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

Exercise 10 - 15./19.01.2026

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 practice what is relevant for an application that serves as a **SaaS**. We want to protect access and make it configurable to **customize** it for more than one customer. We also want to split frontend and backend to get a more distributed system.

We continue coding on our Spring Boot webshop application.

Modern web APIs, especially those, which are intended to be used by others, are often documented using tools like **swagger UI**. This is a more convenient documentation approach that explains REST endpoints including methods and JSON schemas for data. We will learn how to include such documentation as well.

Practical Exercises

1) We want to extract the frontend from our monolithic application and just consume data from our backend endpoints.

You should have a ProductCatalogController.java or a CatalogController.java using the annotation ***@Controller*** which returns an evaluated Model and View (Thymeleaf template with data).
Copy that existing catalog endpoint to a new Controller file (e.g. SaaSCatalogController.java) to define a controller annotated with ***@RestController*** (shortcut using @Controller and @ResponseBody capabilities).
The new catalog endpoint should return products in the catalog as JSON (List<Product>). You might prefix the path with /saas or /api to not have a conflict in URL mapping.

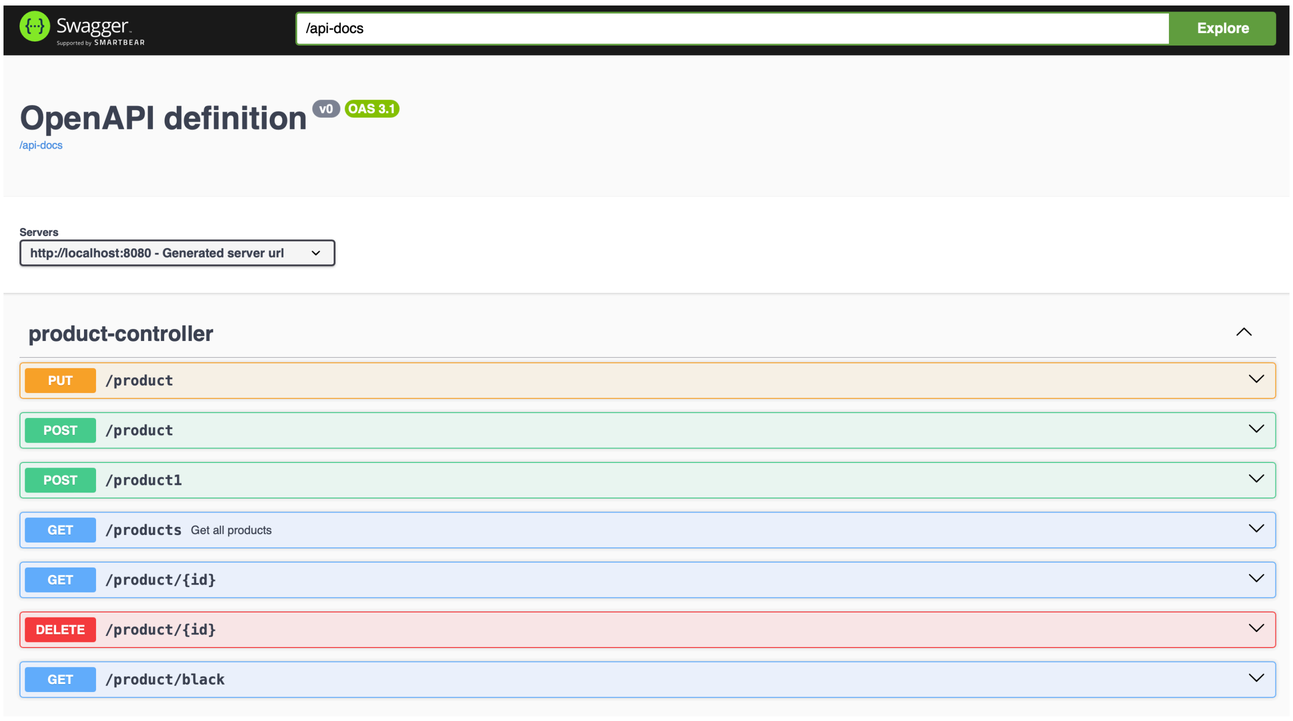
Run your application and ensure you can see the JSON in the browser opening http://localhost:8080/saas/catalog

2) Now make sure your endpoint is documented to external users via Swagger UI. This way, the API can be understood in terms of existing URL paths, example response values and the JSON schema of the responses. Knowing which attributes are returned or expected (from a POST call) helps connected a frontend or another backend to that external API. Even before you might get a licence or access to the API (via API key e.g.).
In order to benefit from automated swagger UI documentation, we need to install a dependency from springdoc.

