Software Quality Project
Documentation

Members:
- Croitoru Razvan
- Douet Marie
- Filipescu Iustina-Andreea
- Istrate Liviu

Introduction:

The purpose of this project was to implement and test an application which assists a user in creating a timetable of the faculty. We decided to separate the application into two parts (Frontend and Backend) so it could be easier to test each component individually.

From the UI part, the user is able to select different values to insert into the timetable. After the values are chosen, they are sent to the backend (REST endpoint) where they are validated and depending on the validation result, the time slot may be added to the schedule or a validation error with a specific message may be returned to the user in the UI.

Anytime during the time slots insertion process, the user may see his progress by viewing the updated timetable. The timetable is displayed in a single table and for a timeslot, multiple classes may be recorded (as long as they don't cause any conflicts).

After the user is satisfied with the timetable that he created, he may then save the table as a html file.

App Architecture:

1. Backend
2. Frontend

Backend:

- Controllers:
    - Implementation: Iustina
        - For the backend part, after building the models we determined that we need 5 controllers with APIs.
            - Schedule Controller
                - manages the schedule within the system. A schedule contains timeslots. This controller offers the **list** method (HTTP Method: GET) which returns a list of all timeslots in the system as a JSON array containing timeslot objects with their details, and the **add** method (HTTP Method: POST) which creates a new timeslot in the schedule after passing the validator.
            - Room Controller
                - is responsible for managing rooms in the system. It provides the **list** method (HTTP Method: GET) which

returns a list of all rooms in the system as a JSON array containing room objects with their details.

- Teacher Controller
  - is responsible for handling teachers within the system. It provides the **list** method (HTTP Method: GET) which returns a list of all teachers in the system as a JSON array containing teacher objects with their details.
- Subject Controller
  - manages subjects within the system. It provides the **list** method (HTTP Method: GET) which returns a list of all subjects in the system as a JSON array containing subject objects with their details.
- StudGroup Controller
  - handles student groups within the system. It offers the **list** method (HTTP Method: GET) which returns a list of all student groups in the system as a JSON array containing studgroup objects with their details.
  - subject objects with their details.

- Testing: Iustina (+Razvan for assert)
  - Unit testing phase:
    - Schedule Controller Testing
      - The Schedule Controller manages timeslots within a schedule in the system. In the unit testing phase, the following test cases can be considered for the list and post methods:
        - Test Case 1: List subjects
          - Description: Verify that the list method returns a list of all timeslots in the system.
          - Test Steps:
            - Prepare test data by creating multiple timeslot objects.(taking it from our storage)
            - Invoke the list method.
            - Verify that the returned list contains all the created timeslots.
        - Test Case 2: Create Schedule
          - Description: Verify that the post method creates a new timeslot for the schedule in the system.
          - Test Steps:
            - Prepare test data by creating a new schedule object.
            - Invoke the post method with the test data.

        - Verify that the returned schedule matches the created schedule.

- Room Controller Testing
    - The Room Controller is responsible for managing rooms in the system. In the unit testing phase, the following test case can be considered for the list method:
        - Test Case 1: List Rooms
            - Description: Verify that the list method returns a list of all rooms in the system.
            - Test Steps:
                - Prepare test data by creating multiple room objects.(taking it from our storage)
                - Invoke the list method.
                - Verify that the returned list contains all the created rooms.
- Teacher Controller Testing
    - The Teacher Controller is responsible for handling teachers in the system. In the unit testing phase, the following test case can be considered for the list method:
        - Test Case 1: List Teachers
            - Description: Verify that the list method returns a list of all teachers in the system.
            - Test Steps:
                - Prepare test data by creating multiple teacher objects (taking it from our storage).
                - Invoke the list method.
                - Verify that the returned list contains all the created teachers.
- StudGroup Controller Testing
    - The StudGroup Controller handles student groups in the system. In the unit testing phase, the following test case can be considered for the list method:
        - Test Case 1: List student groups
            - Description: Verify that the list method returns a list of all student groups in the system.

- Test Steps:
    - Prepare test data by creating multiple studgroup objects.(taking it from our storage)
    - Invoke the list method.
    - Verify that the returned list contains all the created student groups.

- Subject Controller Testing
    - The Subject Controller manages subjects in the system. In the unit testing phase, the following test case can be considered for the list method:
        - Test Case 1: List subjects
            - Description: Verify that the list method returns a list of all subjects in the system.
            - Test Steps:
                - Prepare test data by creating multiple subject objects.(taking it from our storage)
                - Invoke the list method.
                - Verify that the returned list contains all the created subjects.

        - With the use of asserts, we validated that our controller databases weren't corrupted by checking that the database wasn't empty or that some elements were still stored. (e.g. we asserted that room C309 was in the Rooms database)
- Models:
    - Implementation: Liviu
        - In order to make this application, we used the following entities: days, rooms, student groups, teachers, subjects, the schedule and its timeslots. We also implemented getters and setters to operate with these entities easily.
- Validator:
    - Implementation (Razvan + Liviu):
        - For validation, we created an utility class which handles multiple validation usecases. The issue here was that there were a lot of validations that needed to be done and you could easily get lost in them and forget some along the way. To mitigate this issue, we decided to start by adding the most granular validations (e.g. check if two classes are in the same room or at the same time) and later use those simple validations to compose more complex ones. We avoided the need of too many validations because we added constraints into

the FE component (e.g. a user may only select an end time which is higher than the class start time)

- Testing (Razvan + Liviu):
  - Even though we tried to cover all cases, we missed a few along the way, issue being noticed only after the implementation of the validator unit tests. For the unit tests, we used asserts from Java jUnit to confirm that our validation logic was correct.

Frontend:

- UI: Marie
  - For this project, we first thought of making a graphical interface using Swing, a Java graphics library, but we ended up preferring to make a web application in JavaScript, as we already had experience with this programming language, unlike Swing.

    The UI is very simple: the home page is directly the form that the user must fill in to create a new entry in the timetable. Two buttons at the bottom of the page allow the user to submit the selection, which leads to a page summarizing the characteristics of this entry, or to view the timetable in a new page.

    The UI consists of a JavaScript index and several pages in ejs format. We chose ejs over html because it enabled us to integrate JS scripts directly into these pages, modifying the form according to the criteria chosen by the user. For example, the group selection field does not appear if the course chosen is a seminar, as it is open to all groups in the class. We used JSON files to store the characteristics of courses entered in the timetable.
- Backend Integration: Razvan
  - As we decided to separate the application into two parts(FE + BE), we needed to also implement a communication mechanism between those two so that the application may work as expected. The work here consisted in configuring the FE and BE models so that they are compatible. After that, we also needed to make sure that we sent the right data in the right format from the FE part so that it is correctly translated into the BE models upon arrival there. The final part was to actually model the responses from the BE and return them accordingly and display them in the UI.
- Testing: Marie
  - For the UI test, we only performed unit tests. To do this, we used the Mocha JavaScript testing framework. As our pages don't display messages or return constant values, we decided to simply test them using their status code. So, for each of them, we expected a status code 200 indicating that the page had been correctly loaded and displayed.

    These tests were successful for the home page (the form) and the timetable page, but not for the course summary page. In fact, the page must be displayed by retrieving the information entered in the form by the user, which raises a DNS problem that we were unable to resolve.