Passenger Management Service Documentation for Usage and Maintenance

Microservices MA2 - MAY.2020

Teacher:
Morten V. Christiansen

Group Members:
Constantin Razvan Tarau
Marius Daniel Munteanu
Paul Panaitescu

Table of contents

I. Project description
II. Tools used
III. Kafka communication
IV. RESTful API communication
V. Database
VI. Setup
VII. Unit Testing

I. Project description

This documentation offers an overview of Passengers Management service for the SD-Air company to help others to use and maintain this service in the future.

II. Tools used

For this project we have used the following tools in order to build the proposed system:

1. **JetBrains Rider** - used to implement the code and for debugging
2. **C#** - our chosen programming language for creating the CRUD API, and Unit Tests
3. **ASP.NET Core** - C# framework used to implement the Web API
4. **MongoDB** - used to build our database
5. **Kafka & ZooKeeper** - used for asynchronous communication between processes (Producer/Publisher)
6. **Postman** - to make HTTP requests to the CRUD API for verification
7. **Docker** - used Docker Images to run the business logic
8. **NUnit** - used this framework to implement Unit Tests

III. Kafka communication

We used Kafka for asynchronous communication between processes. We have implemented a Producer that sends out relevant and updated JSON objects for Passengers when using the Create, Update and Delete operations in the Passenger Management Service.

Relevant services: Accounting and Ticketing
Topics:

1. **PassengerCreateTopic** - Contains records of newly created passengers
    - Publishers: passenger-management
    - Subscribers: relevant services
2. **PassengerUpdateTopic** - Contains records regarding changes to passengers
    - Publishers: passenger-management
    - Subscribers: relevant services
3. **PassengerDeleteTopic** - Contains records regarding passenger deletions
    - Publishers: passenger-management
    - Subscribers: relevant services

Flow:

- Client sends API request
- Service validates and processes request
- Service updates the database
- Service publishes changes in the respective Kafka topic
- Service sends response containing the JSON representation of the passenger from the database

IV. RESTful API communication

We created a RESTful API for synchronous communication between services. We have implemented the API in C# and ASP.NET Core with the following operations:

Action	HTTP method	Relative URI	Returns
Get a passenger	**GET**	/api/passenger/*id*	**JSON of a passenger**
Get all passengers	**GET**	/api/passenger	**JSON list of all passengers**
Get all passengers, including deleted ones	**GET**	/api/passenger/all	**JSON list of all passengers, including deleted ones**
Create a new passenger*	**POST**	/api/passenger	**JSON of newly created passenger**
Update a passenger	**PUT**	/api/passenger/*id*	**JSON of Modified passenger**
Delete a passenger	**DELETE**	/api/passenger/*id*	**JSON of Removed passenger**

* Passing multiple passengers in the same API call is currently not supported.

Other microservices send API requests to passenger-manager, and passenger-manager replies to the respective microservice using http responses. If the API request is Post, Put or Delete, passenger-manager will also publish the change as a record in the respective Kafka topic of each request, as needed.

V. Database

For our database, we choose a document-based database for being easy to use and implement. Specifically, we used MongoDB which is built around JSON documents, with the key-value relationship that is easy to change and maintain.

In the following table are described the fields of the Passenger object and their data type.

Index	Field name	Data Type
1	Id	string
2	Enabled	bool
3	Cpr	string
4	FirstName	string
5	LastName	string
6	Age	decimal
7	Gender	string
8	PassportInfo	string
9	Nationality	string

VI. Setup

Default:
Run `docker-compose up --build` using this docker-compose.yml.
In this setup, the API will listen for Kafka at address `kafka:9092` (`kafka` being the name of the kafka container), but the API will need to be manually linked to Kafka's container using the `--link` flag.

Custom:
The Docker image.
The GitHub repository.

The configuration file can be found at `api/appsettings.json`.
The updated setup can be executed by `docker-compose up --build` using `.../api` as the current working directory.

VII. Unit Testing

The PassengerServiceTest class has the following methods:

- Setup
- Teardown
- TestCreate
- TestRead
- TestReadAll
- TestUpdate
- TestDelete

We have implemented 5 test cases to verify the passenger service functionalities, and 2 methods for setup and teardown.

First we made a separate connection to the database, then a passenger variable with dummy data.

In the Setup() method we created a new instance of the passengerService class.

After each test case, the Teardown() method is called, where we drop the Passenger Collection, so that the following test case uses a newly created collection.

We checked if the API is working properly, by creating test cases for the 5 methods (Create, Read, ReadAll, Update, Delete).

As an example, in the TestCreate() method, we create a new passenger with dummy data, then verify if the object was added to the database and returned properly.