WA25_exam3_Train

Web Applications – Exam #3 (deadline 2025-09-14 at 23:59)

"Train"

FINAL version - modifications highlighted with RED text.

Design and implement a web application to reserve seats for a train journey. The application must meet the following requirements.

For simplicity, only one train is handled. The train is composed of 3 cars only: one first class car, one second class car, and one economy class car. Each car has seats arranged in a grid format with R rows and S seats per row. The number of rows and seats per row depends on the car class: *first class* (R = 10, S = 2), *second class* (R = 15, S = 3), and *economy* (R = 18, S = 4).

On the main page of the website, accessible without authentication, it is possible to select a class from the available types (first, second, economy). The page shows the seat availability for the selected class. The graphical implementation of the two-dimensional seat visualization is left to the student. However, each seat should have a code indicating its row (1, 2, 3, ...) and position within the row (A, B, C, ...). For example, the first seat in the 10th row will have the code "10A," while the fourth seat in the 3rd row will have "3D", etc. Each seat in the visualization must display its status (either occupied or available) in addition to its code. The status has to be represented by colors (occupied in red, available in green). The seat visualization page must also display the total number of occupied seats, available seats, and the total number of seats in the car for the selected class.

After authentication (login), a user should be able to see a page to handle its reservations (remember, for simplicity, only one train is handled). The page is divided in two parts, left and right.

On the left side, the page should always display a button to make a new reservation and a list of the existing reservations identified by their ID (an integer number).

On the right side, the page will show, in the top part, the three buttons to select the class (first, second, economy) and, below the buttons, the two-dimensional seat visualization for the selected class, similar to the previous description, but with the possibility to see the user reserved seats and to select additional seats for a new reservation.

In detail, clicking on an existing reservation must update the right side of the page to show the actual occupied seats by that reservation in purple ~~orange (see later)~~, and the seats occupied by others in red. On the contrary, clicking on the new reservation button allow to select the class, to see the updated two-dimensional seat visualization for that class (see later) and to interact with it to perform a new reservation. Beware, a reservation can include *one or more seats but they can only be in the same class*.

The reservation happens by directly interacting with the seats using the two-dimensional visualization, taking into account the seats' status (and displaying them accordingly). Seats can have ~~three~~ four states:

- Occupied by others (red): seats cannot be requested since it is reserved by other users.
- Occupied by the user (orange): seats cannot be requested since it is reserved by the current user in another reservation.
- Available (green): seats can be requested by clicking on them. Once clicked, the seat should *immediately* be displayed as Requested on the screen. This action MUST NOT be communicated to the server.
- Requested (yellow) seats, which can be released if clicked (becoming Available again).

The seat visualization page should always display the number of occupied by others, occupied by the user, available, requested, and total seats in the currently selected class. These numbers should be in accordance with the actions performed by the user on the seats.

Additionally, the user can set the value of a field near to the map to automatically request a number of available seats. Such seats must be randomly selected among the available ones. Such value can be set more than once, and if some seats have already been requested by the user, also those seats are considered while changing the status of the seats (e.g., if 5 is entered and 3 seats are already requested, they are kept, and 2 more become requested; if 4 is entered and 6 seats are already requested, any of those 2 seats go back to available state).

In summary, the user can click on seats and/or set a value in the field repeatedly until he/she is satisfied of the selection. The user can also change the class and the visualization must be correspondingly updated, in this case the requested seats in the previous class must be made available again.

Note that while the user is interacting with the seats the action must not be communicated to the server, thus, it may happen that other registered users may request and/or occupy the same seats. This is normal and should neither be prevented nor notified to the user.

Once the seat selection operations are completed, the user can *confirm* the current request, without leaving the reservation page. Confirming the reservation means that the application will communicate the request to the server that, in case of success, will answer with the reservation ID. If, at the time of confirmation, all requested seats are still available, they should all become occupied simultaneously, and the list of reservations must be updated accordingly. Otherwise, the reservation operation will be entirely canceled, and the cancellation reason should be indicated by highlighting the seats already occupied by others on the two-dimensional seat visualization in blue for a duration of 7 seconds, and also a notification must be shown to the user.

A logged-in user may additionally delete his/her own reservations from the reservation handling page, thereby releasing all seats that were occupied by him/her in the specific reservation. Deleting the reservation will make the seats available again and it will update the visualization accordingly.

Note that, to be able to confirm reservations in first class, at authentication time the user must have used the 2FA procedure with a TOTP, otherwise the reservation will not be confirmed and an appropriate error message must be shown. Note that users can also choose to authenticate without using the 2FA procedure: in this case they can perform all operations except confirming reservations in first class.

The organization of these specifications into different screens (and potentially different routes), when not specified, is left to the student and is subject to evaluation.

For the 2FA procedure, for simplicity, use the following secret for all users that require it: `LXBSMDTMSP2I5XFXIYRGFVWSFI` (it is the same used during lectures and labs).
NB: Very important: using the same secret for all users is just for simplicity, the secret has to be stored separately, one for each user that requires 2FA, in the database (as seen during lecture examples).

