Frequent Errors in project submissions
Web Applications, a.y. 2024/25

Advice coming from previous editions of the course
v. 2.1

Note:
The project can be submitted in any condition regardless of the satisfaction of the points described in the following, which are not a necessary condition to pass the exam (except otherwise stated, and except for following the correct submission method). However, in general, addressing all these points should result in a good mark because the most serious problems are typically avoided.

QUICK CHECKLIST

- Application must work with more than one client
- No unreasonable assumptions can be made on input data, identifiers, etc.; if in doubt, ask.
- Be careful about where operations must be done (client, server, both)
- Perform all the necessary authorization checks in the server
- Use the correct HTTP methods in the APIs
- Clean and well-organized code, avoid excessive usage of useEffect
- Only use React components, no bare JS DOM methods of any kind (including alert())
- Reasonable management of error conditions
- Always provide a way to navigate the pages, without using browser buttons
- Check the project on a case-sensitive system

APPLICATION LOGIC PROBLEMS which generally lead to loss of points or <u>exam failure</u>

- It is **UNACCEPTABLE** that the application does not work when two separate users operate at the same time. Such condition will almost certainly lead to **exam failure**. Web applications are, by their nature, multi-client and multi-user, so this problem means that the student did not understand the fundamental concept underlying the course. An example of this type of error is storing information related to a specific client without associating the unique information of the client itself, for example using a generic global variable in the server to store information. In these cases, think about whether it is better to keep the information in the database or in the session object instead.
- It is **UNACCEPTABLE** to make assumptions regarding the value of the identifiers present in the database. Such assumption will probably lead to **exam failure**. For example, if the project requires 4 users, the code must NEVER assume that their identifiers are 1,2,3,4, let alone perform mathematical operations on these identifiers, e.g. to derive an index into a vector etc. Identifiers must be transferred as properties of the elements of which they are part (e.g., in the case of users, together with the name). In a generic application the identifiers could also be non-consecutive and even non-numeric.
- IDs must NOT be created on the client when they must be later used as identifiers (unique keys) in the DB: the IDs MUST be created by the server, and then returned to the client. Please do not use UUID libraries to create permanent unique IDs (i.e, to be stored in the DB), as they are not guaranteed to be unique (even if collision might happen with very low probability).
- For the IDs that needs to be determined when creating a new unique element, preferably use the mechanism provided by the DB (e.g. auto-increment column). Other solutions are

generally wrong and lead to a lower final mark (creation/calculation on the client, calculation of the max on the DB column, etc. – such strategies are reasonable only for a temporary ID, soon to be overwritten by the actual one coming from the DB/server). Remember that the application can be used by multiple users at the same time, who may not even work on the same data, but the uniqueness of the ID must be guaranteed. Furthermore, the ID generated by the DB is, typically, not a value that can be decided by the user; also, it cannot be modified directly by the user, so its value should not influence the application logic. Also, in general, it should not be explicitly displayed in the application interface.

- Always use the correct method when writing and calling HTTP APIs (e.g. never use POST instead of GET, GET instead of POST, PUT instead of POST, etc.): each method has its own correct semantics.
- It is always necessary to check, on the server side, the roles/permissions of those who read/write information in the DB (additions, modifications, etc...). Initial authentication is not enough: it is necessary to verify that the authenticated user (that can only come from “req.user”, not from the client) in the session has permission to perform the operation (e.g. by adding a “where” condition to an SQL query or running a specific additional db query in the server-side code). This part is very important and, if wrong, will yield a quite low final mark, if not exam failure.
- Those who do not pay attention to authentication/authorization mechanisms on the server will be severely penalized in the final mark, especially in the “Web applications” course which is part of the CYBERSECURITY course of study. In particular, this refers to students who confuse client-side verification with server-side verification, which is a very serious mistake. Anyone orally saying: "**I checked on the client that action X cannot be performed...**" did not understand the problem, and will be severely penalized in the final mark (or resulting in an exam failure). Verifying the correctness of the values and the possibility of carrying out certain operations ONLY on the client side is **CONCEPTUALLY WRONG**, as clarified during the lectures. Moreover, it is one of the reasons why there are so many security problems in web applications nowadays. Therefore, such students will be heavily penalized in the final mark of the exam, and this may also include exam failure.
- AVOID writing custom regular expressions (regex) to verify the formal correctness of information/data that can be easily managed by a library: you waste time, you are very likely to make mistakes, and during the exam, often it will happen that a case that has not been considered will be tested. Example: verify correctness of email or date. A programmer in general does not have to spend his time writing regex to consider leap years and other peculiarities. The programmer must use a (reasonably reliable) library, have the dates converted and transferred into some standard format if necessary (e.g. ISO string format).
- DO NOT write a single huge component with the logic of almost all the entire application: unreadable, unmanageable, and almost certainly buggy (e.g. difficult to manage useEffect dependencies etc.)
- DO NOT make additional assumptions which are not in the exam text or clear from the text, especially if they diverge from the text, neglecting even just a part of it. For example: if you are asked to add a new course by specifying (name, code, credits), taking a course from a predefined list is wrong: if I choose it from a predefined list I CANNOT add the name I want. Moreover, I do not implement (i.e. I unduly simplify the project) a part of controls in the form that defines the course, e.g. that the number of credits is a number, and greater than 0.
- DO NOT invent weird data formats to compact information transfer from client to server and vice versa (e.g. separating fields manually with commas or other special symbols, which leads to the necessity to manage these cases with additional code). In general, the text fields of a form (titles, texts, etc.) must be able to support spaces, semicolons, commas and various

