

# PG6301 Exam April 2019

The exam should NOT be done in group: each student has to write the project on his/her own. During the exam period, you are not allowed to discuss any part of this exam with any other student or person, not even the lecturer (i.e., do not ask questions or clarification on the exam). In case of ambiguities in these instructions, do your best effort to address them (and possibly explain your decisions in the readme file). Failure to comply to these rules will result in an **F** grade and possible further disciplinary actions.

The students have 48 hours to complete the project. See the details of submission deadline from where you got this document.

The exam consists of building a web application using a Single-Page-Application (SPA) approach (with *React* and *React-Router*), using *NodeJS* (serving static files, REST API and WebSockets). **There must be only a single NodeJS instance**, serving both the frontend (e.g., HTML, CSS and *bundle.js*) and the backend (business logic, REST API and WebSockets). The main goal of the exam is to show the understanding of the different technologies learned in class. *The more technologies you can use and integrate together, the better.*

You must write your project in JavaScript and JSX. You are **NOT** allowed to use other languages that do transpile to JavaScript (e.g., TypeScript, CoffeeScript, Flow and Kotlin) or that do compile to WebAssembly (e.g., C and Rust). Your code folders **MUST** be structured in this way:

- **src/client**: code for the frontend (e.g., *React* components)
- **src/server**: code for the backend (e.g., RESTful API)
- **src/shared**: any code shared by both frontend and backend
- **public**: containing all your static assets, e.g., *index.html* and *style.css*
- **tests**: where you are going to put your tests

Note that the build tool (e.g., YARN) can create new folders, e.g., *node\_modules* and *coverage*, and that is fine. The application topic/theme of the exam will vary every year, but there is a set of requirements that stays the same, regardless of the topic/theme of the application.

The application has to be something new that you write, and not re-using existing projects (e.g., existing open-source projects on *GitHub*). If you plagiarize a whole project (or parts of it), not only you will get an **F**, but you will be subject to further disciplinary actions. You can of course re-use some existing code snippets (e.g., from *StackOverflow*) to achieve some special functionalities (but recall that you **MUST** write a code comment about it, e.g., a link to the *StackOverflow* page).

Once the deadline for the submission is passed, it is recommended (but it is NOT a requirement), to publish your solution as an open-source project (e.g., LGPL or Apache 2 license) on *GitHub*. This is particularly important if you want to add such project in your portfolio when applying for jobs. However, wait **at least** two weeks before doing it, as some students might have deadline extensions due to medical reasons.

You **MUST** have a *readme.md* file (in Markdown notation) in the root folder of your project. To avoid having a too long file, if you need you can have extra “.md” files under a “doc” folder, linked from the *readme.md* file.

In the documentation, you need to explain what your project does, how it is structured, how you implemented it and which different technologies you did choose to use. Think about it like a “pitch sale” in which you want to show a potential employer what you have learned in this course. This will be particularly important for when you apply for jobs once done with your degree. Note that the documentation **MUST** be written in a *readme.md* file. You are **NOT** allowed to write it in other formats, like .docx or pdf.

Furthermore, in the *readme.md* you also **MUST** have the following:

- If you deploy your system on a cloud provider, then give links to where you have deployed it.
- Any instruction on how to run your application.
- If you have special login for users (e.g., an admin), write down login/password, so it can be used. If you do not want to write it in the documentation, just provide a separated file in your delivered zip file.
- **State which parts of the requirements (discussed later) you have not done.**

The marking will be strongly influenced by the *quality* and *quantity* of features you can implement. For **B/A** grades, the more features you implement in your project, the better.

Note about the evaluation: when an examiner will evaluate your project, s/he will run it and manually test it. **Bugs and crashes will negatively impact your grade.**

As a general rule, you are allowed to add extra third-party libraries in your projects, even if those were not used during course. However, you are not allowed to use replacements for the ones shown in the course. For example, you are not allowed to use *Vue* or *Angular*, as in class we used *React*. Likewise, you must use *Express* and not other frameworks such as *Hapi*. However, you are allowed to add extra functionalities, like using for example *Helmet* to add security HTTP headers, *Axios* for HTTP connections and *ESLint* for code quality analyses. In case of doubts, discuss what you did and your choices in the *readme.md* file.

