**Swisscom Rotterdam Task**

**Date: April 14th**

**Joan Janku**

The purpose of this task, is to build a full stack application with Java and Spring Boot as Back-end and Angular as front-end.

Github Link to Project: <https://github.com/joanjanku2000/sw-service-fullstack>

A screenshot of a computer

AI-generated content may be incorrect.For developing the task, I used the above mentioned technologies alongside docker – with which the “It works on my machine” problem is always solved and makes it easier to manage the dependencies.

**Back – End**

For the back-end, I have created an AppService entity and used **spring-data mongodb to** manage the persistence to the mongo datastore. To setup mongo, I used **docker compose** and access it through either localhost:7017 (defined in docker compose) or internally 27017. The endpoints I implemented are the CRUDs (create, update, findById, findAll and delete). These endpoints consume the main entity – AppService which contains the resources and the resources contains the owners. The find-all endpoint returns the results listed sorted by default. (a **createdDateTime** is stored in the db). However, the update call, can also be used **to update only a specific owner, multiple owners, add/remove owners**. Same is equivalent for resources. The logic is achieved through mappers. Each of the **3 domain** objects **gets assigned a random-uuid on persist**. When an existing record is updated the **uuid** is preserved (similar to a sql database but managed in Java code in this app).

Back-end package structure is shown in the right.

The **back-end is tested with integration tests**, using **MockMvc** for API calls and uses **TestContainers** to setup the mongo database for test purposes only. Integration tests are created as a separate service inside docker-compose and they share the same docker instance with the other containers outside the tests. If I were to run this inside the docker back-end app, it would be a kind of “docker-inside-docker” situation, so for the web-app image, I skip the tests run during the mvn install stage, but the tests are guaranteed to run as a separate service always.

1. **Thread Safe Application**. The endpoint most vulnerable to thread concurrency problems is the “update” endpoint, since at the same time multiple requests can be issued to the same Service to update it. To mitiage this, I have implemented a manual **optimistic locking mechanism** by introducing a “**version**” field. Each time an update is sent the client (front-end) shall provide the version it gets. Only if the version matches the update shall complete. If not it will throw a ConcurrentModificationException. This behaviour is also tested in one of the integration tests, precisely the test named “givenConcurrentModification\_thenOneUpdateFails” which issues the update rest api for the same resource in 2 separate threads. One of them will complete first and bump the version, so the other should fail. This is done in 2 separate threads to simulate a near real-life behaviour, when which request will be consumed first is not guaranteed.
2. **2 Spring Profiles** were configured: Local and Prod. Local profile accesses the local mongo database, and also contains profile specific information for CORS configuration – allowed origins (different for prod and local). Also in the local profile, the in-memory cache used for the application is spring’s default, while in production profile, Redis is used – redis is also set-up using docker.
3. **Cache is configured for both profiles local and prod**. Local uses spring’s default cache implementation and production uses Redis – which runs in the same network that the other containers run from the docker compose. The annotations and mechanism for handling the caches are the same for both profiles. They use Spring’s annotations.

The application has 2 caches, one for find-all operation, and one for findById operation.

The second cache for find-by-id is named **“app-service”** and stores the documents like this

“key=uuid, value=data”. While the first cache (“**app-services**”) holds all results a findAll call returns based on the input hashed. (input for find-all is page input since find-all uses pagination)

*The following table lists the endpoints and the caches that they update and return (where applicable)*

|  |  |  |
| --- | --- | --- |
| Api Call | Updated Cache | Returned from cache |
| POST | Evicts app-services  Adds the resulting record in app-service where key=uuid |  |
| PUT | Evicts app-services  Adds the resulting record in app-service where key=uuid |  |
| findById |  | app-service, key = uuid |
| findAll |  | app-services, key = page input param |
| delete | Evicts app-service (where key=uuid), and evicts all app-services |  |

A **global exception handler** created with @RestControllerAdvice is used to handle all custom exceptions and **Jakarta validation exceptions** inside the app.

I have also configured **logs** to **be stored separately** in a file and they can easily be read using the **tail -f command** inside the container (readme). (*for future development, the logs can be triggered to point to another file outside the project and filebeat can be configured with logstash to index them in a elasticsearch cluster so they can be viewed and filtered in a kibana app)*

**Front-end**

The UI is simple, built with **Angular**, and **consumes** the endpoints generated by **the back-end**, It has **3 main components** (app-service-create which handles create and update forms, app-service-list for find-all and app-service details for findById and pointing to the other update delete operations) and 1 shared header component. **HttpClient** is used to call the **REST APIs** in a service layer. The models are inside a domain layer. The package structure of front-end is shown to the right. It also has 2 profiles (or 2 environment setups) , development (local) and production. For local development simple ng serve command is used. However, for production, the Dockerfile of the front-end specifies production build by using ng build commands and leveraging http-server to display the staticly generated resources inside the dist folder. Simple error handling, showing an error message when APIs fail is done in the interactive components as well.

**Infrastructure**

The applications have both **2 profiles** “local”/”development” and “prod”/”production”.

In order for them to communicate correctly they should be started with the same profile.

Each of them has a **Dockerfile** with the specifcs to build the image and run it. The docker-compose.yml file, sits in the root of both project folders.

For local development, the script “**local\_setup.sh**” is used and does the following:

* Starts mongo db container using docker compose
* Starts back-end container in profile local exposing port 8181 to the outside
* Starts front-end container, building the image first using a Dockerfile.local to differentiate from the way the production distribution is done (inside the Dockerfile)

**Github actions**, a **workflow** has been created for the purpose of this task even though not required. A **deploy.yml** file contains the details of the deployment (jobs). A VPS in Linux **Ubuntu** was created using Hostinger and through **SSH** key, USER and HOST variables , which are stored in the **SECRETS** of the repository and NOT in the code, the triggering of a bash script in the host machine (setup.sh also found in the repo) is triggered, whenever a commit is done – mimicking the **CI/CD** behaviour.

The **setup.sh** script does the following:

* Checks if a directory with the github name exists, if yes it deletes it
* Performs a fresh git clone (repo is public)
* Check for docker installations, if not present it installs docker and docker-compose
* Spins up the docker containers present in the docker-compose.yml and that’s it. Everything dependency downloaded, build, tests are ran and applications deployed. Docker allows the very easily packaging of our apps.

The production version can be shown in the following URL: <http://62.72.33.67:4200/>

So the user getting the code, when developing locally, only has to run the **local\_setup.sh** script. **No node js, or java, or maven is required in the user machine. Only Docker must be installed – for the local development.**

For the production deployment, no need to have anything else besides docker installed in the linux server.

**Typical developer workflow – almost like in a real-world scenario:**

1. Makes a code change including the integration test
2. Runs the local script, checks the test logs (if needed easy port swaps can be done and the apps can be ran from the outside, also tests can be ran directly outside as they are)
3. Tests the app locally
4. Pushes changes to git (if needed can be configured so that the workflow is activated only on master, but as it is, even if other branches are present, only master branch is deployed)
5. After the github workflow finishes couple minutes, they can visit the URL and check the changes.

Below are some screenshots of the running app and github actions, host logs and setup.sh script

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Github Workflows

A screenshot of a computer

AI-generated content may be incorrect.

We can also see the logs when running the script

A screenshot of a computer

AI-generated content may be incorrect.

Logs viewing example in Host machine logging through ssh by public/private key pair and a view of the running containers.

A screenshot of a computer

AI-generated content may be incorrect.

Application Local Startup:

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.