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

Exercise 2 - 30.10/03.11.2025

Practical Exercises

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.

Learning goals for today

Today we want to learn using all the CRUD methods from SpringWeb with controller endpoints. We will also trigger all endpoints from a static HTML file using a plain HTML form (and separate @RequestParam values) vs using the fetch API (with a <script> in the HTML file) to benefit using @RequestBody for a complete object.

At least from backend (controller) side, DELETE and PUT methods will be implemented - they are testable via the fetch API.

Practical Exercises

1) Ensure you did the exercises from the two previous weeks to have the starter code for this exercise. If not, please revisit those to get a basic SpringBoot / SpringWeb application.

2) Extend the Webshop application from the last weeks. In our **ProductController** class there should not only be GET request mappings to retrieve existing products but we also want to create new ones via **POST**.

Add a **@PostMapping** Controller endpoint which can act on requests to add further products to your existing hardcoded list. It should use the ProductService to deal with details like ensuring the added product is no duplicate.

There are two options sending data (which is more than 1 field) in Spring POST requests

a) you can add **@RequestParam** to handover the attributes via request parameter (?x=b)

We did that last week for the UserController.

If you define the types for the attributes (String, Long, Double, ...), Spring will automatically try to convert anything received.

Add a **HTML form** to a new static HTML file to actually call the POST endpoint. You can link this page from your dashboard for convenience. We need now a separate input field for each of the product attributes. Do not add a <script> in this exercise, the goal is to use only plain HTML.

How would the complete request URL look like?

Hint: You cannot easily check in your browser devtools network tab because a redirect is happening with the returned response..

Commit and push your code.

b) You can add the content inside the body of the request to not create very long URLs with many parameters. It also helps shortening the code in our Controller endpoint.

Create a second **@PostMapping** Controller endpoint and use the annotation **@RequestBody** in front of the model object parameter in the method arguments. The Model parameter will be the object type Product. This way all potential attributes are automatically mapped if the request body is matching the class attributes. Code will be much more readable in the controller method.

Hint: you need to create a different URL for the second PostMapping, otherwise the Spring application will crash. For this demo exercise this is fine, even though it will make the API „less RESTful“.

If you simplify the backend, in the frontend part, code gets more complex now. Just using HTML forms is not helping to create a clean POST request with a JSON body. You will need JavaScript.

Create a second HTML file with the same form to submit a product. You can now use the fetch() API to trigger POST requests with a JSON body from JavaScript. Add a <script> to your static HTML file to trigger the POST request upon form submit (+ parse to JSON).

Hint: You will need to prevent the „default event“ which is fired when the form is submitted. You will also need a piece of code that is then parsing the form data to JSON to be added to the fetch() call to the backend. Use console.log() for debugging if you get stuck. And ensure your HTML is not outdated because it is cached in your browser.

```
let formData = new FormData(this);
var object = {};
formData.forEach(function(value, key){
    object[key] = value;
});
var json = JSON.stringify(object);
console.log(json);
```

Alternatively, if you get stuck parsing the form data, feel free to hardcode an example product JSON that you can call the fetch API on button click.

Hint: You can use the response from the fetch call to the backend in the Promise callback (fetch(…).then(response => response .json() …) and use the received response content to update your HTML without reloading the file.

document.getElementById('response').textContent = … can help you adding the received data to the visible UI.

Alternatively, console.log(…) can help you debugging.

What could be the HTTP response codes for success and each of the possible failures?

Commit and push your code.

—-

Mapping frontend and backend for API calls is not a trivial task, so you might need a lot of debugging. If only one param name has a mismatch, your whole call might crash.

—-

3) Add a **deletion** endpoint in the controller for products with a path parameter for the ID. Use the annotation **@DeleteMapping**.

The actual deletion from the list shall be done by the service, not the controller. The service should return a list with all remaining products that the controller can then return to the calling frontend.

Hint: Java Collections have the utility method „removeIf“

If you are short in time, postpone connecting the frontend.

If you have time to create the frontend: Create a HTML file with a delete button (you can hardcode the product id to be deleted or read from an input field). You can again use the <script> with the fetch API and method: ’DELETE’ to call your backend.

Hint: You can add a click handler to the button using
`document.getElementById('delete').addEventListener('click', ...`

Create a response message indicating the frontend that the deletion succeeded. You can then use the fetch API response to log sth in the browser console or add a success message to the HTML.

What could be the HTTP response codes for success and each of the possible failures?

4) Add an **update** endpoint for products in the controller using **@PutMapping**. A service method needs to be added that searches the existing products to find a matching id of the product to be updated. If such a match is existing, all other parameters of the product shall be replaced of those provided by the put request.

If you are short in time, postpone connecting the frontend.

If you have time to create the frontend: We need data (from a form / fetchAPI call) as for the POST request, but we are not creating a new product, only updating an existing one. The identifier is the product ID which can be read from the received product object. Return the updated product as JSON and display it in the HTML via Javascript.

You can copy paste almost everything you did for the HTML and <script> from 3) while just changing the method from ‚POST‘ to ‚PUT‘ using the fetch API.

What could go wrong - what exception handling do we need?

What could be the HTTP response codes for success and each of the possible failures?