify.
    - { name: "John Doe" } in the body will only modify the name property of the specified user given the ID in the url query parameter.
  - Returns a status code of 400 if request is malformed, 500 if there was an internal error inhibiting the modification of the user in the database, 404 if the given ID does not exist and 200 if user updating was successful.
  - The PATCH http verb has been used in contrast to PUT as it can not create a new resource, but only modify those users that already exist in the database.
- DELETE /api/users/{id}
  - This endpoint deletes the indicated user resource from the database and returns status code 204 and an empty body if so.
  - If id is not found, then the endpoint returns an error 404, together with a JSON object containing a descriptive message.
- POST /api/users/{id}/addbalance
  - Through this method, it is possible to increment the balance of user, by the amount defined in the required request body JSON object containing the amount property like so
    - {"amount": 30} will add 30 to the specified user's balance.
  - On success, the response will have a status code of 200, and a body containing the modified user. Otherwise error 404 is propagated to notify that a user has not been found.



The service has been split in various layers, namely

- `user.route.js`
  - defines available endpoints, and HTTP verbs can be used on them.
  - validates requests.
  - calls methods from the `user.controller.js`.
- `user.controller.js`
  - defines the operations that can be performed on the user resource, given specified parameters passed from the web layer.
  - interacts with the database through the data object modelling library.
  - propagates any internal error to the web layer.
- `user.model.js`
  - defines the main resource of the service.
  - validates uniqueness of resources.
  - manages validation of fields.

## Design Patterns

### Singleton

In this service, a singleton design pattern can be observed due to the single connection to the database, created when the service starts in `index.js` as follows:

```
mongoose.connect(process.env.DATABASE_URL, { useNewUrlParser: true,
useUnifiedTopology: true })

db.on('error', (error) => console.error(error))
```

This is desirable as having a lot of connections open concurrently can cause excessive bandwidth usage, as well as unnecessary functional overhead.

# booking-management

## Structure

A booking is defined as follows

```
const bookingSchema = new mongoose.Schema({
  booker_id: {
    type: mongoose.Schema.Types.ObjectId,
    required: true
  },
  number_plate: {
    type: String,
    required: true
  },
  from_date: {
    type: Date,
    required: true,
  },
  to_date: {
    type: Date,
    required: true,
  },
  price: {
    type: Number,
    required: true,
    min: 0,
  },
  paid: {
    type: Boolean,
    default: false
  }
} ...
```

Similar to the user service, this service runs on Node.js using the Express.js framework.

Its main role in the system, besides managing the booking resource, is to communicate with the other services to query information about users (i.e. their balance) and vehicles (i.e. their price). This is done through the utility functions created in the userController and vehicleController respectively.

The requests are initiated using the Axios<sup>3</sup> library, which provides a promise approach to handling functions, and their respective errors that might be thrown.

A promise can be thought of as a function that can resolve in one of two callback branches namely `.then()` or `.catch()`. These two branches are to be respectively reached if the code wrapped by the promise is successfully executed, or if there is an error thrown either manually or from a function within.

This is a good approach to building our system as we need to handle what happens if the requests to the user and vehicle services are successful or not, since the services might throw an unexpected error, crash and refuse to be available for further querying.

## Endpoints

- GET `/api/bookings`
  - Returns a JSON object in the body whose `'bookings'` property contains an array of all stored booking objects
- GET `/api/calendar`
  - Returns a JSON object whose keys are the number plates of all the cars that have a booking. Each key points to a JSON array of objects containing the from and to dates of each booking for that particular vehicle.

```
{
  "ABC122": [
    {
      "from_date": "2222-11-24T00:00:00.000Z",
      "to_date": "2222-11-25T00:00:00.000Z"
    }
  ]
}
```

This would mean that from all bookings, vehicle ABC122 is taken during those dates.

- GET `/api/bookings/calendar/{number_plate}`
  - Returns the same as the above endpoint, but only for a specific vehicle and an error 404 if the number plate is not supplied.
- GET `/api/bookings/{id}`
  - Returns the JSON object of the queried booking, together with a status code of 200. If the booking is not found, a status code of 404 is returned, together with a JSON object with an error message.
- POST `/api/bookings`

---

<sup>3</sup> <https://axios-http.com/>

- Expects booker\_id, number\_plate to be booked, from\_date (dd-mm-YYYY) and to\_date (dd-mm-YYYY). Otherwise, it'll return an error 400, corresponding to a bad request
- Queries vehicle service using the given number plate to determine vehicle availability to create booking.
- Checks existing bookings for date overlaps of bookings with the same number plate.
- Returns a json object containing the newly created booking ID if successful
- **DELETE /api/bookings/{id}**
  - Expects the booker id in the url query parameters, and returns a bad request if not supplied
  - Does not delete the booking entry from the database
  - Deducts balance from the user by sending a PATCH request to the user service, if sufficient, otherwise throws a 406 error, corresponding to a not applicable request.
  - Sets its own paid attribute to true.