For the deliverable, you need to **zip** (zip, not *rar* or *tar.gz* files) all of your source files. Once unzipped, an examiner should be able to build it with YARN from a command-line. The zip file **MUST** be named in this way: *pg6301\_<id>.zip*, where <id> must be replaced with the id you get from the submission system: e.g., *pg6301\_123456.zip*. If for any reason you do not get a unique, anonymous id from the submission system, then use your own student id.

You can assume that an examiner has installed **YARN** (version 1.x) and **NodeJS** (version 10.x). To avoid issues with potential differences in library versions, make sure to provide the *yarn.lock* file in your delivered zip file. To build/run the project, you **MUST** use the scripts using in class, i.e.:

```
"scripts": {  
  "test": "jest --coverage",  
  "dev": "concurrently \"yarn watch:client\" \"yarn watch:server\"",  
  "watch:client": "webpack --watch --mode development",
```

```
"watch:server": "nodemon src/server/server.js --watch src/server --watch public/bundle.js",  
"build": "webpack --mode production",  
"start": "node src/server/server.js"  
}
```

If you have (very) good reasons to do modifications to those settings, explain them in the *readme.md* file. Your application should be started with a command like *"yarn dev"*. The web app **MUST** be accessible at *"http://localhost:8080/"*. An examiner is not supposed to install databases or other tools to make your app running, and you are **NOT** allowed to use Docker. Furthermore, your application **MUST NOT** rely on any external service: you can assume that an examiner will shut down his/her internet connection once *"yarn install"* is completed. Note: there is no requirement to use a real database. You can "fake" a database by using in memory objects (as seen in class).

This is not a course on *Web Design*, so it is not critical to have nice looking pages, or with very good UX. However, a minimal amount of CSS is expected (although no strong requirement on it) to avoid having pages that look too horrible.

Students are allowed to re-use and adapt any code from the main repository of the course: [https://github.com/arcuri82/web\\_development\\_and\\_api\\_design](https://github.com/arcuri82/web_development_and_api_design)

However, every time a file is reused, you must have comments in the code stating that you did not write such file and/or that you extended it. You **MUST** have in the comments the link to the file from GitHub which you are using and/or copying&pasting, e.g.:

[https://github.com/arcuri82/web\\_development\\_and\\_api\\_design/blob/master/les01/cards/code.js](https://github.com/arcuri82/web_development_and_api_design/blob/master/les01/cards/code.js)

For an external examiner it **MUST** be clear when s/he is looking at your original code, or code copied/adapted from the course.

Easy ways to get a straight **F**:

- Submit your delivery in a different format than *zip*. For example, if you submit a *rar* or a *tar.gz* format, then an examiner will give you an **F** without even opening such file.
- If you do not provide a *"readme.md"* at all.
- Use two different servers (e.g., a NodeJS for backend and a WebPack for frontend, or 2 different NodeJS for frontend and backend) instead of a single NodeJS instance serving both frontend (e.g., HTML/CSS and *bundle.js*) and backend (e.g., RESTful API and server-side of web-sockets).
- Not following the required folder layout (e.g., **src/client** and **src/server**).
- Submit a far too large zip file. Ideally it should be less than 10MB, unless you have (and document) very good reasons for a larger file (e.g., if you have a lot of images). Zipping the content of the *"node\_modules"* folders is absolutely forbidden (so far the record is from a student that thought sending a 214MB zip file with all dependencies was a good idea...).

Easy ways to get your grade **strongly** reduced (but not necessarily an **F**):

- Skip/miss any of the instructions in this document.
- Your *zip* file is not named as instructed, e.g., submit something called *exam.zip*.
- Bugs.

- Home page is not accessible at: <http://localhost:8080/>
- Some parts of the exam are not completed, but it is not specified in the “*readme.md*” file. For example, if you do not have working web-sockets, you **MUST** state so in the *readme.md*, e.g., “Requirements for a **B** grade were not satisfied because I did not include web-sockets”.
- If you use empty spaces “ ” in any file/directory name. Use “\_” or “-” to separate words instead.

