WEB ARCHITECTURES
Assignment 1

Riccardo Gennaro

30 September 2022

1 First part: simple web server modification

1.1 Introduction: problem statement

The problem that this project addresses consists in enabling a client, via HTTP, access to a server-side service, namely an external process used to reverse a string given via a GET request parameter. The reversed string will be returned to the client browser and shown. To do so, the client uses a browser and access the service located at the URL
`http://localhost:8080/process/reverse?par1="yourInput"`,
where `par1` is the GET parameter containing the input to reverse. Also the server must be able to handle wrong spelled URL paths, requests different from GET, and parameters with names different from `par1`.

1.2 Proposed solution

In order to enable the client-server HTTP communication, TinyHttpd, the class seen during lesson, was used to open a socket on port 8080 on server side. The server listen on this port and wait for an HTTP request.

To handle the HTTP request, the HTTP daemon used during lesson was modified to address the problem of this assignment.

The main daemon modifications consist in the request parsing and in the implementation of a call to an external process, the one used to reverse the string.

- `Request parsing:` the parsing of the request is handled by the constructor of the class `RequestParser` that run the same checks implemented in the original TinyHttpdConnection class, but also checks if the path corresponds to `process/reverse`, and if the parameter has name equal to `par1`. If not, depending on the case, status codes `404 Not Found` or `400 Bad Request` are returned. Once this checks are completed, the value of the parameter is extracted from the request and parsed in order to handle strings containing spaces.

- `Call to external process:` this call is handled by the constructor of the class `RunReverse` that uses the class `java.lang.ProcessBuilder` to start the program (`Reverse.java`) used to reverse the input parameter. In order to retrieve the output of `Reverse.java`, the output stream is captured by the parent process (our RunReverse.java class), parsed and stored in a variable.

If the request is found to be valid, a response is crated and sent to the client.

1.3 Screenshots

Following, various screenshot of the running application on the client browser.

1.3.1 Index Page

Image 1 shows the index.html, accessible via the URL `http://localhost:8080`. It contains a form in which it is possible to input the string to reverse. It is also possible to input the string as a parameter named `par1` via the URL.

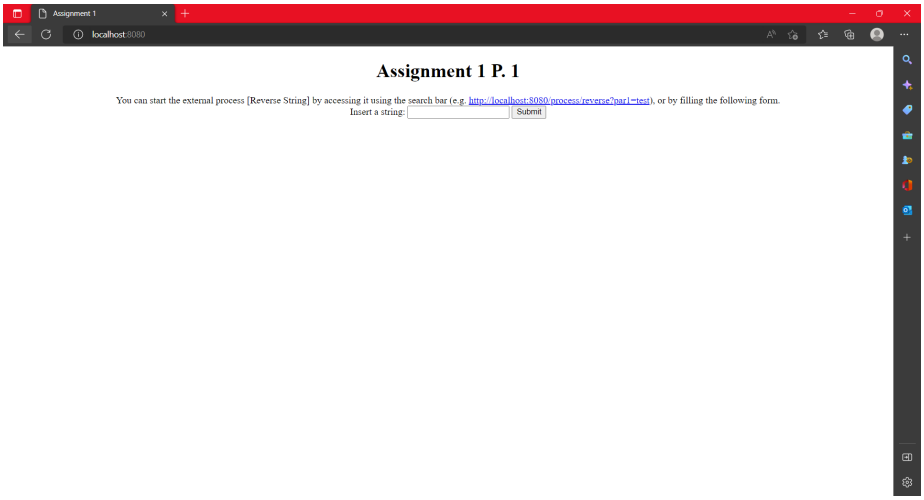

Figure 1: index.html

1.3.2 Output Page

After having submitted the form, or having sent a request via the search bar, a page is shown with the reversed input.
Image 2, depicts a test request via URL `http://localhost:8080/process/reverse?par1=roma torino`.

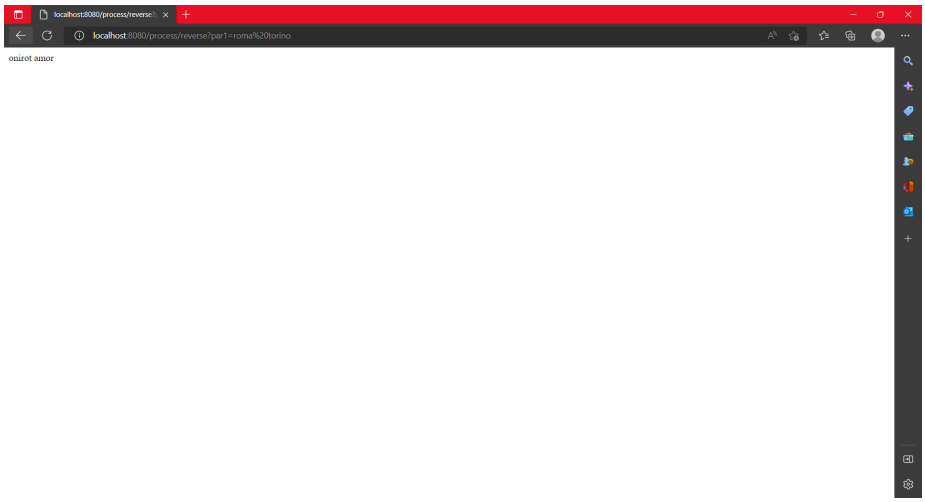

Figure 2: Output page

1.3.3 Error Pages

Following, the pages shown after having tried to input a path different from `process/reverse` (image 3), and after a sending a parameter with the wrong name (image 4).

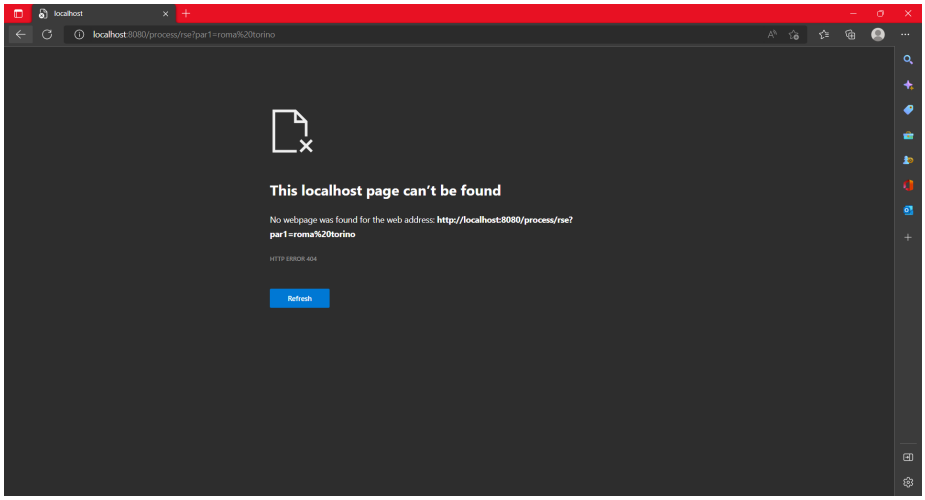

Figure 3: Not Found error

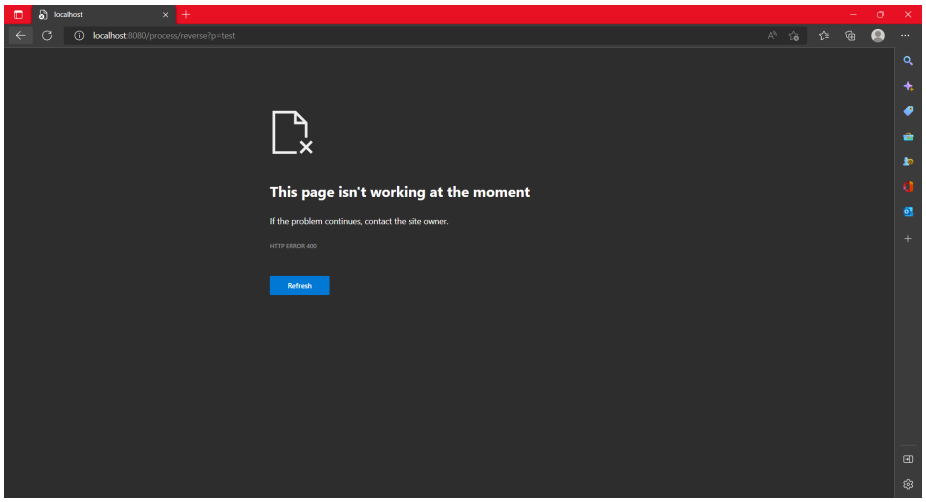

Figure 4: Bad Request error

1.4 IntelliJ Run Window

The server-side run terminal after a valid request with parameter `par1` = "roma torino".