symbols. Where do you live? In a city named “Ascoli Piceno” (name with space). In Mondovi' (with the accent/apostrophe), and so on. Use JSON to serialize/deserialize information contained in Javascript strings, as explained in the course. It will manage everything for you in a correct way.

- Avoid using useEffect if you don't actually need it. For example: avoid using it to react to a change in the value of a form field. If the form is controlled, the handler can manage this. Many useEffects in the code, especially for cases that are easily handled differently, make the components difficult to understand, modify and maintain. Instead, do use useEffect in all cases where it is actually needed (asynchronous data loading etc.). In general, avoid using useEffect with location or history dependencies to handle page switching when pagination has been implemented via a Router. (Instead, it is expected to use useParam to show different content depending on parametric portions of the URL)
- If something different is not explicitly required, the operations that work on the DB following the call of an API can include multiple queries executed separately (i.e. not in one transaction, which is complicated and difficult to implement in sqlite3, so it is not required for the course). For simplicity, it is assumed that between one query and another during the execution of the code that corresponds to a server API, no changes occur to the DB, as stated in the slides of the lectures.

  NB: However, it <u>cannot be assumed</u> that between a call to an API and another call to an API the DB necessarily remains without including modifications requested by others.
- As a result, an atomic operation (at the level of the application logic) must be accomplished within a single API. Consider this when designing APIs. The typical case is the creation/modification/deletion of complex information which requires multiple server-side operations (e.g. multiple queries). For example, creation/modification/deletion of a questionnaire with all its questions, of a study plan with the related courses, a user with all the resources connected to it, etc. It is a seriously flawed logic to make multiple calls from the client to the server APIs to perform an atomic operation at the level of application logic. This way of operating will lead to a low final mark.
- For the same reason, in general you should NOT create an API (and therefore a call from the client to the server) for each operation to be executed on the DB. In other courses something similar may have been suggested (typically in other contexts), but for this course you must refer to the guidelines provided during the explanation of how to design APIs that can be called directly from the browser.
- If it is necessary to make changes to a complex structure that must respect given constraints, you must FIRST check that the constraints are satisfied for the situation that will be created and, ONLY AFTER THAT check, you can make the changes (e.g. queries) that implement them, to avoid error conditions that become difficult to rollback (especially in the absence of transactions, which it is preferred not to use for simplicity).

USER INTERFACE ISSUES: if serious they can cause mark reduction. They are very annoying during functionality tests at the exam

- If the user did not do anything yet, no errors should appear. For example, if a form still needs to be filled out, it must not show errors (e.g. form fields bordered in red) before using it. Another example: login with username and password that are red as soon as you land on the page simply because they are empty. Only after user actions, e.g. pressing the login button (or “Send” or form submission) any errors should appear (such as: an email is required, the email is not valid, the password field is empty, etc.). This can be partially relaxed to dynamically check validity while typing, but not before starting typing.