**Necessary but not sufficient** requirement to get at least an **E** mark is:

- Write a home page with React.
- At least 2 other React pages that can be accessed via React-Router.
- At least one page should have some “state”, whose change should be triggerable from the GUI (i.e., there should be some actions for which a React component should be re-rendered and produce different HTML).
- From each page, it should be possible to go back to the homepage without having to use the “Back” button in the browser. In other words, do not have pages in which, once reached, it is not possible to navigate out of them. Example: if you are displaying a list of items, and then you have a link to a page to display the details of a specific item, then from such page there should be a link back (or at least to the homepage).

**Necessary but not sufficient** requirements to get at least a **D**:

- Create a RESTful API handling at least one GET, one POST, one PUT and one DELETE (*besides* the ones for authentication/authorization of users), using JSON as data transfer format.
- The REST API **MUST** follow the best practices for API design (e.g., on the naming conventions of the endpoints).
- The frontend **MUST** use such API (e.g., using *fetch*).

**Necessary but not sufficient** requirements to get at least a **C**:

- You need to handle authentication/authorization, which **MUST** be session-based via cookies (as seen in class).

**Necessary but not sufficient** requirements to get at least a **B**:

- Add WebSockets to your app, with the frontend using it.

Regarding **A** grades: recall that the grade **A** means “*Outstanding*”. Doing everything requested in this document is not necessarily enough to be outstanding. You should add extra functionalities to your application. The choice of functionalities is up to you. However, if you are aiming for an **A** grade, then you **MUST** explicitly say so in the *readme.md* file, specifying all the extra work and functionalities you added to the project and that you want to get considered for the possibility of an **A** grade. Adding some extra functionality does not necessarily imply that you will get an **A**: it all depends on their *quality* and *quantity*. Try to do as much as possible within the given amount of time. Note: in some special cases, an examiner might still decide to give an **A** even if there is no extra work.

## Testing

You must write test cases for both the frontend (e.g., React components) and the backend (e.g., REST API and WebSockets) in your app. Which tests and how many you should write is up to, but there are requirements on code coverage. Your tests need to be written with Jest (using extra supporting libraries like Enzyme and SuperTest). Coverage should be calculated over **ALL** your source files in the *src* folder. In the *readme.md* file, you need to report the value for “% *Stmts*” of the “*All files*” entry when running “*yarn test*”. Note that an examiner will run such command, and verify that what you wrote in the *readme.md* does match it. When calculating coverage, it might be that some files have 0% coverage, whereas others have 100%. What is important is the average value given by the “*All files*” entry.

Test coverage requirements:

- Grade **E**: at least 10%.
- Grade **D**: at least 20%.
- Grade **C**: at least 30%.
- Grade **B**: at least 50%.
- Grade **A**: at least 70%.

Note: this implies that, if you have no tests (so coverage 0%), then you will get a straight **F** regardless of how good your application is.

## Application Topic

The application topic for this exam is about a social-media website (i.e., a very simplified Facebook). Requirements:

1. When the application starts, you must have some existing fake/test data representing valid users.
2. Should be possible to register new users.
3. Each user should have a page displaying his/her information (e.g., name, surname, date of birth and location).
4. A user should be able to post new messages on his/her “timeline”, which should be displayed in chronological order together in the same page with the user’s info.
5. Should be possible to search for existing users.
6. Users can send “friendship” requests to other users. This latter will decide whether to accept it or not.
7. Two friends can see each other timeline / user-details, but not the ones of the other users they are not friend with.
8. The home of a user will be the merged timeline of all of his/her friends, in chronological order, updated in real-time (e.g., using WebSockets).
9. Should have a live-chat (using WebSockets) for friends.
10. When a message contains a URL (e.g., link to an external web page), that should be displayed as an actual clickable link. Pay particular attention to the security aspects of having such a functionality.

Add any extra feature relevant to such type of system. This is going to be very important for **B/A** grades.