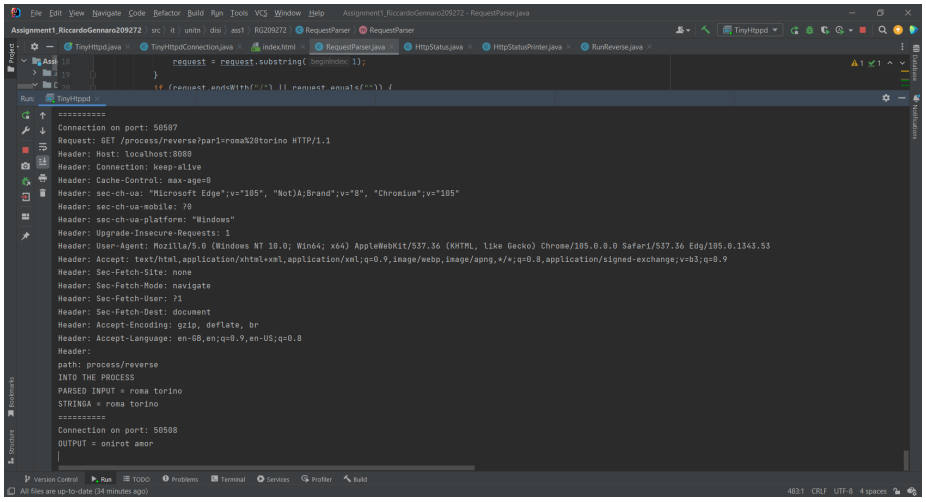

Figure 5: Intellij run window

1.5 Comments and notes

The only problem encountered during the development was the implementation of the handling of the HTTP errors and their return points inside TinyHttpd-Connection. To solve this problem a not so elegant approach was used: after having printed an HTTP error in `RequestParser.java`, a flag is set and made accessible to `TinyHttpdConnection.java` in order to return if the request is not valid.

2 Second Part: accessing .bat through Apache Common Gateway Interface

2.1 Solution Proposed

A batch file is written in order to retrieve the GET parameter saved in `QUERY_STRING` and store it in `VAR1` (an environmental variable of the Apache web server).
Subsequently, the `Reverse.java` used in the first part is run passing `VAR1` as argument. The reversed string, output of the java application, is then modified in order to remove `+` and `%20` contained in GET parameters with with spaces in them.

The resulting string is then send to the client.

In order to run the batch file from the client side, the .bat must be stored in the Apache cgi-bin directory, and accessed by the client via URL `http://localhost:80/process/reverse?par1="yourInput"`.

2.2 Screenshots

Following the screenshot of the code and the browser after serchong for `http://localhost:80/process/reverse?par1=roma torino`.

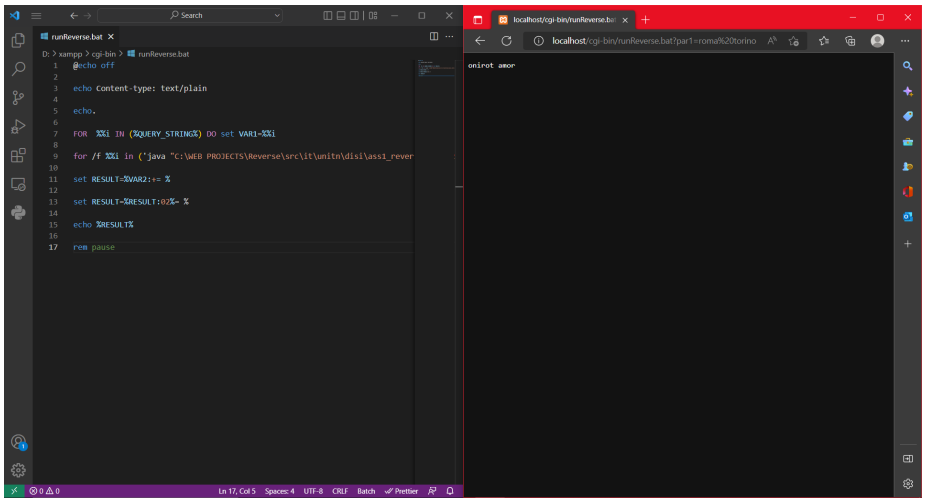

Figure 6: Accessing process through Common Gateway Interface

WEB ARCHITECTURES
Assignment 2

Riccardo Gennaro

16 October 2022

1 Introduction

1.1 Problem statement

This project aims at developing a web application capable of displaying dynamic pages satisfying the MVC framework. More specifically, the cited pattern has to be implemented using different Java API for servlets, JSPs pages and Java beans (see 1.2).

Regarding the application itself, it has to satisfy the following, shown synthetically, requirements:

- Role Base Access Control (RBAC). The application is accessible only by authenticated users. Users can be of three types:

  1. Admin. It can access the login page, registration page, and control page, this showing the active users and theirs score.
  2. Authenticated user. It can access the game, the login page and the registration page.
  3. Unauthenticated user. It can access only the login and the registration page.

  The Admin role is hardcoded, i.e. is specified in the source code and , therefor it is static. Registrated users information must be stored in file.

- Game. The application main logic must show a game in which the user must recognise the capital cities of three different nations chosen randomically between ten different options stored in a given folder containing ten .JPEG files of national flags. If the user rightly guesses all the capital cities, three points are added to it's score, otherwise, one point is subtracted. The user's score has session scope, meaning that it will not be persistance between different login sessions.

1.2 Model-View-Controller

This section wants to convey a basic idea of what the architecture of the application looks like.

The MVC pattern is implemented as follows:

- Model. It represents the application state. In our case, it consists of Java beans containing the data recovered at run-time from the persistent memory (in this case, a .txt file). The Java Bean model is accessed both by the View component (read), and by the Controller component (read/write).
- View. It defines the User Interface. The requirements asked the developer to use JSPs (Java Server Pages) to implement the View component. Using JSP technology enable the application to show dynamic content by establishing an interaction between the View and the Controller components.
- Controller. It is the application core, i.e. it contains the business logic. In this implementation it was required to use Java servlets, Java web filters and Java web listeners to satisfy the requirements.

2 Proposed solution - Workflow

2.1 Connection to the web application

When a user connects to the web application, it's request is redirected to the HomeServlet, as specified in the web.xml. At this point the request is filtered by the AuthServlet that checks whether if the user is an unauthenticated user, an authenticated user, or the admin.
In the first case the request is redirected to the LoginServlet; in the second, it is redirected to the HomeServlet; in the third, to the ControlServlet. This is decided by checking the session attribute `userBean`.

In order to initialize the application data, i.e. admin user definition, loading users info from .txt file, declaring needed data structures, the LoginServlet overrides the init method to call the constructor of the Initializer class.

2.2 Login

Once the request come to the LoginServlet, page login.jsp is sent to the client. Once the login form is filled and submitted, a POST request is sent to the AuthServlet that checks in the previously initialized HashMap if the user credentials are correct. If so, the request is redirected to the HomeServlet or the ControlServlet depending on the user's role. Otherwise, the request is redirected to the LoginServlet.
If the authentication is successful, session attribute `userBean` containing the user info is added, and the ActiveUserListener handle the event by adding the user data to an HashMap containing the active users.

2.3 Registration

A GET request to the RegistrationServelt can be sent by clicking on the registration link in the login.jsp page on client-side.
This servlet send to the client the registration.jsp page, prompting the user to fill the form with the desired username and password (the latter, two times). Submitting the form will send a POST request to the AddUserServlet.
At this point, the servlet checks if the two input of the password are not equal or if the submitted username already exists; if so, the request is redirected to the RegistrationServlet, otherwise, a new UserBean is declared with the user information and is stored both in the HashMap residing at context level and in the .txt file. Lastly, the request is redirected to the LoginServlet.

2.4 Game - Presentation

After having successfully completed the login, and if the authenticated user is not an admin, the HomeServlet will send a response to the client containing home.jsp. This page is dynamically generated since it contains both the username and the score of the authenticated user.
Home.jsp contains a submit button that sends a GET request to the GameServlet. This servlet randomly selects three flags from the flags directory contained in the deployed application directory (using method `javax.servlet.ServletContext.getRealPath()`). Having done this, the three flags are saved in an ArrayList and put in the session table as `chosenFlagIndex` attribute. At this point, a response containing game.jsp is sent. The page is dynamically built to show a form with the three images of the flags. Once the form has been submitted, a POST request containing the index of the chosen answer is sent to the CheckAnswerServlet.

2.5 Game - Answer checking

The indexes contained in the POST request are compared with the chosen flags. If one of them do not match, one point is subtracted from the score attribute of the user UserBean, otherwise, three points are added.
Both in CheckAnswerServlet and in game.jsp, the answers are checked in order to assure that none of them are null or NaN. In game.jsp this check is ran setting the required and type field of every input in the form; while in the servlet, this checked is performed by parsing the request parameter to Integer and handling `NullPointerExeption` and `NumberFormatException`.

2.6 Control page - Presentation

If the user authenticate as admin, then, after validation, the request is redirected to ControlServlet. Here, a chek is performed to make sure that the request was made by a user with admin role; if not, the response status is set to 401, and the relative error page is sent. If the user is in fact an admin, a response containing controlPage.jsp is sent to the client. The page contains the list of active users.