- For the login form: if, after sending the credentials, it is not possible to authenticate (for example because the server rejects the credentials) this situation must be clearly shown to the user, otherwise the user will not even notice that the credentials have been sent to the server, and that there was a problem. If there are multiple places to log in (e.g. specific page but also username and password fields always visible at the top), ALL of them must show a message in case of problems, not just the one on the login page.
- Do not insert authentication methods unless requested in the text (e.g. "login as guest"): using or giving the impression of having to authenticate, even without credentials, when not requested, will be considered a mistake, and cause a reduction of the final mark.
- Test ALL ways of using form fields, in particular HTML5 fields: if you write the number directly instead of using the arrows in the numeric field (e.g. “course credits” numeric field), writing the number directly MUST work, otherwise the mark will be reduced because some features cannot be tested. Also note that numbers must not be assumed integers except where it is clear from the context (e.g., number of physical objects that cannot be split) or explicitly specified.
- DO NOT make unspecified assumptions about data that should be entered by the user: an identification code, for example, is NOT necessarily numeric, unless otherwise stated. The internal ID of the application can be numeric, but the user of the application is NOT interested in this internal ID, and such an ID must not be used as part of a request in the user interface, i.e., the user should not be asked to enter such an ID in a text/numeric field. If a selection is needed, it should be presented in a user-friendly format. If the code has a meaning for the user and it can be manually entered by the user (e.g. “course code”), do not assume that it is only numeric, unless this is explicitly stated in the text. The code of a generic object/element/product is NOT necessarily numeric. It may be convenient to implement it as a number, but note that even in this case it is conceptually wrong to acquire/manage it with fields of a form with e.g. type=number.
- DO NOT make unnecessary, unsolicited, or incorrect assumptions about the data: e.g. at least 6 characters for a course name (examples of valid short course names: "math", or "Greek"). In general, it is NOT the developer's job to think about this aspect if it is not requested in the specifications. Testing whether it is empty or not makes sense (possibly removing leading/ending spaces), but the number of characters in general does not. Another example: specify a place (min 6 characters): the city of Rome and Rho could not be indicated.
- Avoid narrowing the possible choices unless indicated by the specifications. Example: to book an appointment for which a duration must be specified, the duration (e.g. in minutes) must NOT be requested as a multiple of 10, unless explicitly stated in the specifications, or stemming from the logic of the problem. If it is a user interface issue (because it would result e.g. in a combo box that is too long), change the interface (e.g. numeric or text field which will be later checked for correctness). Example: The number of credits for a course must not be predetermined (e.g. 6 8 10 12), but free, possibly also with a comma/dot to indicate a fractional value, unless otherwise stated in the specifications. If in doubt, ask for clarification. In general, constraining these types of values involves more work, which is not required and not evaluated, and often does not allow to easily test the application at the exam.
- If the possible choices of a field are limited (e.g. only one product, person, course code, or anything else already existing in the application) it is advisable to use an appropriate element in the interface, e.g. a combo box (drop-down menu) and NOT a free input field. It is even worse to ask the user to enter the internal application ID of the item or showing only the ID as the identifier for the choice (how would the user know what the ID corresponds to?). However, do not assume that, since the user can only use the drop-down menu, the correctness of the value should not be checked on the server side: missing server-side checking is a very

serious error especially in a Cybersecurity course of study, and may even cause exam failure, as explained before.

- If the selection of a single element among N possible choices is required (e.g. a single user among those available), it is necessary that the selection operation, internally, always identifies the single element uniquely. Identifiers that are not guaranteed to be unique (e.g. name, or first and last name, etc.) must not be used internally.
- The selection of an element must be easily understandable by the user: e.g. you should not ask the user to enter the internal database identifier ("id") to select a user or other object. To avoid this, an alternative could be a combo box or selection box with pre-filled values among the possible ones (e.g. name, surname, email, type, etc.). Obviously, internally, the individual selections will be associated with a unique identifier.
- If the number is, semantically, a string, the field must NOT be numeric (NO type=number). Examples: credit card number, phone number, product code. This is a very basic mistake (basic programming knowledge) and will be penalized in the final mark.
- Always specify the measurement units of what you are asking for, at least in a string beside the form field. For instance: Duration: what does "1" mean? 1h, 1min? In this case it may also be fine to use the HTML5 numeric field (type=number), but the user must also be able to write a value directly without using the arrows to select the number. If the user would like to set 300 as value, it must not be necessary to make 300 clicks or wait for all the numbers running from 1 to 300 with the arrow.
- Date/time fields: avoid making them plain text without examples: how do the user write if there is no example? (Do I use /, - or what other symbol between year, month, and day? and in what order, Italian or English? Is the month in numbers or letters?) In general, prefer the type=date or use libraries to have a box that opens which allows selecting the date. At least an example about how to write it should be present, in the field or next to it, e.g., "YYYY/MM/DD".
- You must NOT check the contents of the password during login but only when creating a new password (this applies only if the assignment requires to implement user registration). Checking during the login phase is generally incorrect, it gives information on how the passwords are made to a possible attacker (e.g. minimum 6 characters, at least one capital letter, etc.). Also, this is inconvenient during the evaluation phase of the application because it does not allow to use simple wrong password to test the failed login case.
- It is convenient that the forms also start the submission by pressing the ENTER key on the keyboard, not just with the click on the button (it saves time during the test at the exam and is also good practice, e.g., login). To do this, just handle the onSubmit event in the HTML form tag.
- DO NOT use the browser window creation functions, **NEVER** (e.g. NEVER use alert(), prompt(), confirm()). Problems: they are not integrated with React application management, the appearance is potentially different for each browser, they can be disabled in the browser, the browser stops them if there are too many, etc. The final mark will be reduced if you show elements which are not managed by React.
- DO NOT use functions to reload the browser page from Javascript (e.g. window.location.reload()), NEVER DO THAT. The application, after being loaded the first time, must not require reloading, otherwise it is not a Single Page Application (SPA). Such behavior will be heavily penalized in the final evaluation mark.
- Form fields: avoid making them non-editable to force the user to use buttons, e.g. + and - in a numeric field. It shouldn't be necessary to make 1000 clicks to write the value 1000...

