Car Sharing

Develop a component based Java application based on the Spring Boot Framework implementing a Car-Sharing service for customers (the drivers), fleet manager and tracking-devices integrated in the vehicles.
This specification document contains the requirements for all three practical exercises together, every exercise covers just a part of it.

Requirements

Goals

- Implement a distributed system based on the **Spring Boot** framework
- Use Spring Web for delivering a simple web-gui for human users (drivers and fleet manager)
- Use **Spring Web** for providing a **REST-API** to the external GPS-trackers which are mounted inside the vehicles (the GPS-trackers are not part of the solution)
- Uses a relational-database-system (e.g. **PostgreSQL**) to persist the data.
- Implement a billing-service which will track the costs created by the drivers when using the car-sharing service.
- The Web-/REST-Server and Billing-Server will communicate via **Message Queuing** (e.g. RabbitMQ)
- Execute the external containers (PostgreSQL, RabbitMQ) in the form of **docker containers**.
- use a **logging** framework for system critical messages and warnings at least
- generate your own **unit-tests** with JUnit

Features

Fleet-Manager:

- the fleet-manager can **login** with a username and password
- the manager can **create new vehicles**, which are managed/tracked by the application
- every **vehicle consists of properties**, at least: name, description, current position (latitude, longitude), current state (free, or occupied, or out-of-order), current driver (when occupied), vehicle-token (the unique id and security-key of the vehicle)
- **vehicles are managed in a list**, and can be created, modified, deleted (CRUD)
- the fleet-manager **starts the create invoice-job** for the billing-service (see below)

Customer:

- customers (the driver) can **register** (or sign-up) using at least the following **properties**: username, password, first name, surname, age, driving-license number and a credit-card number
- customers need to **login** before they can use the car-sharing services, there is also a

possibility to **logout**

- customers can **search for free vehicles** by entering the current position as postal-address; the nearest vehicle is returned
- a customer may **occupy the vehicle** selected, from this point of time the traveled distances and costs are tracked; after travelling is done, the driver sets **the vehicle free**
- a customer is not allowed to occupy more than one vehicle at the same time

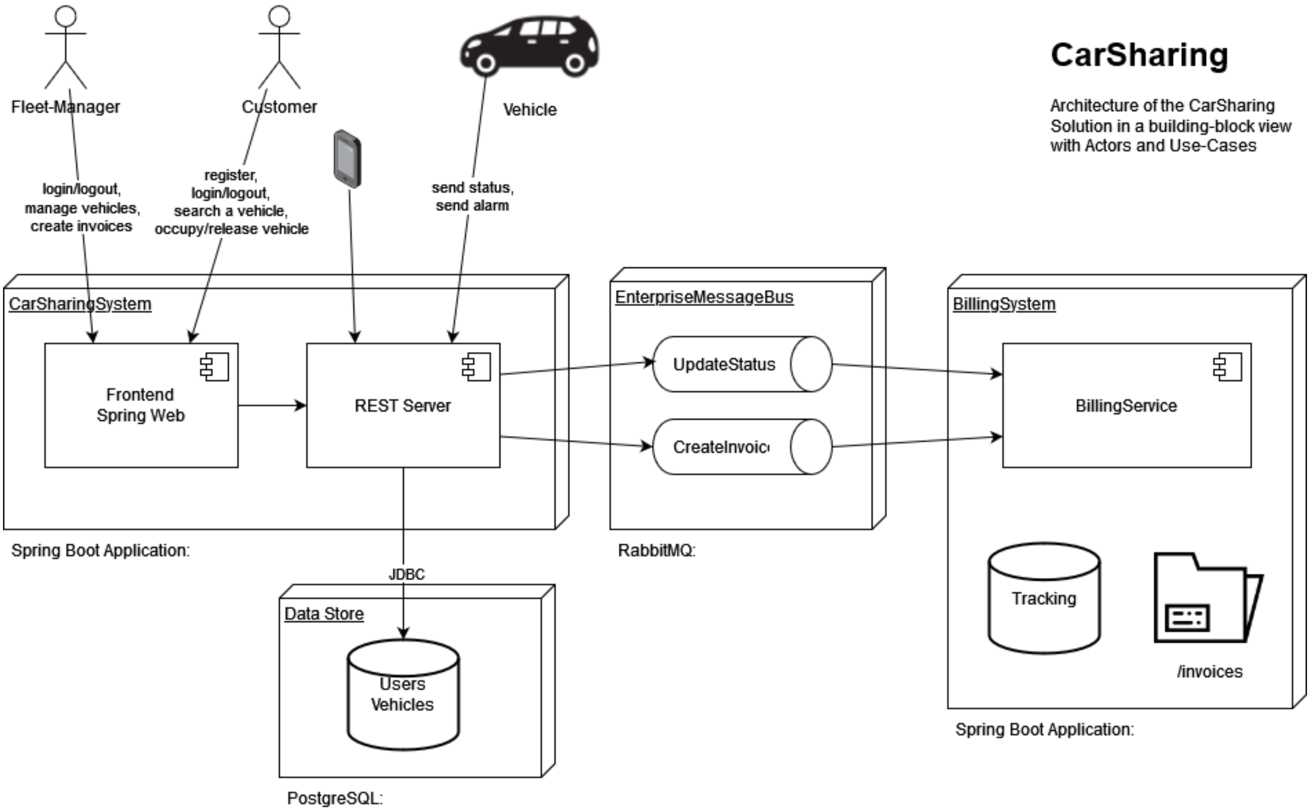
Vehicle:

- the GPS-tracker in the vehicle (not part of the project) will **send its status** to the REST-Server using a unique vehicle-token once per minute
- the **status** contains at least the following **properties**: geo-coordinates (long/lat), current-timestamp, occupy-state, occupied by driver, distance travelled from last status-update, time-duration in seconds from last status-update.
- there is also an **emergency function** integrated, if there is an alarm the tracker will send this alarm to the REST-Server immediately
- the **emergency-information** contains at least the following **properties**: vehicle-status (see above), occupied by driver, priority, emergency-description

Billing-System:

- the billing-system will **collect the status updates** from all vehicles
- on request from the fleet-manager, the billing-system **creates a report** per driver
- the report contains the sum of distances the driver travelled and the calculation of the total-price
- the **total-price** is calculated based on the following formula:
  ```
  <nr-of-trips> * 10€
  + <distance-in-km> * 0,10€
  + <time-duration-in-hours> * 5€
  ```
- the report is **stored as a text-file** in the `/invoices` – directory of the server

Architecture



Exercises

See next pages.

Exercise 1: REST Server

Program a REST-Server with Spring Boot.

Part 1.1: REST Project:

- Create a new Spring Boot Project for the REST-Server part

Part 1.2: User Management:

- Create a controller class for User/Session management (used by the smartphone-app)
- Implement the following endpoints for User/Session management:
    - `POST /api/users/register` – pass the User information in the request body
    - `POST /api/users/login` – passes username and password with basic authentication; will return a token-string in the response body (if successful)
    - `POST /api/users/logout` – takes an authentication token-string; will logout the user
    - `GET /api/users` – takes an authentication token-string; if the user-role is “fleet-manager”, then a list of all users is returned in the response body; otherwise HTTP 403 (forbidden)
- Store all users in memory using a List or Map.

Part 1.3: Vehicle Management:

- Create a controller class for Vehicle management
- Implement the following endpoints for Vehicle management:
  Attention: all requests need to provide a valid authentication-token (bearer) of a logged-in fleet-manager user, otherwise HTTP 403 (forbidden) is returned
    - `POST /api/vehicles` – pass the vehicle information in the request-body. A new vehicle is registered
    - `GET /api/vehicles` – returns a list of all vehicles
    - `GET /api/vehicles/{id}` – returns the vehicle of the id
    - `DELETE /api/vehicles/{id}` – remove the vehicle of the id
- Store all vehicles in memory accordingly.

Exercise 2: Message-Queuing

Implement Message-Queueing with RabbitMQ

Part 2.1: Setup RabbitMQ

- Setup a RabbitMQ-Server running in a docker container. Hint: you may also use the prepared docker-compose.yml file from the RabbitMQEchoService Demo Project.
- Add the required AMQP/RabbitMQ dependencies to the Maven `pom.xml` file.
- Create a RabbitMQConfig class which provides the `@Beans` for the ConnectionFactory and RabbitTemplate. Also create the constants for the queues for `UpdateStatus, Emergency` and `CreateInvoice` and provide them as `@Bean`.

Part 2.2: REST-Server Endpoint for vehicle's status update

- Create a controller class for the vehicle's status updates
- Implement the following endpoints for the updates
  - `POST /api/devices/{`*`vehicle-id`*`}/status` – pass the current status of the vehicle in the request body
  - `POST /api/devices/{`*`vehicle-id`*`}/alarm` – pass the emergency-information in the request body
- The REST-Server checks on incoming requests if the vehicle is found (by vehicle-id) and the vehicle-token is valid; if not a HTTP 401 Unauthorized is returned
- The status-information is immediately put into the `UpdateStatus`-Queue, for delivery via RabbitMQ Server
- The emergency-information is immediately put into the `Emergency`-Queue.

Part 2.3: BillingService to record updates

- Create an extra application for the BillingService. Hint: You may do it as Spring Boot application (then you need an additional java-project) or as normal console-application)
- Add a consumer-method for the `UpdateStatus`-Queue, fetch all the information and just write it to the application log.
- Add a consumer-method for the `Emergency`-Queue, fetch all the information and also write it to the application log.

Part 2.3: Invoice generation

- Create a controller class in the REST Server for invoices:
  - `POST /api/invoices/{`*`user-id`*`}` – authenticated fleet-manager users (see Part 1) are allowed to advice the billing system to create the invoice. The user-id is for whom the invoice should be created.
- The invoice-task start is immediately put into the `CreateInvoice`-Queue.
- Add a consumer-method for the `CreateInvoice`-Queue in the BillingService, fetch all the information, create a fake text file for the bill and store it on the local file system.

Exercise 3: Persistence

Program Entities with JPA & Repositories.

Part 3.1: Setup PostgreSQL

- Setup a PostgreSQL-Server running in a docker container. Hint: you may also use the prepared `docker-compose.yml` file from the HibernateJPA Demo Project.
- Add the required JPA and PostgreSQL dependencies to the Maven `pom.xml` file.
- Create the database on the PostgreSQL server.
- Add database-connection and jpa-settings into the `application.properties` file

Part 3.2: Entities and Repositories

- Make your model classes User and, Vehicle into entities by using the corresponding annotations from the `jakarta.*` package:
    - `@Entity` (for the class),
    - `@Id` (for the field containing the primary-key)
    - `@GeneratedValue(strategy = GenerationType.IDENTITY)`
- Make your repositories `UserRepository` and `VehicleRepository` into JPA-repositories by extending the interface `JpaRepository<…>`. Remark: you will need to take away all unnecessary code, which is automatically provided by Hibernate.

Part 3.3: Write Unit-Tests to check your persistence layer

- Create unit-tests to check that `UserRepository.findByUsername(…)` works correct.
- Create unit-tests to check that `VehicleRepository.findByToken(…)` works correct.