2.7 Control page - Active users

In order to keep track of all the active users, a web listener has been implemented. The listener waits for a state change in the session (i.e. attribute added, attribute removed, attribute replaced), and depending from the type of event, it adds or remove a given user from the HashMap of the active users. This events can be triggered by authenticating as another user, or by letting expire the session (as specified in the web.xml, the session timeout is 3 minutes).

2.8 Filters

This web application uses two filters:

- AuthFilter. This filter is used for every request to every paths, with exceptions being /LoginServlet, /RegistrationServlet, /AddUserServlet, /AuthServlet, /login.jsp, and /registration.jsp. This filter is used to enforce the above cited RBAC.
- JspFilter. This filter is used to prevent direct access to the .jsp pages. Since multiple pages are dynamically generated, trying to access it without sending the request to a servlet can lead to unexpected behaviour both on client and server side.

Mapping and definition of the filters can be found in the web.xml.

3 Running application

3.1 Login

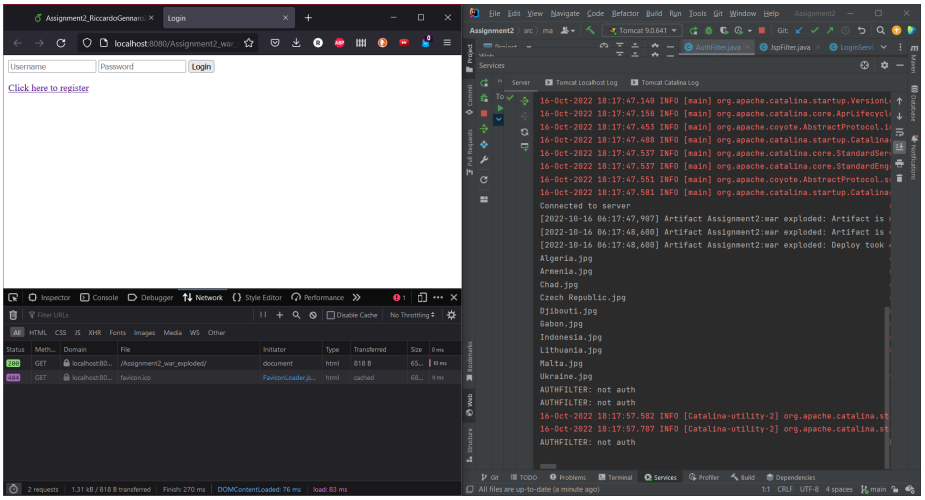

Figure 1: Login page

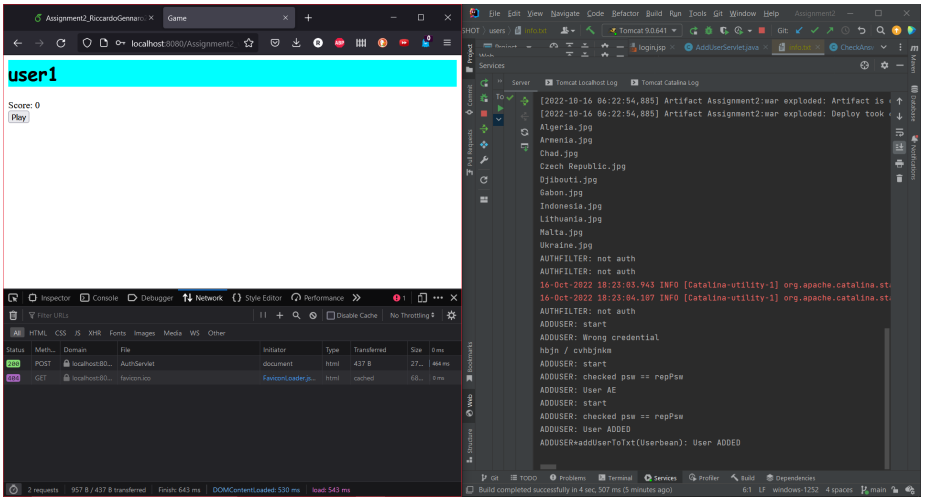

Figure 2: Successful login as user

3.2 Registration

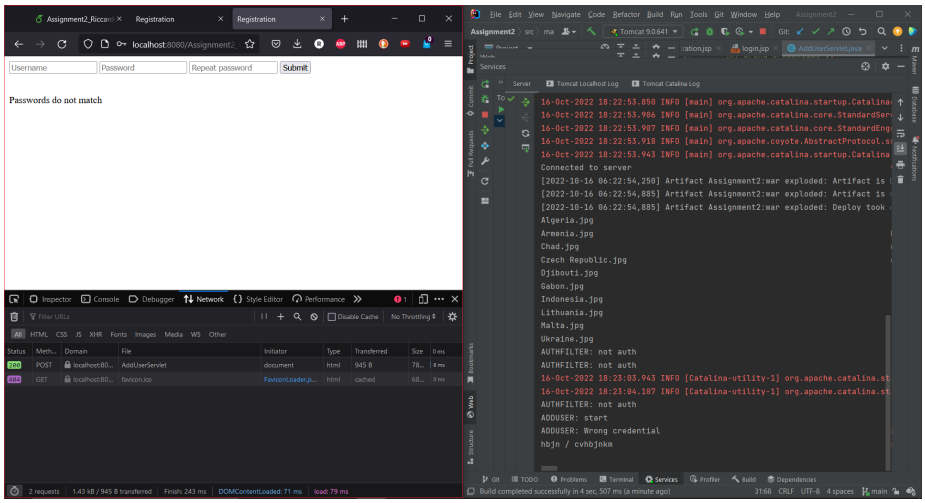

Figure 3: Passwords do not match during registration

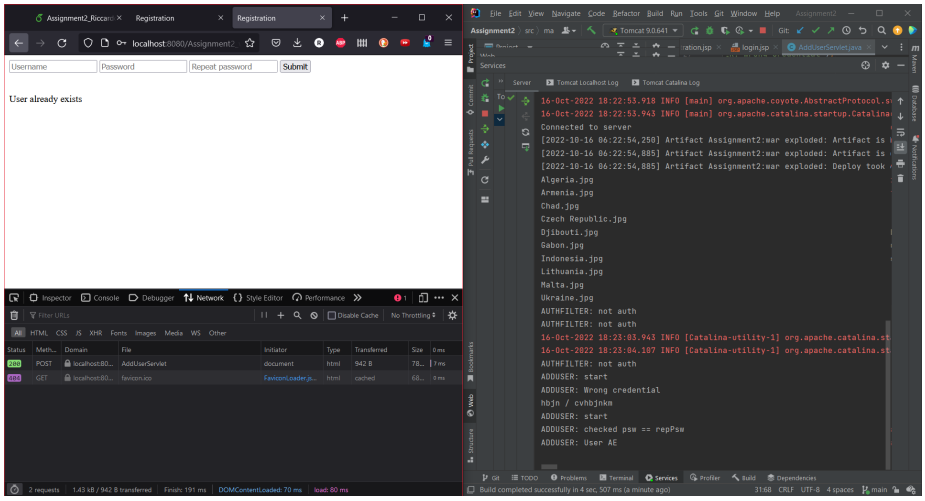

Figure 4: Trying to register already existing user

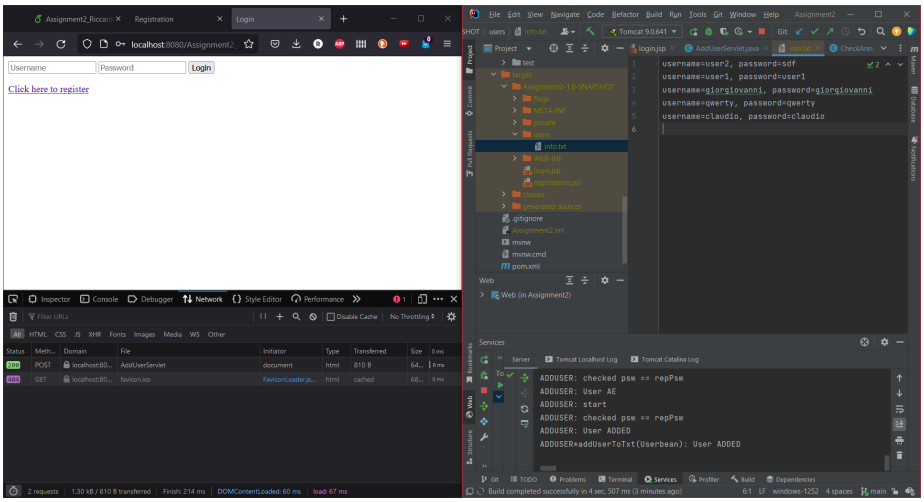

Figure 5: Successful user registration

3.3 Game

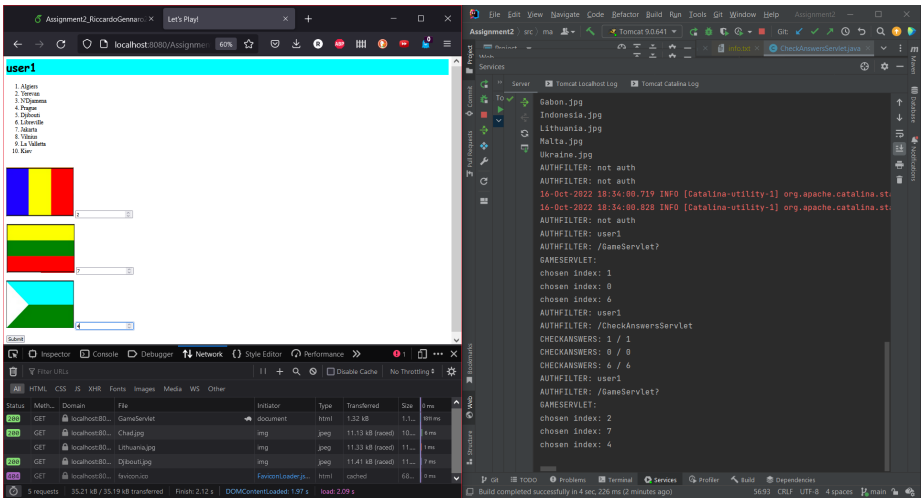

Figure 6: Game page with correctly selected answers

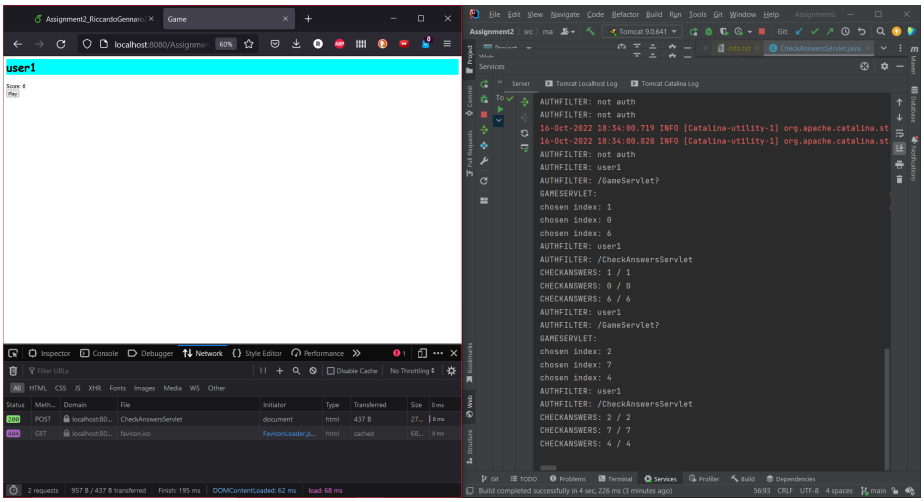

Figure 7: Home page after two all correct answers

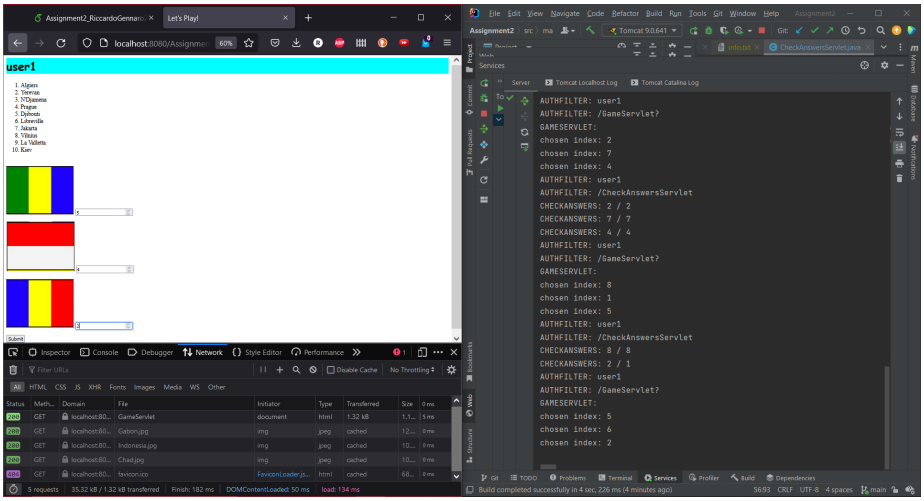

Figure 8: Game page with wrongly selected answers

3.4 Control page

Note that reloading the control page with Ctrl+R (and relative MacOS/Linux key), will resend the request to the AuthServlet, thus not needing to authenticate again after the session has expired.