## Design Patterns

### Façade

The booking service intrinsically acts as a façade to the other services due to the system structure. When a booking is created, i.e. `createBooking` is called from the `bookingController`, a lot happens behind the scenes. Most notably, the vehicle database is queried, to determine the availability of the vehicle.

```
vehicleController.getVehicle().then((vehicle) => {  
  if (!vehicle.available) {  
    return reject({ code: StatusCodes.BAD_REQUEST, data: 'Vehicle is not available' })  
  }  
})
```

Moreover, the booking model provides a `pay` function, which can be thought of as a class method in Java.

It serves as a front-facing function that exposes the complex execution flow of checking the balance of a user by querying it in real time, as well as deducting it if possible. All the information required to call the `pay` function on a booking resides in the booking itself, further emphasising the stateless aspect of a REST API adhering to its standards.

### 3. Code Coverage

#### vehicle-management

Overall, the test suite of the vehicle-management microservice resulted in a 94% class coverage, a 75% method coverage and a 90% line coverage, as can be seen in the below table.

Element	Class	Method	Line
vehicle-management	94% (18/19)	75% (85/112)	90% (263/290)

Code coverage was analysed for each layer within the microservice, as described in the below 4 tables.

configuration	100% (3/3)	75% (3/4)	94% (17/18)
HomeController	100% (1/1)	0% (0/1)	50% (1/2)
ModelMapperConfiguration	100% (1/1)	100% (1/1)	100% (2/2)
SwaggerConfig	100% (1/1)	100% (2/2)	100% (14/14)

Within the configuration layer, the index() method within the HomeController class was not reached as no test was specified within the vehicle-management test suite. Since the URL mapping is a Spring Boot functionality, it has been assumed that it has been tested upon development of the framework. It is important to note that although the line is not covered, the functionality of the code is correct. This is due to the fact that when running the application typing <http://localhost:9001/gorju> in the browser redirects to <http://localhost:9001/gorju/swagger-ui.html> providing access to the application as required.

```
@Controller
public class HomeController {
    @RequestMapping(value = "/")
    public String index() { return "redirect:/swagger-ui.html"; }
}
```

data	100% (1/1)	100% (17/17)	100% (31/31)
repositories	100% (0/0)	100% (0/0)	100% (0/0)
<i>VehicleRepository</i>	100% (0/0)	100% (0/0)	100% (0/0)
entities	100% (1/1)	100% (17/17)	100% (31/31)
<i>VehicleEntity</i>	100% (1/1)	100% (17/17)	100% (31/31)

services	100% (5/5)	100% (36/36)	100% (147/147)
VehicleHandlerService	100% (1/1)	100% (12/12)	100% (100/100)
Vehicle	100% (1/1)	100% (18/18)	100% (35/35)
Motorcycle	100% (1/1)	100% (2/2)	100% (4/4)
FamilyCar	100% (1/1)	100% (2/2)	100% (4/4)
CommercialVehicle	100% (1/1)	100% (2/2)	100% (4/4)

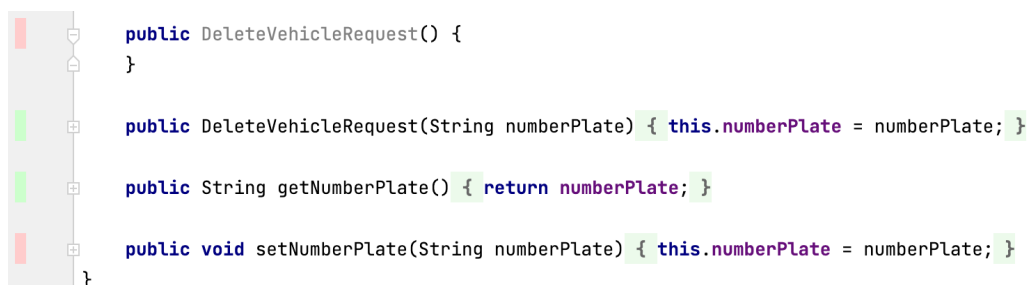
web	88% (8/9)	53% (29/54)	72% (67/92)
controllers	88% (8/9)	53% (29/54)	72% (67/92)
<i>VehicleHandlerController</i>	100% (1/1)	100% (5/5)	100% (20/20)
responses	100% (4/4)	43% (7/16)	52% (10/19)
<u>UpdateVehicleResponse</u>	100% (1/1)	25% (1/4)	40% (2/5)
<u>GetVehicleResponse</u>	100% (1/1)	50% (2/4)	60% (3/5)
<u>DeleteVehicleResponse</u>	100% (1/1)	50% (2/4)	60% (3/5)
<u>CreateVehicleResponse</u>	100% (1/1)	50% (2/4)	50% (2/4)
requests	75% (3/4)	51% (17/33)	69% (37/53)
<u>UpdateVehicleRequest</u>	100% (1/1)	52% (9/17)	74% (23/31)
<u>GetVehicleRequest</u>	0% (0/1)	100% (0/0)	100% (0/0)
<u>DeleteVehicleRequest</u>	100% (1/1)	50% (2/4)	60% (3/5)
<u>CreateVehicleRequest</u>	100% (1/1)	50% (6/12)	64% (11/17)

