**Documentation - Fuel Station Data Collector - Group R**

# GitHub-Link

<https://github.com/wi22b085/DistributedSystems>

# Software Architecture

The **Spring Boot App** receives the customer ID entered in **JavaFX** via an HTTP request (POST). The program then sends the data read from the Rabbit MQ stream ECHO\_IN\_QUEUE\_ID. After a specified amount of time the Spring Boot App receives a GET request from the Java FX interface. When this method is called is first checks if the file Storage directory has the wanted file. If it exists the path is passed on. If it is not in the wanted directory the program returns a 404 message showing the file is not found.

The data read into the ECHO\_IN\_QUEUE\_ID is then taken by the **Data Collection Dispatcher**. This dispatcher is connected to a database, which contains the URL as well as the longitude and latitude values of each current location. The System is concepted like this, so that if needed other stations can be easily added and the current system can be expanded. An Entity is created to extract the values for each station, which are then stored in a list. The Collection dispatcher goes through each element of the list to extract the url so that the other components can also access the necessary data. For each row in the list a new queue is defined which contains the db-url aswell as the current customer id. This is then sent to the ECHO\_IN\_URL\_ID. Another queue is sent to the Data collection receiver, with a message, which informs that a new job has started. We defined a test to check the passed variables by mocking the RabbitMQ queue and the StationRepository. The database entities are created as server entities and defined with a URL. When the mocked StationRepository calls the findAll() method to retrieve all the stations from the database, we return the predefined server entities. Then, as an act, we call the sendStations function from the GetStations class (with the mocked RabbitMQ template and repository data) using a specified customer ID. After the call, we verify that the findAll() function was called and that the outputs to the ECHO\_IN\_URL\_ID match what should be created, to ensure that our code works.

The **Station Data Collector** consists of a DataSourceConfig, in which the different databases are defined with the url, username and password. When the incoming message from the Data collection dispatcher the customer id is extracted. The remaining string is then taken and checked which database is referenced. All values are then extracted and put into a list. For every result from the sql a new Charge Entity is defined, which consists of the id, kwh(used electricity) and the customer id. When the data is finished extracting the list is then iterated through and the kwh are summed up to build a total for each Database. This value is then passed to the Data Collection Receiver in the ECHO\_OUT\_QUEUE\_VALUE. Furthermore the charging station is also passed on to identify which station has been used for this calculation.

The **Data Collection Receiver** receives for every job four messages from two different queues: One message comes from the Data Collection Dispatcher, that informs that a new job has started. The other three messages come from the Station Data Collector and contain the used kWh of a specific customer for all three charging stations. For all these three messages a key is created, that contains the customer-ID and the charging station. The data is then saved together with the key into a HashMap. If all three keys for a specific customer are present in the HashMap, the data is extracted from the HashMap. Then the corresponding costs are calculated. Finally, all the sorted data is sent together with the current datetime to the next queue and the entries get removed from the HashMap.

The **PDF Generator** receives a message from the Data Collection Receiver. This message contains all the data needed for the PDF-Invoice creation except for the first- and lastname of the customer. Therefore, a CustomerEntity is created with the data of the customerdb-Database to collect the first- and lastname. The database is connected via a Jdbc-Template. Then the actual PDF-Creation function gets invoked and the PDF is created in the defined path with the customer-ID and the date as a name.

# Setup/Installation- and User-Guide

The setup for this project is rather simple. At first the file should be downloaded and the files extracted to a chosen directory. The next step is to start the docker desktop app and navigate the docker-compose.yml located in the project and start all services there. Once that is completed all projects needs to be started. When all projects are running navigate to the JavaFX application. There a popup window should open and the customer can then Input their id into the provided field to get the invoices. Before starting you should also check if the needed ports (8081-8085) are not being used. If needed these can be changed in the application properties of each application. Furthermore the ports in the Docker compose file should also be available to ensure the communication of the different files as well as the databases. Lastly it should be said, that if some projects the datasources were defined as Jdbc-template in a config file, whereas other times the databases were configured in the application properties. The Ports or urls can be changed there, as well as the password or username. This can also be said about Rabbit Mq.

# Unit Testing Decisions

**Spring boot App**

We coded 3 tests in the Springboot App to check all the functionality. The fist test checks if the the methode “getCustomer” works and send the incoming customer id to the fitting Rabbit Mq stream so that the PDF file is created correctly. The other 2 tests check if the “send Invoice” function works correctly. One of the 2 tests checks if the right path is passed when on exists. The second one checks if a 404 Error is passed when the Document is not found.

**Data Collection Dispatcher**

We decided to write a test for the class “GetStations”. The unit test that we wrote mocks the StationRepository (the connection to the db) and injects the mock into the class, as well as the mocked rabbitMq Template. We chose to write this test, to ensure that the created class actually passes all the wanted data to the Station Data Collector.

**Station Data Collector**:

Three test were chosen to be implemented here. All of them test the connection to another db with different amount to also check if the sum is correctly calculated. This can be controlled by mocking the JDBC-Templates and returning a List of ChargeEntities. Furthermore we also check if the calculated price is correctly passed on to the ECHO\_OUT\_QUEUE\_VALUE with the sum, customer id and the id of the station to properly identify which station it came from and keep track of them.

**Data Collection Receiver:**

We decided to run the “sendDataPdf” (has the data for a specific customer from a specific charging station as input) four times: Three times for every charging station of Customer 2 and one time for a different customer. Then we verify, that data would be sent to the next expected queue exactly one time (only happens, if the data for every charging station is present for the specific customer), with the expected message. Finally, we verify that the three entries of that specific customer get removed and the single entry of the different customer persists.

**PDF Generator:**

We decided to verify, that the defined query is executed exactly once on the database via the Jdbc-Template and that the PDF Generator creates a PDF-File with the expected name at the expected location. Furthermore, we verify, that the created PDF-File contains the expected text (e.g. Customer-ID, Name of Customer, Costs). Finally, at the end of the unit test, the created PDF-File gets deleted.

# Lessons Learned

# Tracked Time

|  |  |  |
| --- | --- | --- |
| Feature | Time | Team-Member |
| Create project structure | 1 | Alexander Hickelsberger |
| Code Spring boot App | 3 | Alexander Hickelsberger |
| Code Station Data Collector with tests | 15 | Alexander Hickelsberger |
| Code Data Collection Dispatcher with tests | 8 | Alexander Hickelsberger |
| Code tests for the Spring boot app | 2 | Alexander Hickelsberger |
|  |  |  |
|  |  |  |
|  |  |  |

Reno (vorerst):

Data Collection Receiver: 16h

Test: 6h

PDF Generator: 12h

Test: 4h

Documentation: 6h

Alexander(vorerst)

Zusammenfuegen (2h)

Documentatiosn (3h)