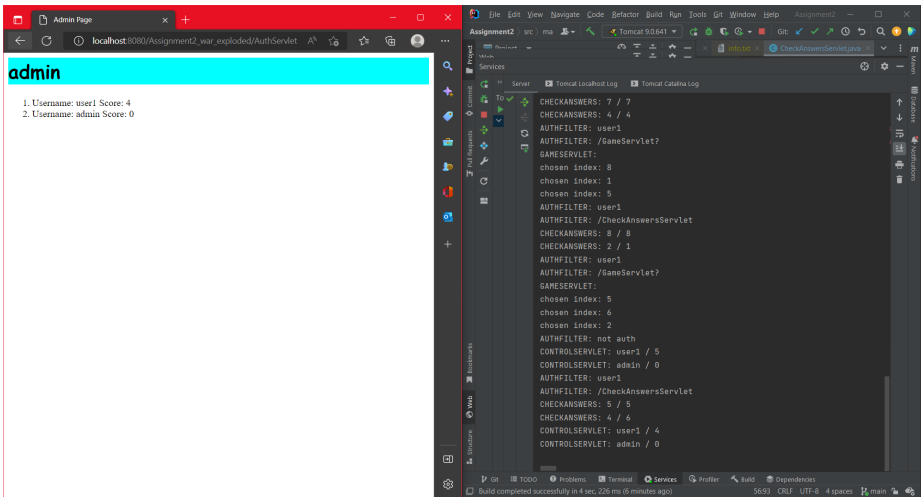

Figure 9: Successful login as admin. Control page

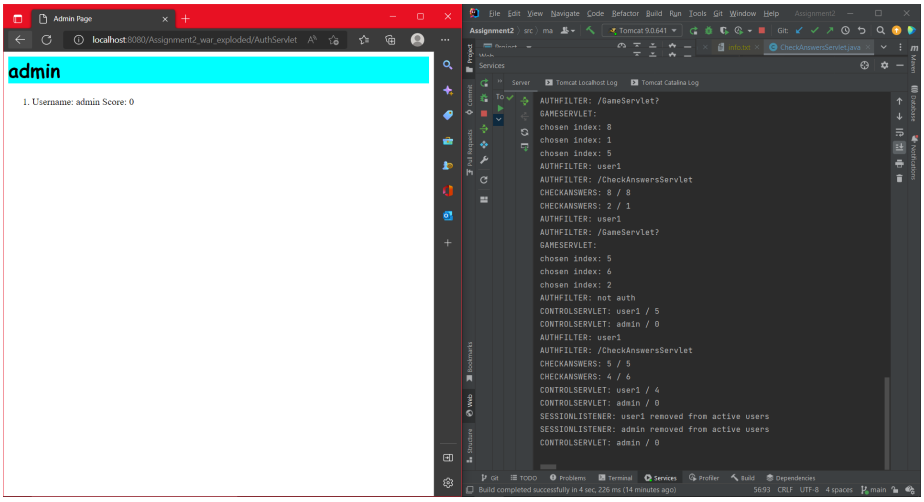

Figure 10: Control page after user1 session has been destroyed

3.5 Comments and notes

The only problem that I've encountered was during the read/write operations on the user info text file. It appeared that the solution proposed in one of the link listed at the end of the project requirements was not working as intended (problems with the charset of the output stream). In order to quickly solve the problem I used a simple FileWriter and FileReader to implement the file I/O, I then manually parsed the text document using String.split().

WEB ARCHITECTURES
Assignment 3

Riccardo Gennaro

November 6, 2022

1 Introduction

1.1 Problem statement

This project aims at developing a single-page web application that shows to the users a simple spredsheet.

Regarding the application itself, it has to satisfy the following, shown synthetically, requirements:

- **Spreadsheet**. The application must show to the user a matrix of 16 cells. These cells can be read/written by anyone who access the web application. The spreadsheet implements the basic four arithmetical operators.
- **Asynchronous communication**. The client-server communication must be carried out in an asynchronous way. For this project, the communication was implemented through AJAX requests using JSON files.
- **Polling**. Once a user cause the value of cell to change, this operation must be propagated to all other users. To do so, a client-to-server polling was implemented.

2 Proposed solution

Figure 1 represents the server-side workflow depending on the Javascript function called client-side. Following, a description of the workflow of the application.

2.1 Connection to the web application

When a user connects to the web application, it's request is redirected to *SheetServlet*, as specified in the web.xml. The servlet send a response containing *spreadSheet.html*.

