Distributed Applications WS2025/26 (Prof. Dr. Cathrin Möller)

Exercise 12 - 29.01./02.02.2026

Please commit and push to GitLab regularly and ensure @*cathrin.moeller* and my two tutors @pinky-jitendra.tinani and @rahul.patil are invited as a reporter.

Learning goals for today

Today we want to continue practicing what is relevant for an application that serves as a **SaaS**.

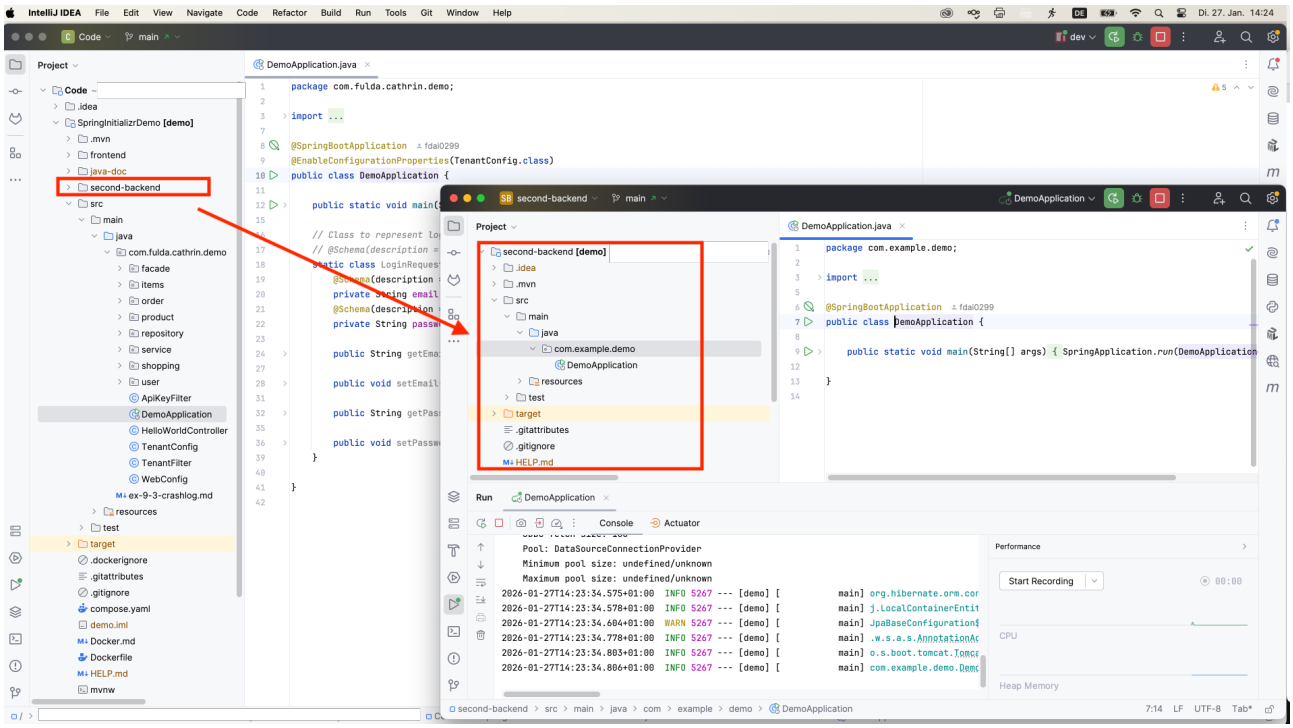
Multi tenancy, configurability and **scalability** are important aspects.

In order to allow scalability of specific parts of an application individually, it usually makes sense to split up a monolithic application into multiple microservices or in multiple tenants that provide physically separated layers.
If divided in microservices, the split is usually done per „domain“ or feature - often reflecting to objects that are stored in 1 table.
In order to do so, very often „refactoring“ is done which implies copy pasting code parts to another app and deleting / fixing references and imports. If a domain has been fully extracted to another microservice, the domain can be deleted from the original code. We will not delete anything today - as this is a lot of effort, but **demonstrate how to allow 2 JavaSpringBoot backends to communicate**.
We also gain some practice in refactoring, extracting code, working with Java lists and ensuring the data model between two backends must be equal (Product(Model).java in this case) in order to merge information. You learn also how to change the port for your running server.
Typically, we have to work with multiple open IDEs at once for distributed backends or frontends. At some point it makes sense containerising the application (e.g. via docker) to stop and start all sub parts in parallel.