Add this to the pom.xml and make sure it is installed in your project in your IDE (I needed an IntelliJ restart and './mvnw clean install', sometimes clearing caches can avoid the restart need)

```
<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.8.14</version>

</dependency>
```

Now running your app you should be able to visit localhost:8080/swagger-ui/index.html to see the API docs:

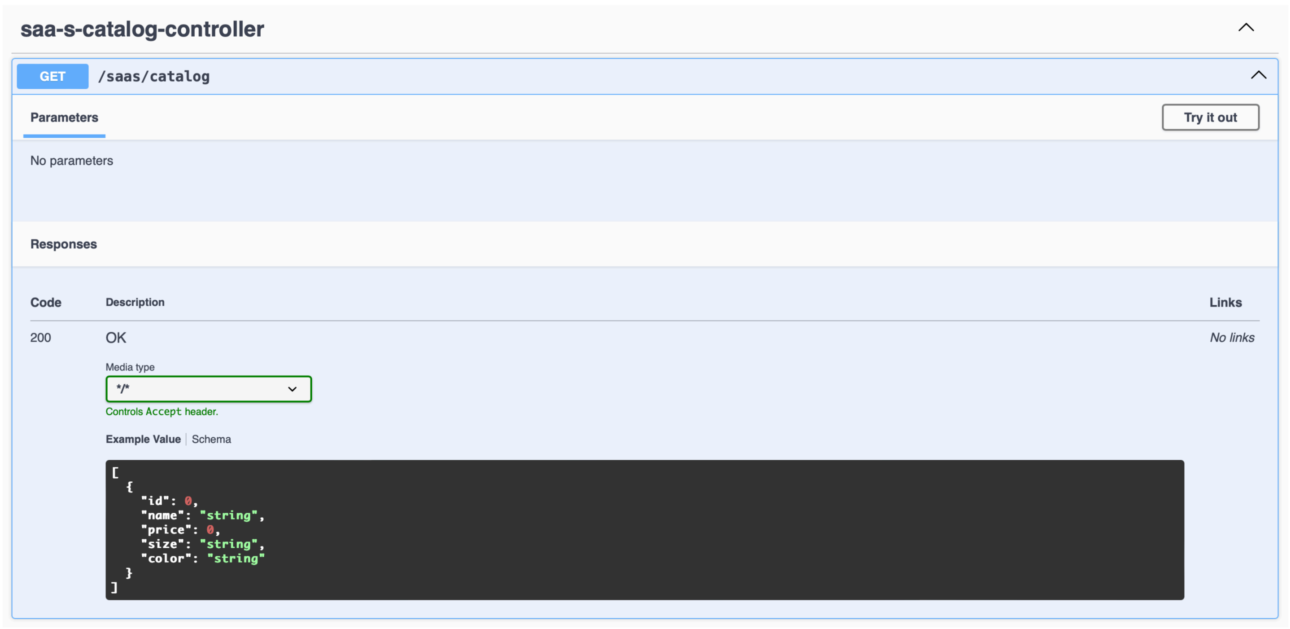


OpenAPI definition v0 OAS 3.1
/api-docs

Servers
http://localhost:8080 - Generated server url

product-controller

PUT	/product
POST	/product
POST	/product1
GET	/products Get all products
GET	/product/{id}
DELETE	/product/{id}
GET	/product/black


saa-s-catalog-controller

GET /saas/catalog

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	OK	*No links*

Media type: */*

Controls Accept header.