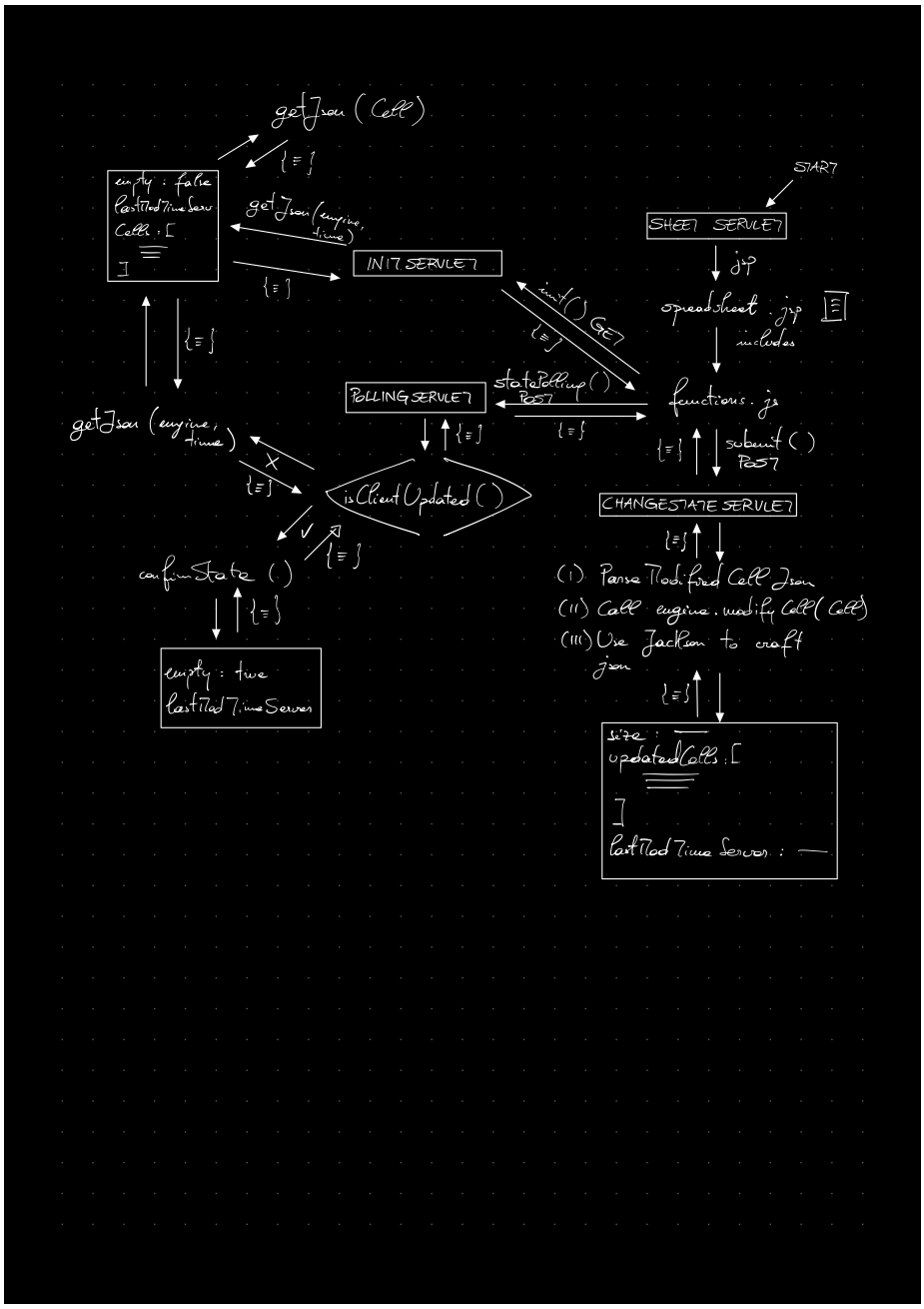

Figure 1: Server side workflow.

2.2 Spreadsheet

At this point, the .html page is rendered client-side. File spreadSheet.html includes file *js/functions.js*, containing the javascript code used to build the page and communicate with the server.

2.3 Client-side functions

File *js/functions.js* contains the below functions.

- `init()`. Called on page load. Sends a GET request to *InitServlet*. When the servlet answer with a JSON containing the cells, `init()` parse the JSON, initialize an array of cells, and render the html table containing the cells. This function also defines the event listeners related to input text area used to edit a cell.
- `submit(formula)`. This function is called via a wrapper, `submit_wrapper(formula)`, that checks if the formula to submit has been changed or if it is empty. Once `submit(formula)` is called, the function send a POST request to *ChangeStateServlet* with a JSON containing the modified cell's data. Server answer with a JSON containing the list of modified cells and the timestamp of the last modification. Changes are applied client-side.
- `statePolling()`. This function is called every five seconds. It sends a POST request to *PollingServlet* with a JSON containing the above mentioned timestamp of the last modification. If the last server-state modification occurred after the sent timestamp, server sends the updated state of the spreadsheet through a JSON file; otherwise, it sends a confirmation that the state has not been modified through a JSON containing an `empty` flag. Once client receives the JSON, it checks the `empty` flag: if it's true, the state is not modified; if it's false, the cells values and formulae are updated. This function is needed in order to keep track of other users' changes.

File *js/functions.js* also contains other auxiliary functions used to manage the page CSS and other events.

2.4 Server-side servlets

2.4.1 SheetServlet

As previously specified, this servlet is called at the connection to the web application. It sends *spreadSheet.html*.

2.4.2 InitServlet

As previously specified, this servlet is called through a GET request at the loading of *spreadSheet.html*. It initialize the provided application `engine`, build a

JSON containing the cells' values and formulae through `Json.getJson(SSengine, long)`, and sends it to client.

2.4.3 ChangeStateServlet

This servlet answer the POST request sent by client through `submit(request)`. It parses the JSON using the *Jackson* library, and uses `engine.modifyCell(String, String)` to update the server state. If state is modified, a JSON containing the modified cells and the timestamp of current modification is crafted and sent to client; otherwise a JSON containing attribute size set to zero is crafted and sent to client.

2.4.4 PollingServlet

This servlet answer the POST request sent by client through `statePolling(request)`. It checks if the client-side timestamp of the last modification is *before* the server-side one. If so, it crafts JSON containing the updated engine state and the last modification timestamp through `Json.getJson(SSengine, long)` and sends it to client. If last modification is *after* the server-side one, a confirmation that the state is up to date is sent.

