Telerik Academy

Beer Tag

Team Project Assignment

Java Team Project

This document describes a **sample team project assignment** for the **Java cohort** at Telerik Academy.

Project Description

Your task is to develop BEER TAG web application. BEER TAG enables your users to manage all the beers that they have drank and want to drink. Each beer has detailed information about it from the ABV (alcohol by volume) to the style and description. Data is community driven and every beer lover can add new beers and edit missing information on already existing ones. Also, BEER TAG allows you to rate a beer and calculates average rating from different users.

Project Requirements

UI

Create a "don't-make-me-think", simple, intuitive UI, so you have enough time to implement the backend carefully and adequately.

- Use **Spring MVC Framework with Thymeleaf** template engine for generating the UI.
- You may change the standard theme and modify it to apply own web design and visual styles. For example you could search and use some free html & css template to make your web application look good.
- You may use Bootstrap or Materialize

Web application

Public Part

The public part of your projects should be **visible without authentication**. This includes the application start page, the user login and user registration forms, as well as list of all beers that have been added from different users. People that are not authenticated cannot see any user specific details, neither they can interact with the website, they can only browse the beers and see list and details of them.

Private Part (Users only)

Registered users should have private part in the web application accessible after **successful login**.

The web application provides them with UI to add/edit/delete (CRUD) beers.

Each beer has name, brewery that produces it, origin country, ABV, description, style (pre-defined) and picture. Users can add tags to each beer. They can select among already added or create new.

Each beer can be marked as “drank” and “want to drink”. When beer is marked as drank message should appear with some text, for example “Cheers!”.

Each user can rate a beer and average rating is calculated. Both ratings should be visible for the users.

All beers are listed and can be sorted by rating, ABV, alphabetically. The list can be filtered by tags, style, origin country.

There should be master-details view that will show beer details when one is selected.

It is your choice of design where and what properties to visualize, but all data should be visible somewhere.

The app should have profile page that shows user’s photo and name from their contact on the device and their top 3 most ranked beers, they’ve tasted.

Administration Part

System administrators should have administrative access to the system and permissions to administer all major information objects in the system, e.g. to create/edit/delete (CRUD) users and other administrators, to edit/delete beers and related data if they decide to.

REST API

In order to work with collaboration with the QA team or provide other developers with your service, you will need a REST API.

The **REST API** should leverage **HTTP** as a transport protocol and clear text **JSON** for the request and response payloads.

API documentation is the information that is required to successfully consume and integrate with an API. Use Swagger to document yours.

Note: Please keep in mind that your web application should be build using **Spring MVC Framework with Thymeleaf template engine**. The REST API is only for external usage of your services.

Database

The data of the application MUST be stored in a relational database - **MySQL**.

You need to identify the core domain objects and model their relationships accordingly.

Technical / Development Requirements

General development guidelines include, but are not limited to:

- Use **IntelliJ IDEA**
- Following **OOP** principles when coding
- Following **KISS, SOLID, DRY** principles when coding
- Following **REST API** design best practices when designing the REST API (see Apendix)
- Following BDD when writing tests
- You should implement sensible **Exception handling** and propagation
- Every time you use System.out.println or e.printstacktrace() in a none-joking, serious manner, a kitten **dies**

Backend

- The minimum **JDK version is 1.8**
- Use tiered project structure (separate the application **components in layers**)
- Use **SpringMVC** and **SpringBoot** framework
- For Persistence use **MySQL/MariaDB**
- Use **Hibernate/JPA (and/or Spring Data)** in the Persistence layer
- Use **Spring Security** to handle user registration and user roles
    - Your registered users should have at least one of the two roles: **user** and **administrator**
- Service layer (e.g. "business" functionality) should have **at least 80% test unit coverage**

Frontend

- Use **Spring MVC Framework with Thymeleaf template engine** for generating the UI
- Use **AJAX** for making asynchronous requests to the server, where needed.

Deliverables

Provide link to a **Git** repository with the following information:

- Each project source code MUST be available in a dedicated **GitLab/GitHub** repository and **Trello board**
- Commits in the GitLab repository should give a good overview of how the project was developed, which features were created first etc. and the people who contributed. Contributions from all team members MUST be evident through the git commit history (so don't squash commits)!
- Read https://dev.to/pavlosisaris/git-commits-an-effective-style-guide-2kkn and https://chris.beams.io/posts/git-commit/ for a guide to write good commit messages
- The repository MUST contain the complete application source code and any run scripts
- The project MUST have at least **README.md** documentation describing how to build and run the project
- Screenshots of the major application user facing screens with some data on them
- **URL** of the application (if hosted online)
- Link to the **Trello** board

Optional Requirements (bonus points)

- Integrate your app with a **Continuous Integration server** (e.g. Jenkins or other). Configure your unit tests **to run on each commit** to your master branch
- Host your application's backend in a public hosting provider of your choice (e.g. AWS, Azure)

Public Project Defense

Each team will have a **public defense** of their work to the trainers and students (~5 minutes). It includes a live **demonstration** of the developed web application (please prepare sample data). Also each team will have a defense of their project with the trainers where they must explain the application structure, major architectural components and selected source code pieces demonstrating the implementation of key features.

Expectations

You MUST understand the system you have created.

Any defects or incomplete functionality MUST be properly documented and secured.

It's OK if your application has flaws or is missing one or two MUST's. What's not OK is if you don't know what's working and what isn't and if you present an incomplete project as functional.

Some things you need to be able to explain during your project defense:

- What are the most important things you've learned while working on this project?
- What are the worst "hacks" in the project, or where do you think it needs more work?
- What would you do differently if you were implementing the system again?

Appendix

Guidelines for designing good REST API

https://blog.florimondmanca.com/restful-api-design-13-best-practices-to-make-your-users-happy

Guidelines for URL encoding

http://www.talisman.org/~erlkonig/misc/lunatech%5Ewhat-every-webdev-must-know-about-url-encoding/

Always prefer constructor injection

https://www.vojtechruzicka.com/field-dependency-injection-considered-harmful/