- The text in the form text fields can be anything, unless otherwise noted, and in general such fields must be able to support spaces, punctuations, etc. In addition, the application should not fail without giving clear error messages, e.g., the presence of an unexpected space (which may be invisible and/or entered by mistake) or other characters.
- When an operation has been performed, feedback should be given to the user: either something changes in the interface (adding/removing an element in the interface), or a confirmation/result message is shown. In particular, in case of error, the user should understand what is wrong. (Managing all possible errors is not easy, but a good compromise can be reached, especially by following the rules above potential problems can be minimized). This does not imply to use the "optimistic update" approach, just give a reasonable feedback to users that may be ignorant of the inner workings of your application.
- Use error logging via console.log() for development only. The delivered project should not use any of them to handle conditions that may occur during application execution. Any errors should always be shown to the user (e.g. by defining a state and a handleError function that sets it).
- Test the application even when the API server is down. Often, during tests at the exam, the server does not work for various reasons (e.g. a npm package is missing, the DB file name is spelled incorrectly due to capitalization, the server crashed). Make sure this case is handled in some way, if possible.

APPLICATION NAVIGATION

- Make any navigation button CLEARLY VISIBLE, not hidden in places such as pop-up menu which are not shown by default (e.g. user icon): it is a waste of time for those who test/use the application, risking it being considered a missing feature in the project.
- If there is a legend of symbols (symbols that are not intuitive in certain applications) it must always be visible or at least accessible without changing the view from all the application screens where the symbols appear.
- There must always be a button to navigate back or to a known view/condition, e.g. "Cancel/Back" (preferable), or at least "Home", displayed very clearly, not in places known only by the website (e.g. "MyApp" or various logos are not intuitive as a way to navigate to the home page). If the button/link cannot be identified easily, there is no alternative for the tester to use the browser's Back button or reload the URL, which is not recommended, as explained in the course, and will not be done unless there is no other choice.
- Avoid inserting artificial delays on the server side to show the loading status, because this slows down the application test. If done, keep the delay value to about 100 ms maximum, not more. While inspecting the application code, the student will be able to point out the presence of such feature (e.g., "loading" states).

GOOD PRACTICES

- Show if the user is authenticated or not in some places always visible in the user interface. If the user has a role (student/administrator/other) also show the current one.
- Always show the header of tables and lists: what is the value in the first, second column etc...? Sometimes it is intuitive, but many times it may be not, particularly when testing the application with "random" data to type content fast.
- Graphical appearance: avoid excessive graphical effects (box shadows etc...) which risk not being well tested and appear bad when content is bigger than expected (a list becomes longer

than expected, there are many elements, etc.). Such graphical appearance does not grant additional points in the final mark. The default templates (from Bootstrap or other similar libraries) are already beautiful enough.