3 Running application

3.1 Start

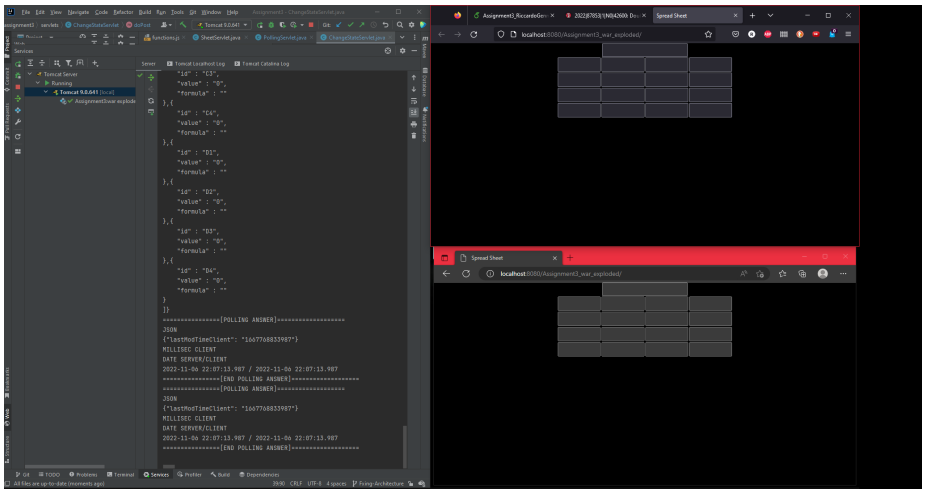

Figure 2: Initial page. No modifications

3.2 Successful submits

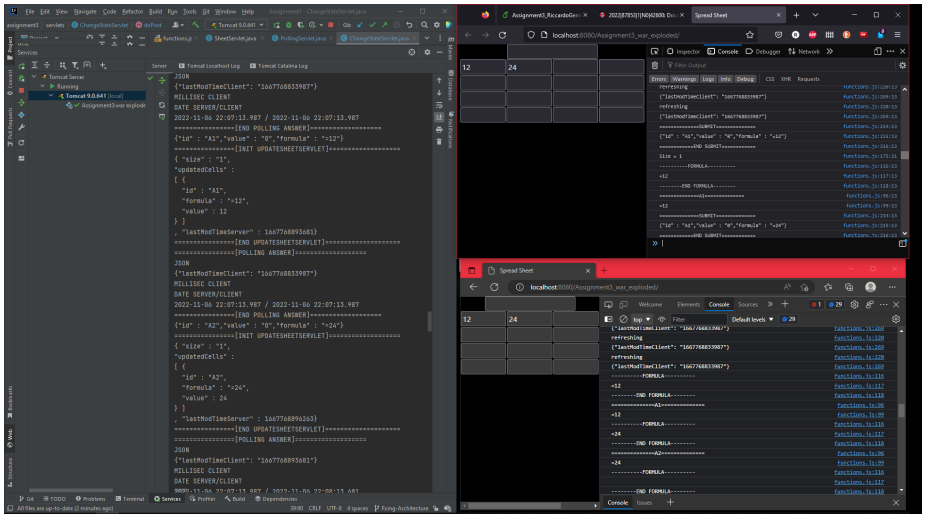

Figure 3: Successful submit of values 12 and 24 in cells A1 and A2

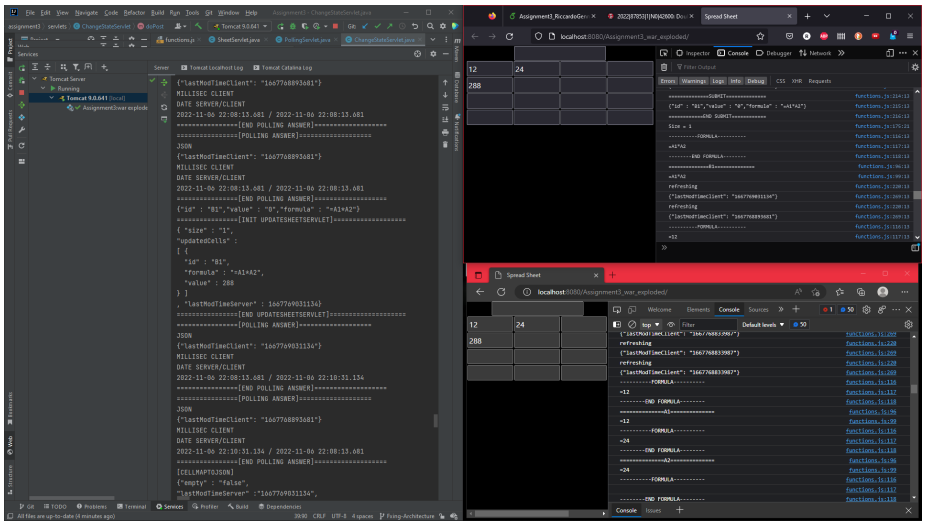

Figure 4: Successful submit of operation =A1*A2 in cell B1

3.3 Unsuccessful commit

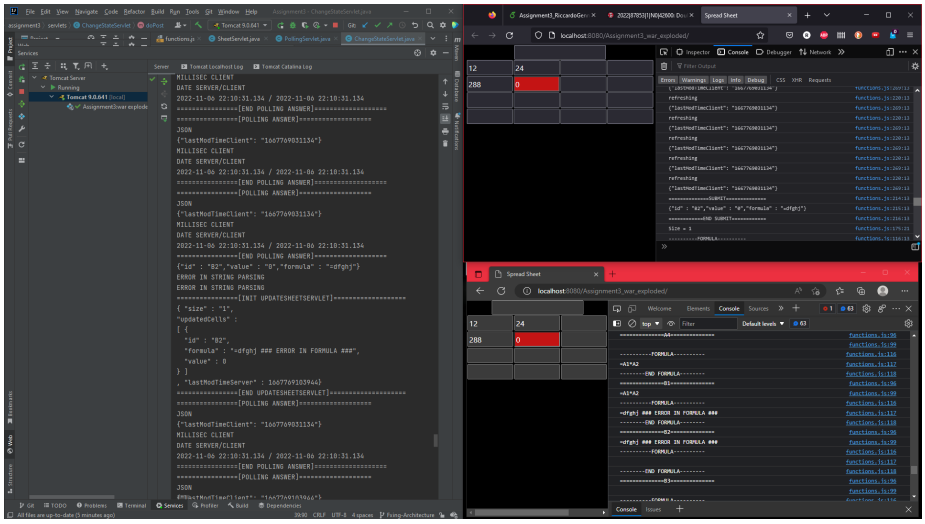

Figure 5: Error while trying to submit not valid formula

3.4 Comments and notes

Note that submitting a formula with circular dependencies will not change the server state. I tried to change some code in SSEngine::modifyCell(String, String) in order to obtain an error message on client-side, but doing so caused the Server to throw a StackOverFlow exception whenever a formula with nested circular dependency were evaluated. The changed code (now commented) can be found in SSEngine::modifyCell(String, String). The StackOverFlow exception was thrown by Cell::recursiveEvaluateCell(), since storing the value of the cell with circular dependency caused the recursive evaluation of the above mentioned cell.

WEB ARCHITECTURES
Assignment 4

Riccardo Gennaro

December 20, 2022

1 Introduction

Problem statement

This project aims at developing a simple web application capable of displaying to a user information about a given student. In particular, there are two use case: the request for student info, that displays anagraphics and choosen courses, and the advisor choice, that displays the anagraphics of a given student and the teachers of the courses he/she is enrolled in.

As requested by the assignment, the business logic must be implemented through EJB and deployed on a WildFly server that lays outside of the Tomcat were the client side must reside.

2 Proposed solution

This assignment is divided in two projects, one deployed on WildFly, one on Tomcat.

Tomcat - Servlets

HomeServlet

It is the welcome file, and it simply presents the `index.jsp` from which it is possible to input (through a form) the student ID for requesting the student info or to access the advisor choice. Using a non-registered student ID will display an error message.

StudentServlet

This servlet makes use of the `StudentManagerBD` to access the data of a given student and displays it through `studentPage.jsp`. The requested data are:

- Student name;

- Student surname;
- Student ID;
- Courses in which the student is enrolled;
- Grades of the above courses, if any.

StudentServlet

This servlet makes use of the `AdvisorChoiceBD` to access the data of a given student and displays it through `advisorChoice.jsp`. The requested data are:

- Student name;
- Student surname;
- Student ID;
- Names of the teachers that theac the courses in which the student is enrolled;

Tomcat - Business Delegates

Both classes `AdvisorChoiceBD` and `StudentManagerBD`, are used to access facade `AdvisorChoiceManagerFacade` and facade `StudentManagerFacade` functions. To do so, the business delegates instantiate these interfaces through the Service Locator.

Tomcat - Service Locator

RemoteServiceInitializer

This class is a singleton used to set the JNDI properties and to operate the lookup in order to make accessible the services that resides in the WildFly server. It also implements a caching of the already looked up services. The caching is implemented through the `cache` hashmap;

WildFly - Facade Beans

As anticipated above, this solution impents two facade. This facades implementations are two beans that expose their methods to execute high-level operations. To do so, To answer the client, this beans converts the entity to a detatched version of the object, a DTO. To build the DTO, class `DTOAssembler` is used.

WildFly - Entity Beans

This beans are not exposed to the client, but are used in order to interact with the entities through the `EntityManager` class. This beans implement the communication with the database. This beans are accessed by the facade implementation through injection operated by the @EJB annotation used in the facade bean.

Database and entities

Following the database ER-schema.

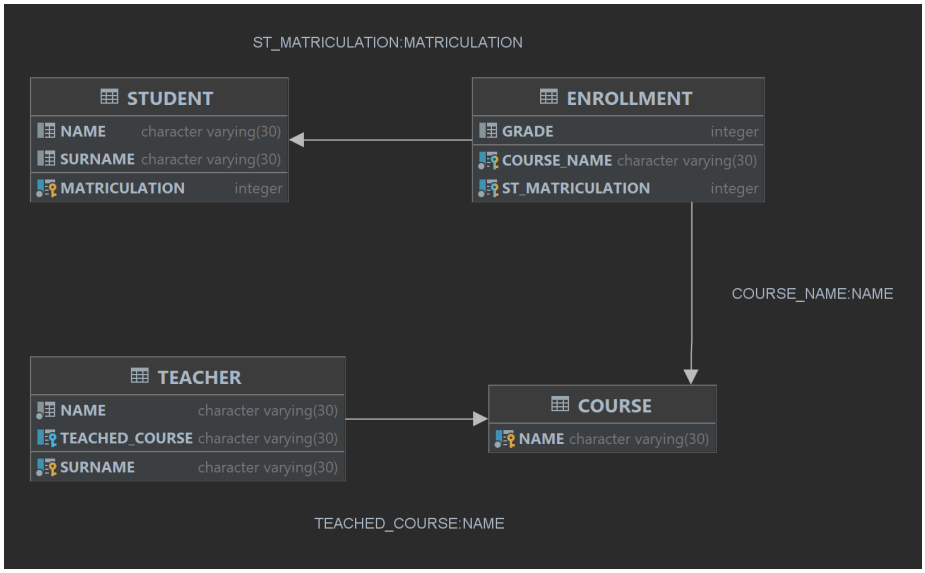

Figure 1: Database ER

There are four tables

- STUDENT, with three columns:
  - NAME
  - SURNAME
  - MATRICULATION (Primary Key)
- TEACHER, with three columns
  - NAME
  - SURNAME (Primary Key)
  - TEACHED_COURSE (Foreign Key, for 1:1 relation with table COURSE)

- COURSE, with a single column
  - NAME
- ENROLLMENT, table used to implement the N:M relation between tables STUDENT and COURSE.
  - GRADE
  - COURSE_NAME
  - ST

  ENROLLMENT primary key is given by pair (COURSE_NAME, ST_MATRICULATION).

