Karlsruhe Institute of Technology

Implementation Report on the microservice-based Variant of CoCoME

Niko Benkler

March 28, 2019

Contents

1 Introduction

This short report summarizes implementation decisions, technology descriptions and other important information we gathered during the implementation of the microservice-based variant of CoCoME. The implementation was based on a practical course by Nils Sommer, who has already created the basic architecture, more precisely the mvn projects for each service, a ORM model for each service and the basic REST communication interfaces for each service. Those services are:

- Store
- Product
- Order
- Report

Regarding the implementation, Sommer already crated a documentation[1] where he introduces the architecture and more important, the glassfish and maven deployment.

This document does not aim to fully describe the architecture with text diagrams and so on. It simply describes each decision that was made during the implementation. The order is completely arbitrary and does not follow a schema.The next chapter summarizes each decision and information as own Section.

[1]https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest/blob/master/doc/report.pdf

2 Design Decisions and General Information

2.1 Documentation and Code

The source code [1] contains as much JavaDocs as possible to describe each class and functionality. The README.md[2] provides the deployment information

2.2 Scripts

We wrote several scripts to start/stop/redeploy/undeploy the whole application or simply single services. The Scripts[3] are available for Linux only, as we implemented and tested the microservice-based variant with Ubuntu 16.04. To execute the scripts, one must adapt the paths to the payara/glassfish folder and the source code folder in each script.

2.3 Payara/Glassfish

As glassfish causes some serious trouble when deploying the application, we decided to migrate to payara (*https://www.payara.fish/*), which is a open source project based on glassfish. The asadmin commands are the same. Simply install the newest variant and replace the system variable, also known as path variable of glassfish.

2.4 H2 Migration

Payara comes with the H2 Database, which is faster and more efficient as the Derby DB we used before. Further, the Derby DB caused some trouble we could not solve. Migration to H2 solved the problem

2.5 Frontend vs Backend Architecture

Each service has its own frontend. The service *Frontend* only uses the iframe technology to include the prevailing frontends of the microservices into a frame. The microservice *Frontend*

[1]https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest

[2]https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest/blob/master/README.md

[3]https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest/tree/master/Skripte

itself handles the login functionality, besides the navigating through and including of the right microservice frontend.

2.6 EJB Backend and Frontend

Each service has stateless EJBs as backend and stateful EJBs for the frontend. So each user gets a new session when using CoCoME (for the frontend) but shares the backend functionality with all current users. For more information please use a search engine to get information about the *@Stateless* and *@SessionScoped* etc. Annotations.

2.7 Domain Entities

Each service, apart from reporting, handles its own domain entities. Several entities have many-to-one relationships, for instance: The enterprise has a list of stores that belong to it. The classes that represent the domain entities have annotations like *@ManyToOne* and *@JoinColumn* so that the database is able to handle those data dependencies efficiently. However, we need to send entities to other services via REST interfaces. Therefore, the interfaces generally convert the domain entities into TO (=Transferable Object, included in the service clients) entities, that do not include a list of other entities, but only their IDs. This was decided due to efficiency. Further, the frontend requires another domain entity format to display the entity information. Therefore, the backend-frontend-query also converts the actual database format into appropriate frontend format.

2.8 Exception Handling REST calls

The exception handling for REST calls raised several difficulties. Usual error handling cannot be applies (e.getMessage() etc.). So each REST Controller throws a predefined *NotFoundException* with an error message. The client catches this exception and reads the message by using *e.getResponse().readEntity(String.class)*. This allows us to send actual error messages from one service to another.

2.9 Frontend

Each service provides a frontend which can be found in project *servicename-rest* where the html files are in *src/main/webapp* and the accompanying frontend logic in the folders *org.cocome.servicename.frontend. ...*

2.10 Double Code

One of the drawbacks of the microservice architecture is double code. Each service that provides a frontend as described in the previous sections needs the login functionality and the main frame of the frontend. Further, each microservice has more or less the same code for navigation (navigation bar) and user authentication.

2.11 CORS Filter

The COARS and X-Frame-Options headers prevent to include iframes from different serves. As the proxy frontend and the frontend of the other microservices run on different payara servers, the browser forbids to include the frontends into the proxy via iframe. This is circumvented by using a COARSE filter that overwrites the headers of each request and response, although this causes a security break. However, it is possible to only allow requests from distinct servers. So it would be possible to limit the legal access only to the server URLs that belong to the cocome system.

2.12 Authentication

The authentication is performed in the proxy frontend microservices. According to the role, the navigation menu is adapted so that a store manager for instance, cannot create and adapt enterprises. However, the microservices need to know which use is currently logged in. For example, the Report Service needs to know whether store manager of store with id 1 or 2 is currently logged in, so that the service can create the correct store report. Therefore, the user data is serialized using JSON via the*window.post-API* and sent to the microservice during navigation. The microservice itself is able to read this information and save it in the current session.

2.13 Roles and Rights Management

The roles and accompanying rights of the initial CoCoME report [4] did not correspond to the former hybrid cloud-based variant of CoCoME [5]. For the microservice variant, we used the roles and rights management that is described in the report.

2.14 Caching

Currently, we use caching to prevent multiple database queries while loading the page. Getters are executed several times by JSF, when the page is loaded. To prevent this, we execute the

[4]https://github.com/cocome-community-case-study/papers
[5]https://github.com/cocome-community-case-study/cocome-cloud-jee-platform-migration

getter only when the session is started. This saves s tremendous amount of queries. However, this comes with the price of data consistency, as data entities created by other users do not appear instantaneously, but only in case of a navigation event (as it causes a page reload).

2.15 REST interfaces

The old REST interfaces have not been complete. Also, error handling did not happen. Now, the REST interfaces provide all necessary functionality and error handling.

2.16 Rollbacks

Multiple databases cause problems, as stated by the *CAP-Theorem.* We did not focus on full database consistency, roll back mechanisms and so on as this was part of the project.

2.17 Roll-In Orders

In the cloud-based variant, ordering products is not implemented correctly. It is possible to edit the stock amount (=order more stock) by simply enter a higher amount of current stock. In contrary, the microservice-based variant put the desired functionality in practice: The stock manager needs to navigate to the order service and places the order. Afterwards, the stock manager needs to roll in the order, which updates the stock corresponding to the ordered amount of stock items.

2.18 Supplier

The microservice-based variant implemented the create/add supplier as desired

2.19 Tests

Tests can only be executed by using the maven command *mvn:test.* The eclipse test environment does not work.

2.20 Several Users

As the frontend is session scoped, it is possible that several users use CoCoME at the same time. Therefore, the microservice variant is closer to the reality.

2.21 Navigation

The displayed navigation bar in the proxy frontend and the microservice frontend depends on the currently logged in user. The role *admin* has all rights. Regarding the store id, one need to be aware of the existing store in advance. So if store with store id 3 does not exist, an error message is thrown if somebody want to login as store manager using store id 3.
The cashier is instantaneously redirected to his cash desk view, as s/he has no other rights like create products...

2.22 Express Mode Policy

The cash desk switches to express mode if the last 4 purchases did not exceed 8 stock items. Afterwards, the cash desk is in express mode which means that only 8 items can be purchased. Being in express mode is indicated by the express light. Further, the cashier is able to switch into standard mode again.