Practical Exercises

1) **Create a second Java Spring Boot application** using the Spring Initializer (https://start.spring.io/). Install dependencies for Spring Web, optional: (Spring Data JPA and H2).1
   Open your second backend in another IDE window as it lies in a different folder.

If you want to open all in 1 folder / IDE project, you would have to refactor the folder structure /project/backend-1, /project/backend-2, /project/frontend.
IDE detection is tricky, so if you get stuck, you might build and start from command line (./mvnw clean install && ./mvnw spring-boot:run)



Change the **port** for the second backend in the **application.properties** adding „**server.port=8081**“. This special key is detected by Spring and will automatically change the setting!

Run the frontend and both backend applications and ensure you still see something in your FE fetching data from localhost:8080/saas/catalog and you can open localhost:8081 from the browser. The new backend shall display a „whitelabel error page“.

2) Now let’s **add a REST endpoint to your second backend** that will not be consumed by the frontend, but by the first backend. The goal is the frontend will talk to one backend (running at port 8080) and the first backend will fetch additional information by the second.
Copy paste some product domain content (Category (enum), Product(Model), ProductService, Product Controller) to your second backend - we are interested only in **fetching a hardcoded list of**

products from this second backend. You can delete all tenant information in those files copied to the second backend until the code compiles again. You can also delete all JPA methods and annotations (repository or entity related).
The only service method we are interested in keeping is „**fetchAllProducts**()“ which should **now return a hardcoded list of products**.
The only **@RestController** endpoint we need is a GET endpoint returning all products, e.g. via /**api/catalog**.
Clean up the code and **create an ArrayList of 5 hardcoded products within the ProductService**. The **goal is that opening a web browser, localhost:8081/api/catalog** returns a **JSON with 5 hardcoded products**. Change the product names, e.g. to „Fancy T-Shirt“, „Fancy Socks“ - so you will be able to distinguish them from the products of the other backend later.
This should simulate a scenario that an e-commerce webshop is incorporating a product feed from an externalized service. We just created the externalized service and will call it from our e-commerce application later.

3) Now we want to get that JSON via our SaaS controller in our first backend and deliver the merged content to our frontend. Spring allows injecting a bean of type „***RestTemplate***“ which we can use to fetch data from other endpoints.
The import is from the Spring Web package:
org.springframework.web.client

Create a file ***RestConfig.java*** and copy this:

```
@Configuration
public class RestConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
```

```
    }
}
```

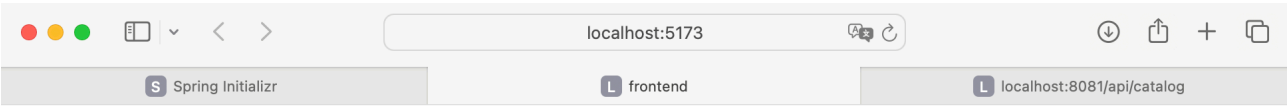
Now you can **inject the dependency of that RestTemplate** into a service, e.g. ExternalProductService (called by SaaSCatalogController) **and fetch data** like this:

```
List<Product> productsFromOtherBackend = new ArrayList<>();

String url = "MY_BACKEND_URL";

productsFromOtherBackend = Arrays.asList(restTemplate.getForObject(url,
Product[].class));
```

Ensure you have no typo in the url as e.g. an extra slash at the end causes a 404!

Merge the two product lists from your first and second backend (Java Lists have the method ***addAll***()) and ensure your SaaS endpoint delivers the complete list of products to your frontend.

The frontend should look now similar (or prettier) to this:



Product Catalog

Trousers Black M 19.99€
T-shirt Blue M 19.99€
Jeans Blue 34 49.99€
Trousers Black M 19.99€
T-shirt Blue M 19.99€
Jeans Blue 34 49.99€
Fancy T-shirt Black M 19.99€
Fancy Hat Blue M 19.99€
Fancy T-shirt Red M 19.99€
Fancy T-shirt Green M 19.99€
Fancy T-shirt Yellow M 19.99€

Info: In real world projects the approach of using only 1 backend as access entry point for your frontend is often used in multi-tenant layered architectures.
Using a Backend for Frontend (BFF) allows you e.g. to protect your second backend - including its API Key - to not need to be known by the frontend. All code in the browser is public, even if it was not pushed to GitHub, but inserted from a cloud secret, e.g. when building or deploying the frontend.

If your second backend had an API Key as well, only the first backend would need to configure it. Not the frontend.