## Web Applications – Exam #2 (deadline 2025-07-06 at 23:59)

# “Restaurant”

PRELIMINARY version - the final version will be published on (about) Jun 23, 2025.

Please leave doubts and questions as COMMENTS in this doc, DO NOT DELETE or RESOLVE any existing comment, DO NOT edit text. Before adding doubts/questions, please check if a comment for your doubt already exists, please do not insert duplicate questions.

Design and implement a web application that allows users to configure personalized food orders from a restaurant menu. The application must satisfy the following requirements.

The restaurant offers a selection of **three base dishes** (pizza, pasta, salad) which differ in **size** (Small, Medium, Large) and the corresponding cost (5, 7 and 9 euros, depending on the size, regardless of the base dish).
Each base dish can optionally be customized with a variety of **ingredients**, defined by a name and a price (in euro). Small dishes can include up to 3 ingredients, medium up to 5, and large up to 7.
Ingredients (with their price in euro, in brackets) are: mozzarella (1.00), tomatoes (0.50), mushrooms (0.80), ham (1.20), olives (0.70), tuna (1.50), eggs (1.00), anchovies (1.50), parmesan (1.20), carrots (0.40), potatoes (0.30).
Each ingredient may require another ingredient or be incompatible with one or more other ingredients. Such **constraints** are:
- eggs are incompatible with mushrooms and tomatoes; ham is incompatible with mushrooms, olives are incompatible with anchovies.
- tomatoes require olives, parmesan requires mozzarella, mozzarella requires tomatoes, tuna requires olives.

Some ingredients have a limited number of portions available: initial availability is 3 mozzarella, 2 ham, 2 tuna, 3 mushrooms, 1 anchovies. No limit on the number of portions for the other ingredients.

Note that ingredient definitions, prices, constraints, and availability must be stored in a database. The system must be designed to support adding or modifying ingredients and constraints **without changing application code** (server or client). Just assume changes happen offline (app stopped, DB edited correctly, then restarted). Also, no need to include a web interface to perform these changes.

Any user (authenticated or not) can freely browse the list of base dishes and all ingredients with their prices, current availability, dependencies and incompatibilities.
Authenticated users can make their own order by selecting one base dish and its size, then the user can add/remove ingredients according to current availability and constraints. Ingredients must satisfy requirements and incompatibility rules. Required ingredients cannot be removed if they are dependencies for others. Also, incompatible ingredients cannot be added unless the incompatibility is removed. Attempting to violate constraints must result in a **clear message** or visual explanation (e.g., tooltip, modal). Note that the user can also change the size of the dish at any time: if more ingredients are present than the amount that would be supported by the new size, prevent the change with a suitable message until the change is possible.

When configuring an order, the user interface must show, for convenience, on the same page, both the **entire ingredient list** with the same information as the freely accessible page with the list of ingredients (in the left part) and the **current configuration**, i.e, the base dish, size and the selected ingredients (in the right part). The total price (sum of cost of base dish and ingredients) must be shown in the top right part, and it must be updated dynamically while the user interacts with the web application.

When the user submits the order to the restaurant, the restaurant checks that the order satisfies all the requirements in this text. If not, a suitable message is shown (e.g., "not enough tuna"), and the user is sent back to the configurator to fix their order, starting from the situation as left before submission. Note that the availability of ingredients must be updated by the restaurant only upon order confirmation.

Authenticated users can always check the list of their **past orders**: the list should show an entry for each order, with a detailed view containing the same information that was presented in the configurator.
**Tip:** an immutable configurator view can be recycled for this purpose if deemed appropriate.

Users can, in addition to the previous operations, also cancel an order already confirmed by the restaurant. Note that if ingredients with limited availability have been used in the order, the portions become available again. To be able to cancel an order, at authentication time the user must have used the 2FA procedure with a TOTP. Note that users can also choose to authenticate without using the 2FA procedure: in this case they can perform all operations except cancellation.

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).

## 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 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>**
- 2FA verification must be implemented by using the libraries explained during the lectures, and with the secret specified in the exam text. **<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.

- 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; 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 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.

## 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 sent two orders each, one for 2 Small dishes, the other for 1 Medium and 1 Large dish. The availability of the ingredients already selected in the orders must be adjusted **so that the application starts with the availability amounts stated in the text (to facilitate testing the application at the exam)**.

## 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 **ingredient configuration 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 administrators 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'), 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; 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).