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

Exercise 11 - 22./26.01.2026

Do not hesitate to ask questions or if you want feedback for your coding style.

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.

Please do not forget to commit your code and push to GitLab!

Learning goals for today

Today we want to continue practicing what is relevant for an application that serves as a SaaS.
A SaaS should be reusable for multiple customers and be optimised in a way we can share resources. This could imply sharing a database.
Multi tenancy, configurability and scalability are important aspects. A big challenge is usually avoiding parsing errors to read configuration strings into a map and typecasting between enums and String.

We continue coding on our Spring Boot webshop application and learn how to configure tenant settings externally to avoid the need for Java Code changes, if new tenants are added or their configuration needs to be changed. Apart from **@Value** to read external config values, the concept of a configuration file using **@Configuration, @ConfigurationProperties** and **@EnableConfigurationProperties** is used. We will also work with additional **filters** to intercept incoming requests.

Practical Exercises

1) We want to get hands-on experience on multi-tenant architecture. One way to deploy our SaaS only once and deliver customised content for each tenant or client, is using tenant IDs, e.g. using a HTTP header or a request parameter to identify the client needs.
   Add two tenant IDs to your application configuration. As config values are usually strings, you might need to **separate possible values with a comma or semicolon**. E.g. „allowed.tenants=shop-123,shop-abc “. Add an **additional TenantFilter** in your Spring Boot application (on top of the API Key filter) that checks for a registered tenant ID in the HTTP

headers. Filters in Spring act like Interceptors. Any incoming HTTP requests might be analysed and the HTTP responses can be modified to set e.g. the response status to Unauthorised.
If you **extend the GenericFilterBean and add @Component to your Filter**, the filter is automatically registered and will be considered in your Spring app. Override the doFilter method. The **TenantFilter** should read the allowed tenants from the external application config, needs to **parse the string** and add values to a list to be able to compare them to the received request header value from the frontend.
Change your external frontend (Svelte or sth else) to send the tenant ID as a second HTTP Header alongside the API-Key.
Test that the filter works by fetching data from FE without the tenant Id (no content returned) versus with a correct (backend side configured) tenant Id.

2) We want to customise our application for different tenants (customers, consumers, shops that make use of our SaaS). Imagine one retailer is only interested in products that are for sale, others might be not allowed to see sale products, others see all products.
Add an **enum** file **Category.java** and an attribute to your products in the database, which marks the products in your catalog to be ***sale*** or ***standard***. Implement the list of possible values as an enum and use the enum values as a reference. For simplification, your product database can store the category information as a String. But the String should be matching a value in your enum. Ideally, you just have to touch **Product.java** for that change.
If you added the category reference attribute you can define a constructor without the category information - setting internally a default value for category (=standard). To customise products for a different category, you need to have a constructor that allows passing the category.
Another file you now have to touch for the change is your **@Configuration** bean that populates your database (might be named

LoadProductDatabase.java). Existing entries in your DB will have „null“ for category. Change the creation script to add some „sale“ and some „standard“ products using the type-safe enum values => Category.SALE Consider deleting and re-populating your database if you want to get rid of the null values. Deleting the H2 DB is as simple as just deleting the DB file.

3) Let’s imagine one of your tenants (the first shop) is interested only in clothes which are flagged with ***sale***. The other tenant is interested in (and allowed to see) ***standard*** clothes.
   Add a **mapping** to the **externalised configuration file** of your Spring Boot application to configure a mapping for which tenant id, which category is relevant.

```
e.g.
tenant.mapping.shop-123=sale
tenant.mapping.shop-abc=standard
```

The idea behind this approach is to make your SaaS ***configurable*** in a way that only application.properties / external helm files for Kubernetes deployments have to be touched if you need to change the configuration or add new tenants. No Java code has to be touched.

You can create a **TenantConfig** class, annotated with

```
@Configuration
@ConfigurationProperties(prefix = "tenant")
```

which can read the externalised config and automatically create a map for the tenantId to gender information. „mapping“ needs to be named equal in properties and in class for the „magic“ to work.

```
private Map<String, String> mapping;
public Map<String, String> getMapping() {
    return mapping;
}
public void setMapping(Map<String, String> mapping) {
```

```
    this.mapping = mapping;
}
```

The only thing you need to do on top is allowing the TenantConfig class to read your global application.properties. You can configure that in your app root using **@EnableConfigurationProperties**:

```
@SpringBootApplication
@EnableConfigurationProperties(TenantConfig.class)
public class DemoApplication {
```

Autowire the TenantConfig in your ProductService and implement a logic in the SaaS controller to provide only that subset of products with the catalog endpoint that matches the interest of your tenant.
You can read HTTP Request headers in your controller endpoint using **@RequestHeader**("X-TENANT-ID") String tenantId.

You might need a new **ProductService** method, a new ***@NamedQuery*** and a new **ProductRepository** method to filter by category and you need to be careful with typecasting between the enum and a String value.
Hint: you could use `String categoryString = tenantConfig.getMapping().get(tenantId);`

As a result your FE should now show only ***sale*** or only ***standard*** products, being able to toggle this, by switching the tenant-id sent as request header.

Debug the network request response for received products or change the config between ***sale*** and ***standard*** for the tenantId you are sending from the frontend to see it works properly.