Example Value | Schema

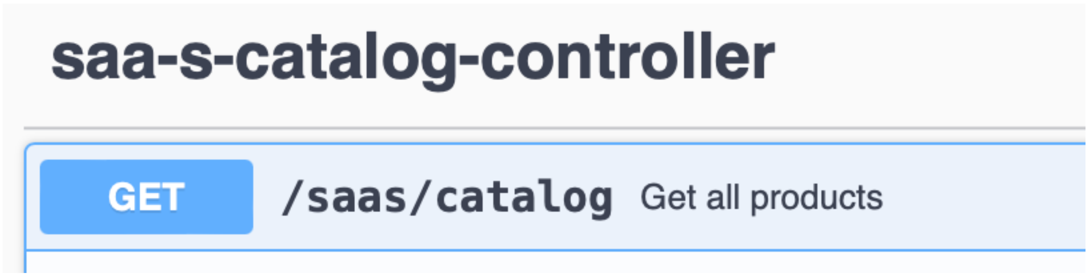
```
[
  {
    "id": 0,
    "name": "string",
    "price": 0,
    "size": "string",
    "color": "string"
  }
]
```

This information can be enriched manually writing additional documentation to the code, e.g. you can use ***@Schema*** to define JSON schemas or ***@ApiResponses*** to explain all status codes returned by a controller endpoint.

The endpoint can also get a description using ***@Operation*** (import io.swagger.v3.oas.annotations.Operation;) on top of a @GetMapping:

```
@Operation(summary = "Get all products")
@GetMapping("/saas/catalog")
```

Try that out and you will see it appear after restarting your app:


saa-s-catalog-controller

GET /saas/catalog Get all products

3) Now we want to consume the SaaS API from outside our still monolithic JavaSpringBoot application. Create an **external frontend application** that will later call this SaaS product catalog GET endpoint. A super lightweight

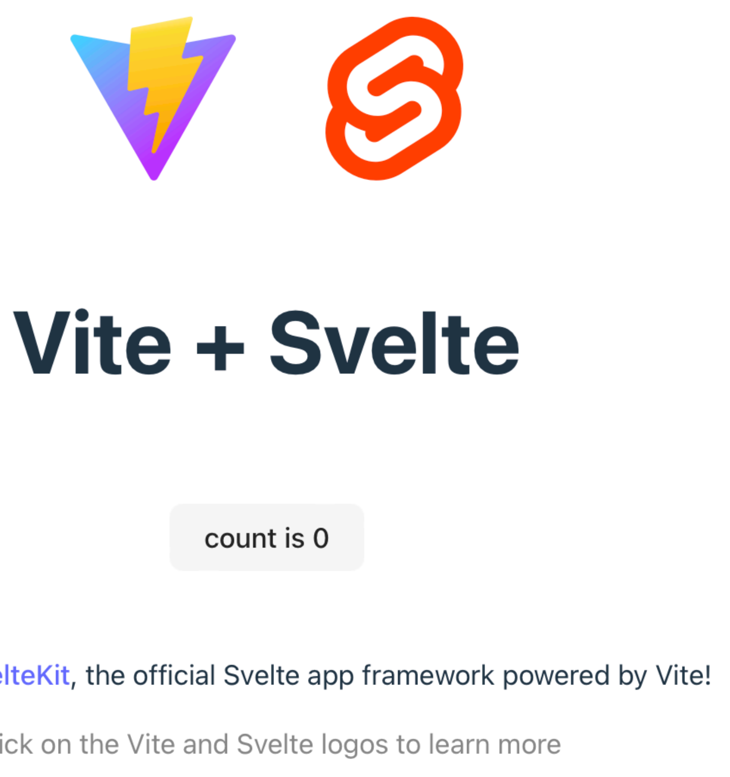
frontend technology which is more stable than just plain HTML+JS+CSS (comes also with its own server) is **Svelte**.
Create a „Hello World“ application within a **/frontend** folder executing this from root folder (e.g. SpringInitializrDemo) via command line:

```
npm create vite@latest frontend -- --template svelte
cd frontend
npm install
npm run dev
```

Your app will run under localhost:5173

If you are very familiar with Angular / React / Vue, you can do so alternatively. Be aware that Svelte might be just more lightweight for our demo purposes.
For more info, visit the official Svelte docs: **https://svelte.dev/**

If successful, you will see this hello world page:



Vite + Svelte

Check out SvelteKit, the official Svelte app framework powered by Vite!