Project requirements

- User authentication (login and logout) and API access must be implemented with `passport.js` and **session cookies, using the mechanisms explained during the lectures**. Session information must be stored on the server, as done during the lectures. The credentials must be stored in hashed and salted form, as done during the lectures. **Failure to follow this pattern will automatically lead to exam failure.**
- The communication between client and server must follow the multiple-server pattern, by properly configuring CORS, and React must run in "development" mode with Strict Mode activated. **Failure to follow this pattern will automatically lead to exam failure.**
- The project must be implemented as a **React application** that interacts with an HTTP API implemented in Node+Express. The database must be stored in an SQLite file. **Failure to follow this pattern will automatically lead to exam failure.**
- 2FA verification must be implemented by using the libraries explained during the lectures, and with the secret specified in the exam text. **Failure to follow this pattern will automatically lead to exam failure.**

- The **API** server must listen on **port 3001**, thus accessible at the URL http://locahost:3001 . Using different ports is not allowed and will result in much lower grading.
- The evaluation of the project will be carried out by navigating the application, **using the starting URL http://localhost:5173**. Neither the behavior of the "refresh" button, nor the manual entering of a URL (except /) will be tested, and their behavior is not specified. Also, the application should never "reload" itself as a consequence of normal user operations.
- The user registration procedure is not requested *unless specifically required in the text*.
- The application architecture and source code must be developed by adopting the best practices in software development, in particular those relevant to single-page applications (SPA) using React and HTTP APIs.
- Particular attention must be paid to ensure server APIs are duly protected from performing unwanted, inconsistent, and unauthorized operations.
- The root directory of the project must contain a `README.md` file and have the same subdirectories as the template project (`client` and `server`). The project must start by running these commands: "`cd server; node index.js`" and "`cd client; npm run dev`". A template for the project directories is already available in the exam repository. Do not move or rename such directories.
- The whole project must be submitted on GitHub in the repository created by GitHub Classroom specifically for this exam (which should include the title of this assignment).
- The project **must not include** the `node_modules` directories. They will be re-created by running the "`npm ci`" command, right after "`git clone`".
- The project **must include** the `package-lock.json` files for each of the folders (client and server).
- The project may use popular and commonly adopted libraries (for example `day.js`, `react-bootstrap`, etc.), if applicable and useful. Such libraries must be correctly declared in the `package.json` and `package-lock.json` files, so that the `npm ci` command can download and install all of them. Failure to do so will result in lower grading.

Database requirements

- The database schema must be designed and decided by the student, and it is part of the evaluation.

- The database must be implemented by the student, and must be pre-loaded with at least 4 users. Two users must have made two reservations each, one user with both the reservations in the same class, the other with reservation in two different classes. A third user must have a reservation in each one of the classes. A fourth user must have no reservations.

Contents of the README.md file

The README.md file must contain the following information (a template is available in the project repository). Generally, each information should take no more than 1-2 lines.

1. Server-side:
    a. A list of the HTTP APIs offered by the server, with a short description of the parameters and of the exchanged objects.
    b. A list of the database tables, with their purpose, and the name of the columns.
2. Client-side:
    a. A list of 'routes' for the React application, with a short description of the purpose of each route.
    b. A list of the main React components developed for the project. Minor ones can be skipped.
3. Overall:
    a. A screenshot of, at least, the **page to handle the user reservations**. The screenshot must be embedded in the README by linking the image committed in the repository.
    b. Usernames and passwords of the users.

Submission procedure (IMPORTANT!)

To correctly submit the project, you must:

- **Be enrolled** in the exam call.
- **Accept the invitation** on GitHub Classroom, using the link specific for **this** exam, and correctly **associate** your GitHub username with your student ID.

- **Push the project** in the **branch named "main"** of the repository created for you by GitHub Classroom. The last commit (the one you wish to be evaluated) must be **tagged** with the tag **`final`** (note: `final` is all-lowercase, with no whitespaces, and it is a git 'tag', NOT a 'commit message'), otherwise the submission will not be evaluated.

Note: to tag a commit, you may use (from the terminal) the following commands:

```
# ensure the latest version is committed
git commit -m "...comment..."
git push

# add the 'final' tag and push it
git tag final
git push origin --tags
```

NB: **the tag name is "final", all lowercase, no quotes, no whitespaces, no other characters**, and it must be associated with the commit to be evaluated.

Alternatively, you may insert the tag from GitHub's web interface (In section 'Releases' follow the link 'Create a new release').

Finally, check that everything you expect to be submitted shows up in the GitHub web interface: if it is not there, it has not been submitted correctly.

To test your submission, these are the exact commands that the teachers will use to download and run the project. You may wish to test them in an empty directory:

```
git clone ...yourCloneURL...
cd ...yourProjectDir...
git pull origin main  # just in case the default branch is not main
git checkout -b evaluation final # check out the version tagged with 'final'
and create a new branch 'evaluation'
(cd client ; npm ci; npm run dev)
(cd server ; npm ci; node index.js)
```

Make sure that all the needed packages are downloaded by the `npm ci` commands. Be careful: if some packages are installed globally, on your computer, they might not be listed as dependencies. Always check it in a clean installation (for instance, in an empty Virtual Machine).

The project will be **tested under Linux**: be aware that Linux is **case-sensitive for file names**, while Windows and macOS are not. Double-check the upper/lowercase characters, especially of `import` and `require()` statements, and of any filename to be loaded by the application (e.g., the database).