Web Applications – Exam #3 (deadline 2024-09-15 at 23:59)

"Concert Seats"

FINAL version - modifications highlighted with RED text.

Design and implement a web application to reserve seats at a set of concerts held in theaters. The application must meet the following requirements.

For simplicity, theaters are composed of a set of seats arranged in a grid format with R rows and C seats per row. The number of rows and seats per row depends on the theater: there are three theaters: *small* (R = 4, C = 8), *medium* (R = 6, C = 10), and *large* (R = 9, C = 14). Their size is written in the database. Any concert is held in only one theater. The association between concert and theater is written in the database.

Note that the system should be implemented in a generic way so that if a new theater (with its size) needs to be introduced, this should only require modifying records in the database, without the need to modify any application code (either server or client). In this case, it is assumed that the application is restarted (servers and clients).

On the main page of the website, accessible without authentication, it is possible to select a concert. After a concert is selected, the page shows the seat availability, with a two-dimensional visualization, for that concert. The graphical implementation of the two-dimensional seat visualization is left to the student. 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 12th row will have the code "12A," while the fourth seat in the 2nd row will have "2D." In the non-authenticated view, each seat should display its status (either occupied or available) and its code. The status is represented by colors, as detailed below. The seat visualization page must also display the number of occupied seats, available seats, and the total number of seats.

After authentication (login), a user should be able to select a concert from the available ones. After a concert is selected, the visualization must show the seat availability for that concert, with a two-dimensional visualization as in the previous case. In that visualization, additionally, the user can make a new reservation for *one or more seats* using one of the following two alternative methods (both are mandatory and must be implemented for the project submission).

1. Specify the number of seats to be reserved by directly entering the number of seats, and the system will automatically reserve the seats (if enough seats are available) which will become occupied. The logic for seat selection is left to the student (random, filling row by row, etc.). The operation is automatic and the user is not involved in the single seat selection.
2. Directly interact with the seats using a two-dimensional visualization similar to the non-authenticated seat visualization page (more details in the next paragraph).

With the first method, if the requested number of seats are available, they will be directly reserved (i.e., occupied), without requiring any confirmation. Otherwise, no seats are reserved and the user should be informed about the issue, including the current number of seats which are still available, after which the user can restart the reservation process using either one of the two previous methods.

With the second method, consider that seats can have three states:

- Occupied (red color): seat that cannot be requested.
- Available (green color): seat can be requested by clicking on it. Once clicked, the seat should *immediately* be displayed as Requested on the screen.
- Requested (yellow color): requested seat, which can be released if clicked (becoming Available again).

The seat visualization page should also always display the number of occupied, available, requested, and total seats present in the two-dimensional visualization. These numbers should be updated according to the actions of the user on the seats. It is ok to show the requested number of seats (as zero value) also in the seat visualization page.

Note that while the user is requesting seats, other logged-in users may request and/or occupy the same seat or seats. This is normal in this phase and should neither be prevented nor notified to the user.

Once the seat selection operations are completed using the second method (i.e., by clicking on the seats making them "requested"), the user can *confirm* or *cancel* his/her reservation. When receiving the confirmation, the server will occupy all the seats at the same time only if all requested seats are still available. Otherwise, no seats will be occupied, the reservation will be canceled, and the cancellation reason should be indicated to the user by highlighting in blue color (for a duration of 5 seconds) the seats already occupied by others on the two-dimensional seat visualization. Then, the visualization should be ready for the user to start another reservation process.

A logged-in user may additionally delete his/her own reservation (if it exists), thereby releasing all seats that were occupied by him/her. Deleting the reservation will make the seats available again and update the visualization accordingly. As a general constraint, an authenticated user can only make one seat reservation (including one or more seats, as before) per concert.

Additionally, implement a second server that is in charge of computing a discount percentage which is supposed to be used in next year's concert season (note: the discount is just a value to be shown, not to be applied nor used in any price computation; no price computation is required anywhere). To simplify the implementation of the discount computation, the discount will be a function of the sum of all the row numbers present in the codes of all occupied seats. For instance, if a user occupied seats 1A, 10B, 10C, 13C, the sum will be 1+10+10+13=34. The sum, divided by 3, plus a uniformly distributed random number between 5 and 20, rounded to the nearest integer, and clipped between 5 and and 50, will be the discount value (expressed in percentage). For instance: 34/3=11.333 + 8.4 (random number) = 19.733, rounded to 20: discount is 20%. If the value is less than 5, it will be assumed 5, and 50 if greater than 50.

Note that some users are marked as loyal customers in the database. For the loyal customers the discount computation is the same used for the other customers, but without the division by 3.

For any authenticated user, the discount value is requested and then shown after the reservation operation is successfully concluded (i.e., the requested seats have been occupied, with any method). A new discount value is to be requested and shown after each successful reservation.

Note that, since the discount is partially computed randomly, each time such a value will be requested from the server2, it can be different even for the same reservation. This is normal, and simply requesting new information whenever needed from server2 is perfectly acceptable.

The organization of these specifications into different screens (and potentially different routes) is left to the student and is subject to evaluation.

Project requirements

- User authentication (login and logout) and API access must be implemented with `passport.js` and **session <u>cookies</u>, 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. **<u>Failure to follow this pattern will automatically lead to exam failure.</u>**
- The authentication server must NOT access server2. Access to server2 must be done only from the client, as done during the lectures. **<u>Failure to follow this pattern will automatically lead to exam failure.</u>**
- 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. **<u>Failure to follow this pattern will automatically lead to exam failure.</u>**
- 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. **<u>Failure to follow this pattern will automatically lead to exam failure.</u>**
- 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.
- Access to servers different from the authentication one (e.g., server2), must be implemented by using properly designed JWT tokens.
- 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.
- The root directory of the project must contain a README.md file and have the same subdirectories as the template project (`client`, `server` and `server2`). The project must start by running these commands: "`cd server; nodemon index.js`", "`cd server2; nodemon 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. You may assume that nodemon is globally installed.
- The whole project must be submitted on GitHub in the repository created by GitHub Classroom specifically for this exam.
- 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, server, server2).
- 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.

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 6 concerts. There must be 4 users (2 of them must be loyal customers) who have a confirmed reservation of at least 2 seats each, in two different concerts; and 2 other users (1 of them must be a loyal customer) without any 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 the **seat selection page**. The screenshot must be embedded in the README by linking the image committed in the repository.
   b. Usernames and passwords of the users, including if they are loyal customers or not.

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').

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').

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; nodemon index.js)
(cd server2 ; npm ci; nodemon 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).