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

Exercise 1 - 23./27.10.2025

Practical Exercises

Ensure you did setup the private GitLab repo „distributed-applications-25-26“ in the correct GitLab (https://git-ce.rwth-aachen.de/) and invited @*cathrin.moeller* as a “Reporter“.

Please also invite my tutors as a „Reporter“: **@pinky-jitendra.tinani** and **@rahul.patil**

You will need this in the following weeks for the submissions proving active participation. It also helps as we will continue working on the same application for multiple weeks.

Please remember to submit to your GitLab repo regularly. You will need the solutions from today for the upcoming exercise classes and for „Active Participation“ we will check that regularly commits happened.

1) Ensure you did **all the exercises from last week** (including the MVC exercise 6 and the bonus exercise 7 and 8) to have the **starter code** for this exercise - if not, please do the remaining tasks which you can find in Moodle.

Commit and push as soon as you are done.

2) Commonly, application code is cleaned up from time to time to allow easier maintenance and extension. **Refactor** your application (if needed) ensuring you have a distinct separate Java class for the **Product** (model), one for the **ProductController**, one for the App and one for a utility **ProductService** (filtering our products by color / size …).

Often, code is sorted by feature domains, like the domain „product“ in our case. Create a package folder „**products**“ and move (refactor) the files belonging to the product feature in here. Ensuring all imports do still work and the code compiles and the app can still be started.

The **ProductService** can be used from the ProductController if it has a @Service annotation because it is then available in the dependency pool. Use both DI methods (from the lecture) to allow the ProductController using the ProductService.

3) Create a **second package** (folder) named „user“ with a controller (for a different GET request endpoint) and a service that is called by the controller. We also need a data model class. The second package will deal with user (management), so let’s name the files UserController, UserService and User(Model). User information shall be used to show a logged in user information like address data, which can be used for orders to be done in the webshop. Get creative defining suitable attributes for the user (e.g. id, firstname, lastname, email, addresses, ...). You can also define another class for the address model. Ensure you have constructors available for users and addresses. Hardcode some users with addresses in a list in the service class.

Ensure the application is running and that both the /users and /products endpoints are working in parallel. Add another endpoint that displays the user by its id (URL /user/{id}).

4) Our Spring Web application can not just return JSON (as a string), we can also serve static HTML files.

If you add a HTML file under `src/main/resources/static` (e.g. index.html)

you will be able to serve that content. Add some HTML here to ensure it is working properly. (Like a title „<h1>Welcome</h1>“).

! If you have a mapping for „/„ from a controller, this will „win“. You might need to open localhost:8080/index.html then to see the content.

Commit and push.

If you comment out any mapping like @GetMapping(„/") you will be able to see the index.html also as a default via localhost:8080.

Feel free to add more HTML or CSS content to your start page if you like. You are going to build a demo webshop during this course.

5) Create a subfolder in src/main/resources/static like src/main/resources/static/dashboard/ and add a new HTML file (like dashboard.html). Add some HTML in here to be able to detect if you do see the correct content (<h1>Dashboard</h1>).

Add a link or a button in the index.html to reach the dashboard file.

! The link path is a combination of the folder name within /resources/static and the file name.

Commit and push.

6) Now create buttons within dashboard to link back to the starting page and add links for all controller GET mappings (with hardcoded ids) you created so far.

E.g. add a button for the endpoint to display the product list (from exercise 0) and a button to reach our user list. You can add hardcoded lists for the distinct user ids or product ids. We will refactor that later.

Feel free to add a bit of CSS and HTML to make it look nicer.

GET endpoints can be tested easily using a browser and calling URLs.

As those endpoints return string content with JSON, not HTML, you can use browser back to navigate back to your static HTML view.

Test, that all links are working. Commit and push.

Mandatory for this week is up to here. Exercise 7) you can start if you are done, if not, you can catch up next week as the exercises are to be done consecutively.

7) You can now make use of having accessible HTML pages as part of the Spring application to not only trigger GET requests (via URL in browser) but also POST requests using a form in your HTML.

Add a <form> to a static HTML page (create/user.html), link that file to the dashboard and add an <input> field within the form for each user attribute like id and name (and optional other attributes you defined in your user class). You can skip some attributes, setting values to empty string as a default in the connected backend code.

The HTML <form> has attributes action and method. Default method is GET, so ensure you change the method to POST. The action sets the URL to be called.

Add a POST controller endpoint in the UserController using @PostMapping which returns the new created user if added successfully to your hardcoded list.

Be aware that list handling in Java can be tricky. Some lists are final, not to be edited. If you create e.g. an ArrayList using *private List<User> users = new ArrayList<User>();* you can add elements later using *users.add(user);*

Using **@RequestParam** in the arguments of your controller method, you can receive content from request parameters one by one. Submitting a HTML <form>, they are automatically added from the <input> fields by the attribute names.

Check your network tab in the browser dev tools to observe what is being sent. Make sure names in HTML match names in the controller file.

Call the /users GET endpoint to see the new user is now part of your hardcoded list.

Commit and push.