SUBMISSION (some problems may lead to loss of points or inability to evaluate)

- The tag to use is "`final`" (written **all lowercase, without quotes, without spaces**), applied to the commit to be evaluated, which must be in the main branch. We recommend checking that everything is OK by checking for the presence of the tag from the **GitHub web interface**. If the tag is not present in correct form, the submission will not be considered, being indistinguishable from the case in which the student does not want the submission being evaluated. Also note that the date/time included the commits is generated locally on the PC before the "push" (this is due to how git works), so **such date/time does not demonstrate anything regarding the time of the submission (in particular, if it is before the deadline or not)**. The ability to submit (i.e., to push commits to the repository) will be automatically closed at the deadline date/time.
- Before submission, test the project on **CASE SENSITIVE** system. MacOS and Windows ARE NOT. MacOS may seem case sensitive, but it is NOT: the file system remembers the case of the names but the files can be opened by programs regardless of the case. Pay particular attention to the db file which cannot be read due to capitalization, and which often does not display an error on the client side of the application if the error handling strategy did not consider this case. Be careful with import/require. If the DB file name is incorrect, the application usually does not work and it is difficult to understand why (often, the message is "internal server error", which does not help). Testing on a case sensitive system is easy: just run the project in a Linux virtual machine. This can be done either on your system (e.g. laptop) or it the cloud: just create and run a virtual machine, test it, then close and destroy it (many major providers offer 12-month trial period and often a free tier forever which is more than enough to test the application)
- Remember to include all packages in the `package.json`, both for the server and for the client, otherwise the application is not testable. In other words, never install any package globally (except nodemon, which is not needed in testing anyway, since no code modifications will be done).
- Always include the `package-lock.json` file. This is especially important for the Cybersecurity course of study as it contains the hash of the packages to be verified during installation.
- Being messy in writing code is not "one's choice". The clarity and order of the code is part of the evaluation. Do not submit a single (or almost) single file with all the components inside. On the other hand, do not do the opposite. Grouping components in the same file by logical functionality is perfectly fine (e.g. administrator/user, or role in the user interface - e.g. navigation bars or menus). As a rough rule of thumb, files of 500+ lines of code are a sign of poor organization.
- CLONE the repository for the exam and DO NOT modify the folder structure, do not rename, or move files and directories, in particular the folders of the client, servers, and README.
- For the purpose of the exam, it is suggested to use the same password for all the users (of course, each one stored with a different salt). It is understood that this is not a good security practice, but this will greatly simplify the manual verification of the application (Also, use simple values for passwords, such as "password", or "pwd"). NO points will be deducted from the final mark for this. On the contrary, this is appreciated in the interest of saving time! Also, having a default user and password already pre-filled in the login form is also greatly

appreciated. This can easily be done by initializing the state of the form component, as suggested during the lectures.

- For the purpose of the exam, using simple and short usernames/emails, and a regular pattern, especially for emails (e.g. u1@p.it , u2@p.it , u3@p.it), is highly appreciated. In general, remember that if the teacher has to spend 2 additional minutes in testing due to copy/paste of difficult-to-remember usernames/passwords/strings, in an exam seat with more than 100 submissions this will typically to one more day of the oral exams (i.e., some students waiting one more day for their turn).

README.md: how to write it

- Write your real first name, last name and student number instead of the generic wording `Student: s123456 LASTNAME FIRSTNAME`. Leave the `Student:` part unaltered (in English) which can be convenient for automatically processing README.md files. Note that the association between projects and student identities does not rely on this, but it is nevertheless convenient to have such information set correctly.
- For the courses delivered in English (e.g., Web Applications), the rest of the document must be written in English. Also, for convenience, avoid modifying the titles already inserted in the example README.md file (`Application Routes, API Servers`, etc.)
- CAREFULLY check the username/password stated in the README.md file. For example, if the email domain is forgotten, or no password is specified, the application cannot be tested, and it will not be evaluated. Do not report the hash of the password in the README.md file: it is useless and indicates a lack of understanding about the best practices of password storage.
- Remember to specify the role of the users (if applicable, depending on the exam specifications): administrator, manager, user, creator, etc., or user type A, B, C, etc..
- Check that the screenshot(s) included in the README.md are linked correctly (path and file name with the right upper or lowercase for each letter in the name). This can be easily verified on a case-sensitive operating system with the VSCode's PREVIEW feature on .md files
- ".md" is a well-defined, although simple, format. It is not just a nice way of writing text files. If you do not respect the syntax, it will not work in the VSCode preview mode, that is the one used by the teacher to read the file at the exam. Check if everything is ok using the VSCode PREVIEW mode. There are also editors to make writing the file easier: search for "markdown editor" on search engines. Some can also be used online, just copy/paste the resulting text.
- For DB tables, DO NOT report the SQL statements used to create the tables: it is difficult to read. Write the table name and the names of the columns (possibly use meaningful names), and the important information in brackets (e.g. primary key if not obvious) or unique fields.
- For APIs specifications: if request/response examples are included, write SHORT examples, avoiding lists of N objects with M fields each. Consider reducing the number of newlines in the JSON to improve the readability of the README.