Click on the Vite and Svelte logos to learn more

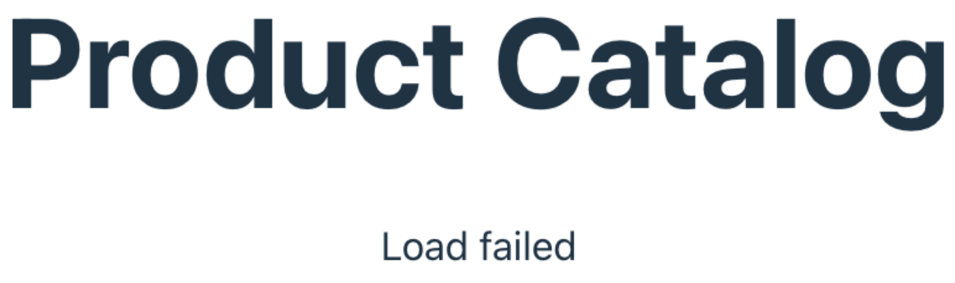
4) We want to see the products in our external frontend. Modify the App.svelte file copy pasting this (or implement in the SPA framework of your choice):

```
 <script>
      import { onMount } from 'svelte';
      let products = [];
      let error = null;
      const catalogUrl = 'http://localhost:8080/saas/catalog';
      async function fetchProducts() {
          try {
              const response = await fetch(catalogUrl, {
                  method: 'GET',
                  headers: {
                      'Content-Type': 'application/json'                  }
              });

              if (!response.ok) {
                  throw new Error(`Error fetching products: $
{response.statusText}`);
              }
              products = await response.json();
          } catch (err) {
              error = err.message;
          }
      }

      onMount(() => {
          fetchProducts();
      });
  </script>

  <main>
      <h1>Product Catalog</h1>
      {#if error}
          <p class="error">{error}</p>
```

```
        {:else if products.length === 0}
            <p>Loading products...</p>
        {:else}
            <div>
                {#each products as product}
                    <div>
                  <span>{product.name}</span>
<!--                  // TODO: copy paste from thymeleaf template to display
all product attributes, ignore the links / buttons -->
</div>
                {/each}
            </div>
        {/if}
    </main>

    <style>
        main {
            padding: 50px;
        }
        .error {
            color: red;
        }
    </style>
```

