Service-Based WebShop Version 2

This is the workspace for version 2 of the web shop. It consists of several RESTful Java services that primarily communicate via HTTP requests. Some services also use message-based communications to publish and suscribe to events (Apache Kafka).

Directory and Folder Structure

There are three special folders in this directory:

- `_docs`: This folder holds general documentation about the system, namely an architecture diagram of the initial state, another one for the final state, and a document that briefly describes each component in the system.
- `_exercises`: This folder holds the 3 concrete exercise descriptions that have to be performed on the system.
- `_scripts`: This folder holds 2 versions of scripts to build and run the components in the system, `.sh` files for Linux and `.bat` files for Windows.

Moreover, every component in the system has a separate folder in this directory. Apart from the `WebUI`, they are all RESTful services, which is indicated by the `Srv` part of their folder name. Every Java service in this workspace is a Maven project and is defined via its `pom.xml` file. Maven commands like `mvn clean install` are used in the various files in the `_scripts` folder to build the executable `.jar` file for each service. The details for this are documented in the `README.md` file of each service.

RESTful Services Implementation

The services use the Dropwizard Java framework to provide their RESTful HTTP APIs. They have two configuration files (`pom.xml` and `config.yml`) and all follow the same package structure:

- `webshop.{serviceDomain}`
  The top-level package of the service that holds the `ServiceApplication` class which is the entrypoint and the `ServiceConfiguration` class that provides getters and setters for all custom parameters in the `config.yml` file. These two classes will not be modified during the exercises.
- `webshop.{serviceDomain}.api`
  The `api` package holds all model classes of the service, i.e. domain entities as well as request and response classes. Some of these classes may also be present in other services, because they operate on the same entities.
- `webshop.{serviceDomain}.db`
  The `db` package holds a repository class that provides operations to retrieve, store, update, or delete the domain entities the service is responsible for. It acts as the sole access point to a persistent database, although none is present in this prototype implementation.
- `webshop.{serviceDomain}.health`
  The `health` package holds a health check class for operational/administrative purposes. It will not be modified during the exercises.
- `webshop.{serviceDomain}.messaging`
  Only some services have a `messaging` package that holds classes related to communication via Kafka topics. It may contain classes for object (de-)serialization and notifier/listener classes.

- `webshop.{serviceDomain}.resources`
  The `resources` package holds a resource class that specifies all provided REST operations of the service, i.e. its interface. It uses annotations to indicate the path, e.g. `@Path("/products")`, and the HTTP method, e.g. `@GET`. Most changes will have to be performed in these resource classes.

Familiarization Period

Before you start the first exercise of the experiment, take some time (~10-15 minutes) to get familiar with the system, its services, and the build scripts. Here are some suggestions:

- Have a look at the files in the `_docs` folder to get familiar with the overall architecture and functionality of the system.
- Open your Java IDE and ensure that all Maven projects have been successfully imported.
- Have a look at some services in your Java IDE. Try to identify the packages and classes mentioned above. Have a look at `_docs/service_descriptions.pdf` and try to find some of the mentioned resources in the code.
- Start the services by executing all `*srv-run.sh` files in `_scripts`. Start the WebUI as well via `web-ui-build-and-run.sh`. Then, in your browser, navigate to `http://localhost:5000` and check if everything is working as expected. Play around with some of the buttons. Navigate to some of the `GET` resources of some services in your browser, e.g. `http://localhost:8050/products` or `http://localhost:8000/customers`, and have a look at the JSON responses.
- You can already start the exercise validation UI via `../exercise-validation/build-and-run-validation-ui.sh`. It will be available via your browser at `http://localhost:5001`. Just be sure to refresh the start page before you want to begin a new validation, because it needs to verify which version of the system you are running by analyzing the active services.
- Whenever you performed changes in a service and want to test them, be sure to terminate the currently running service instance command line window (Ctrl + c) first. Then use the same script to build and run the service again with your newly implemented changes.

Start of the Experiment

Prepare a stopwatch to track the time needed for each exercise. You can use an arbitrary stopwatch of your choosing, e.g. your watch, your smartphone, or the Google Stopwatch.

When you have familiarized yourself with the system and the experiment administrators are ready and give the signal, start the stopwatch. Then proceed to read `_exercises/exercise01.pdf` and begin the implementation. When you are done, test your implementation via the evaluation UI (`http://localhost:5001`) and, if the tests pass, pause your time. Notify an experiment admin to write down your time. Then proceed with the next exercise in the same fashion. It is important to work on the exercises in the prescribed order, as later exercises build on the results of previous exercises.