Week 1

> You have a first basic prototype of an application, which you can deploy in a Jakarta EE application server (e.g. Payara). The application allows the user to navigate between the (empty) pages of your future application. There is a working registration / login page, which is backed by in-memory datastore. You have learned how to do end-user acceptance testing with CodeceptJS.

Suggested planning, assuming a group of 4 students:

Time Box 1: 30 minutes

- **Two students (Pair A)** focus on the **installation of an application server** and the configuration of IntelliJ. It should not take you more than **25 minutes** to have a setup where you can run and debug the application provided in https://github.com/SoftEng-HEIGVD/Teaching-HEIGVD-AMT-MVC-simple-example.
- This webcast should be helpful: https://youtu.be/NLKn_Sy8SSs
- In parallel, **two students (Pair B)** start to read and experiment with **CodeceptJS**. This is one of the tools that we can use to create automated end-user tests. In other words, tests that validate the behavior of the application from the outside (via a controlled browser). We have selected this tool because the setup is quite fast. **Invest 25 minutes** in the first discovery of the tool, make a test on a public site - no need to wait to have your application available to start testing.
- *Whereas we recommend to use IntelliJ for writing the Java code, we recommend to use Visual Studio code for writing the CodeceptJS tests*.
- Make a **quick status check** with the whole team, to summarize / demo what you have done. Spend about 5 minutes on that.

Time Box 2: 90 minutes

- **Pair A** then becomes familiar with Servlets, JSPs and Servlet Filters. Use the example code in the above repo. In 1 hour 15 minutes, you should:
  - Learn how to write a Servlet in a Java class
  - Learn how to write a JSP page with the JSTL tag library
  - Learn how to create a model in the controller (servlet) and pass it to the view
  - Learn how to use the model in the view
  - Learn how to intercept HTTP requests in a Servlet Filter (use the debugger to check that the servlet filter code is called, even if it does nothing at the beginning).
- In parallel, **Pair B** works on the specification of the user interface for the project. Start by making the list of all pages that need to be rendered (there should be about 5 pages). Describe **what information** is displayed in every page. Describe **what actions (commands)** can be performed by the user (by filling out forms, clicking on icons, etc.).
- Make a **quick status check** with the whole team, to summarize / demo what you have done. Spend about 15 minutes on that.

Time Box 3: 15 minutes

- Create a git repository, with the suggested structure
  - Use the standard **maven project structure**, with a pom.xml and a source folder at the root.

- Add a **docker** folder at the root, under which you will add your docker images and "topologies" (have a look at https://github.com/SoftEng-HEIGVD/Teaching-HEIGVD-AMT-Discovery, but place "images" and "topology-amt" under docker)
- Add a **e2e** folder under which you will place your automated end-user acceptance tests.
- Decide who in the team will take care of writing the initial README.md.

```
docker
  images
    payara
    flow
  topologies
    test
      docker-compose.yml
e2e
pom.xml
src
  main
    java
    webapp
      assets
      WEB-INF
        views
          fragments
      index.jsp
  test
README.md
```

Time Box 3: 30 minutes

Now, we recommend that you change the pairs (we now have pairs C and D). Pair C and Pair D do the same thing in parallel, working on a different page:

- **Pick one of the pages defined in your mockups and create**:
  - **1 model class**, which contains the data to be rendered in the page (e.g. a list of questions, statistics, etc.). Use Lombok to create @Value objects and the @Builder pattern.
  - **1 servlet** (e.g. HomePage or HomePageServlet or HomePageRenderer - pick a naming convention that you apply throughout the code base). The only responsibility of this class is to handle GET requests for this page. You only implement the doGet method, not the doPost. Initially, you simply forward the request to the correspond JSP.
  - **1 JSP page**, which presents renders the model at the right place in the HTML page. Start with raw HTML (no css).
  - **1 end to end** test to validate that you can navigate to the page and make assertions about the content (e.g. on the title, on page elements).

When you are finished and have a working and tested implementation, merge your code. Have a status check with the entire team to review your code (maybe you need to agree on the naming conventions and change one of the two implementations).

Next Steps (after the lab session)

At this point, you should have a first skeleton for your project and everybody in the team should understand how the whole thing works (at least to some level). Now, we need to build on this, but we should be able to split the work:

- **Someone needs to take the lead on CSS. (4 hours)**
  - Pick an web template / design system (e.g. plain Bootstrap, Tailwind CSS, one of the Creative Tim templates, etc.)
  - Figure out how to place the assets in the application structure (next to WEB-INF) and reference it from the JSP pages
  - Use the jsp:include tag to reuse the page header and footer, and to split the page in manageable "fragments"
- **Someone needs to take the lead on the "dockerization" of the solution. (2 hours)**
  - With maven, you produce a .war in the target directory
  - When you use a full-fledged container in a Docker image, you can deploy the .war by placing it in a particular folder. This depends on the app server that you have chosen. You will find information in the image documentation, on Docker Hub.
  - Create a Dockerfile for your application, and a script to build it (the script should build the project with maven, copy the .war file in a place where it is available for the docker build process)
  - When you are done, someone should be able to clone your repo, execute your build-image.sh script, move to the topology folder, do a docker-compose up and be ready to test your app in a browser.
  - This webcast should be helpful: https://youtu.be/GHL7U_IkBX0
- **Someone needs to take the lead on the registration and login processes. (4 hours)**
  - When you **register**, you fill out a form and click on a button. When you do that, **you send a command** to the application. Create a Servlet to process to this command. Pick a naming convention (RegisterCommandHandler, RegisterCommandServlet, RegisterCommandProcessor... it does not matter, but apply it throughout the codebase in the future). In this servlet, you will implement the doPost method, not the doGet. This is the first implementation, so we don't have a database yet. Do a quick implementation with an "in-memory datastore", which you can do with a HashMap with usernames as keys and user instances as values.
  - When you **login**, you also fill out a form and click on a button. It is another command, so create another Servlet (LoginCommandHandler).
  - This webcast should be helpful: https://youtu.be/XjeQl0oydmQ
- **Someone needs to take the lead on the e2e testing of the navigation logic. (4 hours)**
  - Create **scenarios** to describe and validate the navigation logic between the pages.
  - As the registration and login pages become available, these processes will need to be tested as well. This is when you will realize that it is important to give IDs and names to your DOM elements if you want to be able to write test scripts.
  - This webcast should be helpful: https://youtu.be/iPhVAmPN4qg