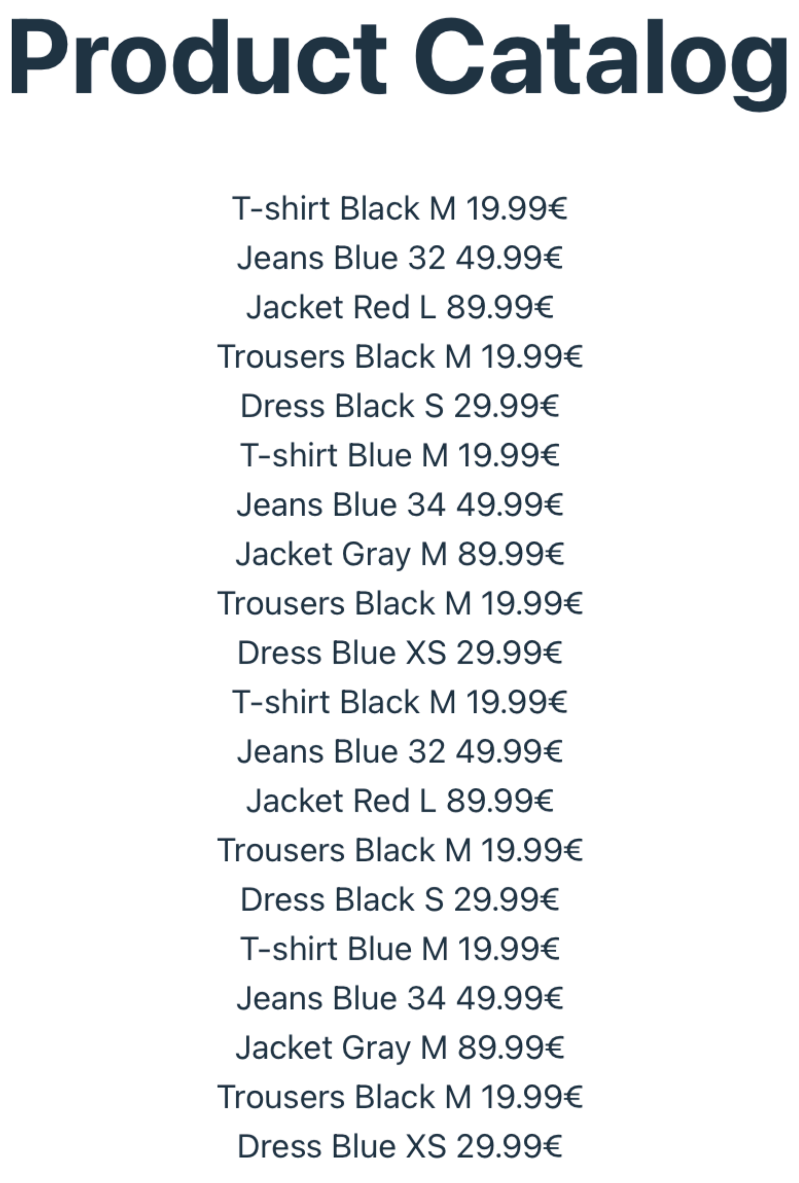
Without CORS setting, your Backend running under port 8080 will disallow the frontend running under port 5173. We will fix this in next step.

Result will look like this:


Product Catalog

Load failed

5) We need to fix the CORS setting. Copy paste the **public class WebConfig** from the lecture to your Spring app and re-run. Continue fixing the template that the products are displayed nicely - as before using your Thymeleaf template. Feel free to make it look prettier than my solution:



Product Catalog

T-shirt Black M 19.99€
Jeans Blue 32 49.99€
Jacket Red L 89.99€
Trousers Black M 19.99€
Dress Black S 29.99€
T-shirt Blue M 19.99€
Jeans Blue 34 49.99€
Jacket Gray M 89.99€
Trousers Black M 19.99€
Dress Blue XS 29.99€
T-shirt Black M 19.99€
Jeans Blue 32 49.99€
Jacket Red L 89.99€
Trousers Black M 19.99€
Dress Black S 29.99€
T-shirt Blue M 19.99€
Jeans Blue 34 49.99€
Jacket Gray M 89.99€
Trousers Black M 19.99€
Dress Blue XS 29.99€

6) Besides configuring **CORS**, a SaaS could use improved protection so not every frontend / application can embed / call our service and we can distinguish between different licenses for the same frontend URL.

Add an **API Key** to protect our SaaS backend endpoint for the /catalog GET request. In Spring the usual pattern to check for an API Key is validating it by a security „**Filter**“ which is automatically called if it extends the **GenericFilterBean** and it is annotated using **@Component**. You can check the path for a defined URL pattern that matches our to be protected endpoints. The Filter acts then as security layer / middleware between incoming requests and our Controllers. Copy the solution from the lecture slides for the **ApiKeyFilter** and read the API Key Value from the application properties file (you will need to revisit ***@Value*** which you used before to read voucher values).

Info: Consider that in a real world example the API Key is usually not hardcoded and not pushed to GIT, but read from a Cloud storage Secret. Just for demonstration purposes, please configure and push the value within the application.properties. For demonstration, it is also allowed, that the frontend just sends it as a hardcoded request header.

Check that once the protection is in place in the backend, the frontend can only fetch the data by sending the HTTP header with the correct API key!

In the frontend you can send it with the fetch request headers adding:

```
headers: {
    'X-API-KEY': ‚YOUR_SECRET_DEFINED_IN_PROPERTIES‘
}
```