To connect the database the file `resources/META-INF/persistence.xml` has been modified to include tag `<jta-data-source> java:jboss/datasources/mydb </jta-data-source>`. The datasource has been configured via WildFly's `standalone.xml`. The edited `datasource` tag of the `standalone.xml` can be found at the base path of this deliverable in file `standaloneSnippet.xml`. The database data have been exported through files `mydb.mv.db` and `mydb.trace.db` and can be found at the base path of this deliverable.

Running application

Welcome Page

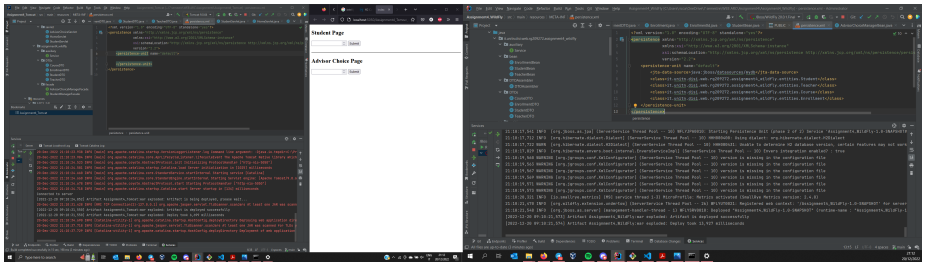

Figure 2: Initial page. There are two forms: one redirects to Student Info, the other to the Advisor Choice

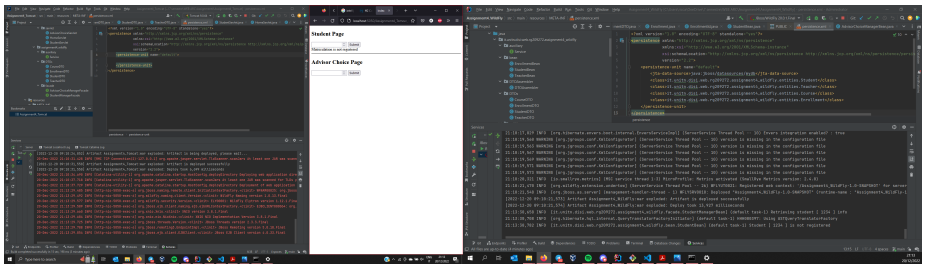

Figure 3: Initial page. A non-registered student ID was requested

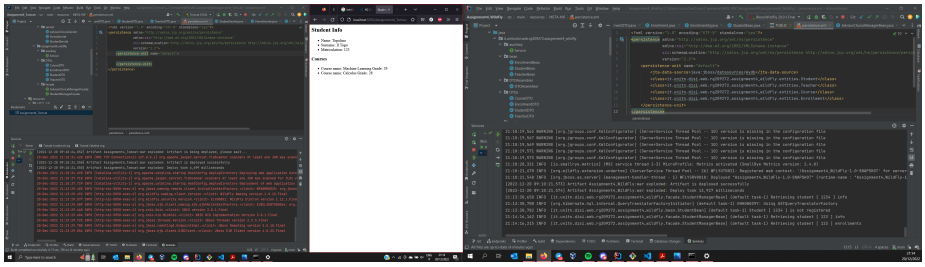

Figure 4: Student page.

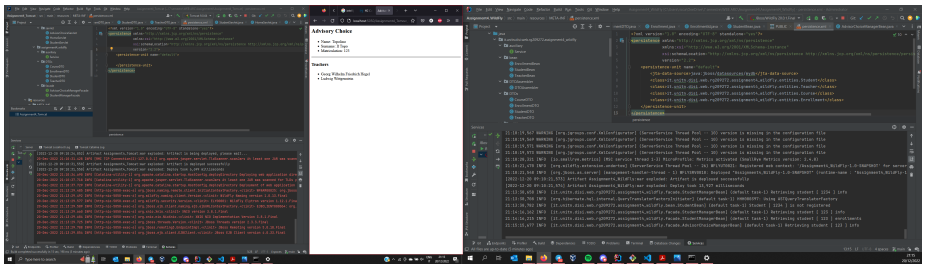

Figure 5: Advisor Choice page

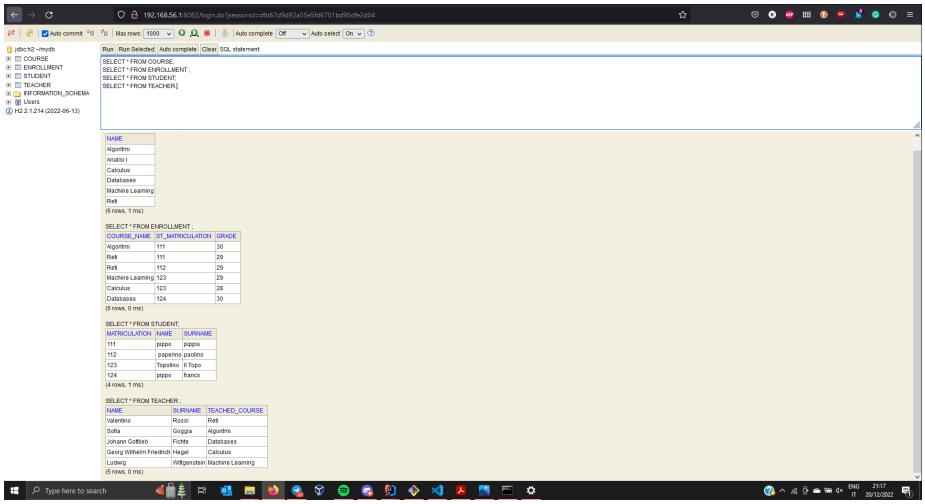

Figure 6: Database Tables

Comments and notes

The project presents duplicated code. In particular, packages `it.unitn.disi.web.rg209272.assignment4_wildfly.auxiliary | DTOs | facade` are duplicated both in Tomcat and WildFly sides. A solution could be including this packages through maven dependencies. I tried doing so, but WildFly threw a bunch of errors about the name of the service and I didn't have time to investigate.

Also, I must say that one of the biggest challenges was the configuration of WildFly and the understanding of the requested patterns.

WEB ARCHITECTURES
Assignment 5

Riccardo Gennaro

January 6, 2023

1 Introduction

1.1 Problem statement

This aims at developing a web application via Angular framework. More specifically, the application uses the Scottish Parliament APIs to retrieve information, in JSON format, about the parliamentarians, and offers to a user

- the **list of the politicians**, including names and personal photos. If the photo is not available a default one is shown.
- for a given politician, the ID, some personal information (name, date of birth) together with the parliamentarian's parties and websites are displayed.

Since Angular is used to develop single-page applications, we change the state of the app using Angular Routing.

2 Implementation

Following, a description of the workflow of the application.

2.1 Connection to the web application

Upon connection to the web-app, the user is redirected to route-path `[base]/list` with component `MspListComponent`. The redirection is defined in the exported Routes in `app.routes.ts`.

2.2 List display

2.2.1 MspListComponent - BackEnd

Redirecting to `[base]/list` triggers `MspListComponent#ngOnInit()`. This method retrieves the MsSP (Members of the Scottish Parliament) data via API

'https://data.parliament.scot/api/members'. In particular, ngOnInit() calls service method MspService#getMspEntries() to retrieve the observable bound to the GET request to the API. Successively, we subscribe to the returned observable. On next subscription, we put the retrieved data in attribute MspListComponent#mspEntries.

2.2.2 MspListComponent - Frontend

In file msp-list.component.html, the data is presented via a grid containing components of type MspGridItem that receive in input an MspEntry. These components are generated through iteration in directive *ngFor.

2.2.3 MspGridItem - BackEnd

Given the mspEntry got through the @Input decorator, this component expose to the previously retrieved data about the iterated parliamentarian. Furthermore, two methods are implemented

- **getImage()**, used to return the PhotoURL in order to display the parliamentarian photo;
- **onItemClick()**, used to change route to [base/msp-details/:id]. This function is called when the component is clicked at frontend.

2.2.4 MspGridItem - FrontEnd

The grid is filled with clickable mat-cards displaying the MSP's name and photo via interpolation.
At this point the list is fully rendered.

2.3 MSP details display

Clicking the card of an MSP will call MspGridItem#onItemClick() changing to route path [base/msp-details/:id], instantiating MspDetailComponent.

2.3.1 MspDetailComponent - BackEnd

Redirecting to [base/msp-details/:id] triggers
MspDetailComponent#ngOnInit(). At this point, the MSP ID is retrieved from the route parameters, and, via a pipe, we get the personal data, the parties and the websites of the MSP.
During the implementation of this method, nested subscriptions were avoided since they cause memory leaks and make the code unmanageable. To solve the problem of this pattern, while still managing the asynchronous actions, a pipe with multiple rjsx#map is used to concatenate the different requests used to retrieve the needed information.

The service methods used to retrieve these information are similar to the one described in 2.2.1. Worth mentioning is `MspService#getMspEntry(id:number)` that can throw an error caused by a not found MSP ID. This error is handled via a `rxjs#catchError` in the pipe. The handling consist in redirecting to route path `[base]/list`.