The web layer was the layer with the lowest code coverage percentages. It was noted that a number of the classes that define the responses and requests contain a number of constructors and methods which are not used within the codebase.

In an attempt to safely delete the aforementioned methods and constructors, the code did not successfully run as they were required by Spring Boot in launching the application.

Thus, these methods cannot be reached by any written tests, resulting in significantly lower code coverage percentages.

An example of such can be seen in the image below:



```

public DeleteVehicleRequest() {
}

public DeleteVehicleRequest(String numberPlate) { this.numberPlate = numberPlate; }

public String getNumberPlate() { return numberPlate; }

public void setNumberPlate(String numberPlate) { this.numberPlate = numberPlate; }

```



## user-management

The library of choice for writing tests in javascript has been Sinon<sup>4</sup> in conjunction with Chai<sup>5</sup> as an assertion library.

The user service has reached complete code coverage due to the fact that it's isolated (i.e. does not query or make requests to other services).

Furthermore, a lot of care has been put into stubbing every possible method, as accurately as possible, to emulate the service's functionality as closely as possible.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
controllers	100	100	100	100	
userController.js	100	100	100	100	
models	100	100	100	100	
userModel.js	100	100	100	100	

The tests can be launched by executing the bash command

`yarn test`

or

`npm run test`

depending on which node package manager is installed. These both translate to

`nyc mocha src/tests/*.test.js`

where **mocha** is used to execute any test file in the `src/test` directory and **nyc** is used to generate the code coverage report.

---

<sup>4</sup> <https://sinonjs.org/>

<sup>5</sup> <https://www.chaijs.com/>

## booking-management

The testing in this service occurs in the same way as per the user service, using the same frameworks.

This time however, even though a fairly high code coverage has been achieved, only 6 lines were not covered by the tests.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	97.71	100	100	97.71	
controllers	96.34	100	100	96.34	
booking.controller.js	100	100	100	100	
user.controller.js	100	100	100	100	
vehicle.controller.js	68.42	100	100	68.42	6-11
models/objects	98.36	100	100	98.36	
booking.model.js	98.36	100	100	98.36	44
models/web	100	100	100	100	
createBookingRequest.js	100	100	100	100	

In the vehicleController, we were unable to stub the Axios get function which sends an HTTP GET request to the vehicle service, querying a vehicle by ID.

Even though this was done successfully for the userController, the test library would throw an error due to the method already being stubbed. This was unacceptable as we surely had to change the mock response, given it's being received from a different service altogether.

## 4. Self-Evaluation

Throughout the development of this Car Rental System, possible future improvements were considered. These involve exposing the services' functionality through a web interface, including the ability for users to log-in and make booking whilst being shown the subset of information relative to them and their bookings - adopting a non-admin/master approach and the ability to query vehicles by more attributes.

Although implemented to a plausible extent, validation can never be enough. It became quite unfeasible to cover all of the branching and possible outcomes of the erroneous responses of the services, and thus the frontend may not account for all possibilities.

When querying the calendar of a vehicle from the booking service, the frontend unnecessarily receives the whole calendar, and then chooses only the vehicle selected by the user. This poses a security risk as a user would still be able to see the calendar of the rest of the vehicles, if they looked at the network traffic.

## References

<https://docs.spring.io/spring-data/jpa/docs/current-SNAPSHOT/reference/html/#query-by-example.running> (Accessed 19.12.22)

<https://refactoring.guru/design-patterns/iterator/java/example> (Accessed 19.12.22)

<https://expressjs.com/> (Accessed 05.01.23)

<https://mongoosejs.com/> (Accessed 05.01.23)

<https://sinonjs.org/> (Accessed 05.01.23)

<https://mochajs.org/> (Accessed 05.01.23)