2.3.2 MspDetailComponent - FrontEnd

This components contains three `mat-card`

- **Personal Details**, that shows data from component `MspPersonalDetailComponent`
- **Parties**, that shows data from component `MspPartyDetailsComponent`
- **Websites**, that shows data from component `MspWebsitesDetailsComponent`

Striving to maintain high modularity, resulted in a high number of components.

The backend of these three components do not contain any notable code. The data is shared from parent to children via Input() decorations, the data is renderend in the frontend via interpolation.
At this point, the MSP details are fully rendered.

3 Deployment

3.1 List current MsSP list

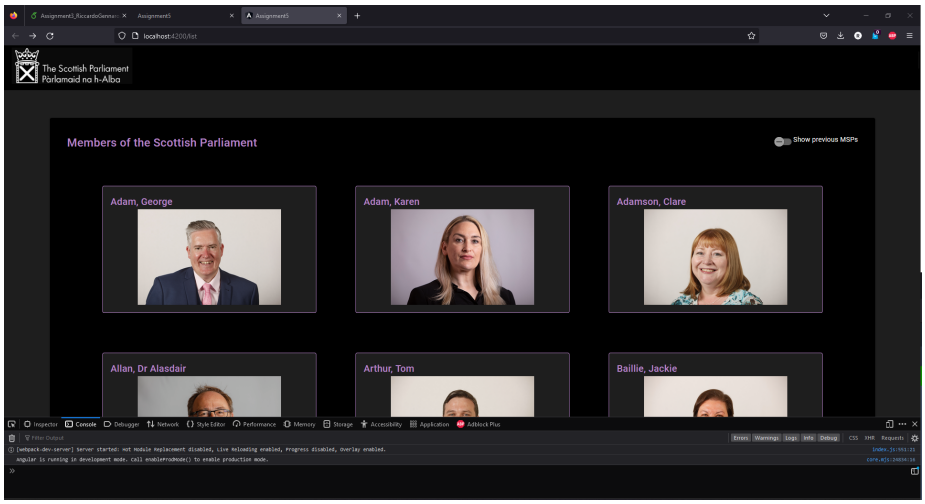

Figure 1: Rendered upon opening

3.2 Current MSP display

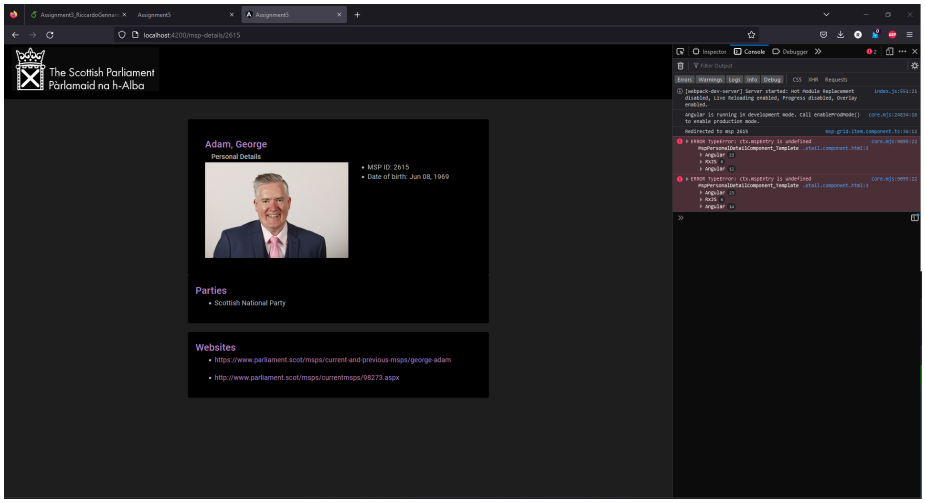

Figure 2: Rendered after clicking on Adam, George card

3.3 List all MsSP list

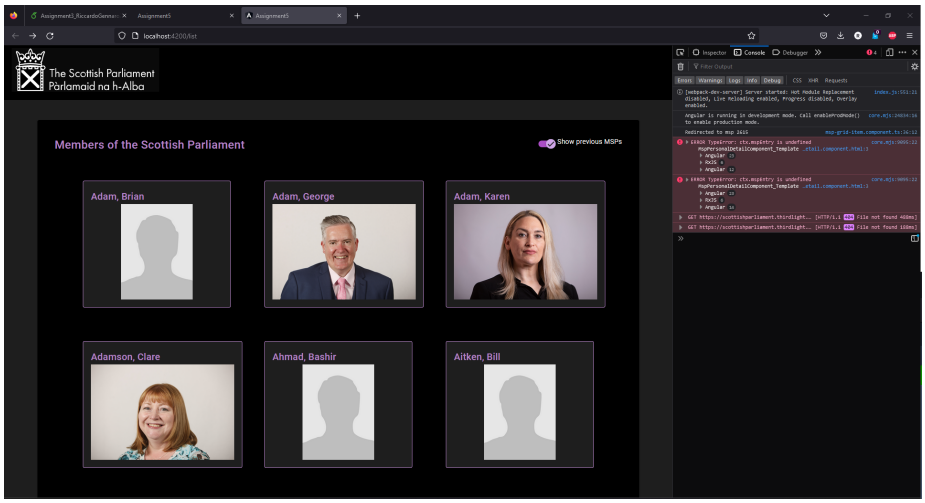

Figure 3: Rendered after checking the slide toggle

3.4 Display non-current MSP

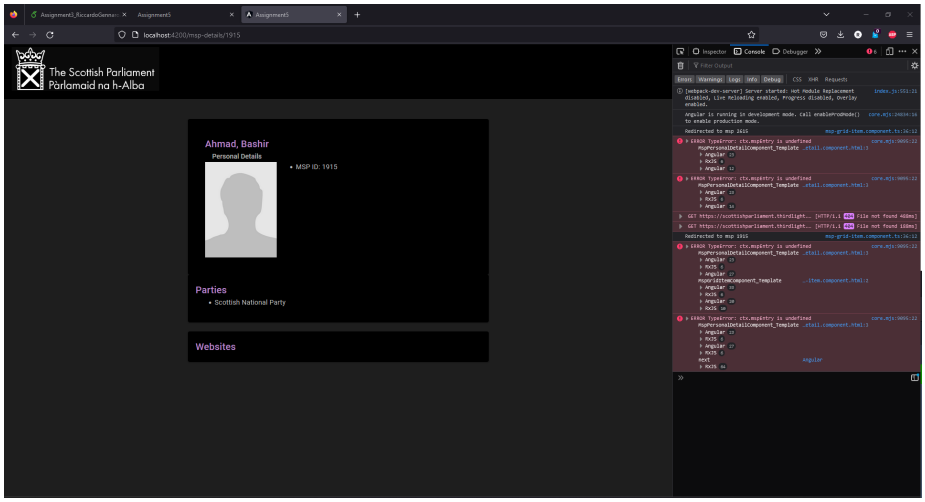

Figure 4: Rendered after clicking on Adam, Brian card

3.5 List all MsSP list - Tomcat

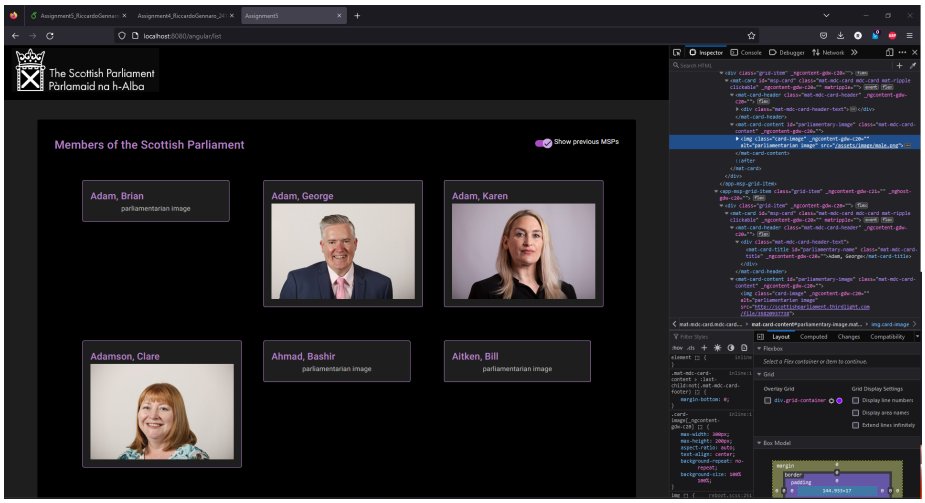

Figure 5: App deployed in Tomcat is the same that the one deployed using command `"ng serve"`. Only one difference - see Comments

4 Comments and notes

For some reasons, when the project is deployed in Tomcat, the default images are not rendered. As you can see from the project structure, and from the image above, even if the default images are in the same folder as the logo of the parliament, and the path of the default images are correctly interpolated, the images cannot be rendered.

This problem is not present when running the project with command `"ng serve"`.

Another problem can be seen at Figure 2. Upon opening the details of a MSP, a TypeError will be thrown with message `"ctx.mspEntry is undefined"`. I tried to locate the source of this error but I was unable to fix it.