

Web Applications I – Exam # 3 (deadline 2023-09-07 at 23:59)

“Guess Who?”

PRELIMINARY VERSION – The final version will be published on 2023-08-23.

Please, ask for clarifications using the “Comment” feature. Do NOT resolve comments, they are useful for everyone to see.

Design and implement a web application for playing a one-player version of the famous “Guess Who?” board game.

In the game, the player must guess the identity of a specific ‘item’ amongst a wide set of items, by making a set of successive guesses. The gameplay defines a catalog of items, and items are characterized by a set of properties, which may assume different values. Each item in the catalog has a *unique* combination of values for its properties. All properties must be defined (i.e., have a value) for all items.

A catalog of items may cover any domain, such as famous people, cartoon characters, dinosaurs, constellations, pokemons, sports players, professors, animals, flowers, etc. You are free to use your fantasy, and the application should implement only one domain of your choice. The set of properties and their values must be customized depending on the type of items. **The number of properties and values should be defined taking into account the characteristics of the catalog.** For example, if the selected items were wild animals, the properties could be “nutrition” (with values: carnivore, herbivore, ...), “legs” (with values 0, 2 or 4), “flying” (values yes or no), “color” (with a wide range of values), etc.

The gameplay consists of a series of matches of various difficulty (easy = 12 displayed items, medium = 24 items, hard = 36 items). At the beginning of each match, the full list of possible items is displayed (by showing a picture for each item, in a bi-dimensional graphical grid), and a “secret item” is randomly chosen by the server. The following two conditions apply:

1. To avoid cheating, the identity of the secret item must **never** be transferred to the client.
2. The set of values associated with the properties of each item should not be displayed, as the player should be able to understand them from the item’s picture.

Each match consists of a set of guesses. In each guess, the player will select one property, and **select** one possible value of that property **out of the possible value options**. Upon confirming this selection, the application will tell the player whether the guess was correct or not (i.e., whether the secret item’s property value matches the player’s guess). The application will then ‘disable’ all the items on-screen that are incompatible with the guess. For example, if the player guesses “flying=yes”, and the secret item is a flying animal, the application will confirm the guess, and will disable all non-flying items. If the secret item is not a flying animal, the application will communicate that the guess is wrong, and disable all flying animals. Note: for any possible sequence of guesses, the secret item will always be in the set of still-enabled items.

To terminate the match, the player selects one of the still-enabled items (it can be done only once, at any time during the match, and it will terminate the match; **it will be the only option if there is only one remaining item**). If it corresponds to the secret item, the player wins, otherwise he/she loses.

Commented [1]: How many properties should an item have?
(A “win” with score “0” suggests having at least 12 properties.
And it seems a bit difficult to find too many visible properties for items of any domain! For example, “nutrition” for animal items may not be seen in pictures.)

Commented [2]: I don’t agree with your computation. A property like male/female would cut in half the size of the items, so the number of guesses is logarithmic in the size of items (if games are well played).
Nutrition cannot be “seen” from the picture, but the ...

Commented [3]: Thank you professor for your informative response.
Based on the facts stated in the text, in case of an eas ...

Commented [4]: Oh, I think I made a mistake about the “guess” rule. Maybe it’s not needed to guess on distinct “properties”. But it must be a distinct “property: value” ...

Commented [5]: If I have 12 items, and I use more than 12 guesses, I’m really a bad player. That is perfectly legal and allowed... I’ll get 0 points if I make ...

Commented [6]: But I don’t see the relationship with the number of properties. In case of animals, we may have: legs, flying, color, carnivore, mammal, swimmin ...

Commented [7]: The problem is that all the properties must be understood (=seen?) from the pictures, and this may make it difficult to find more properties. ...

Commented [8]: The number of items that can be represented is the PRODUCT of the number of values for each property, not their sum. ...

Commented [9]: To close this thread, the official answer is that there are no minimum nor maximum limits to the number of properties or values, it’s up to ...

Commented [10]: OK. Just to clarify my comments for “everyone”: the discussion here is not about how to play the game well. It’s about how to design and implemen ...

Commented [11]: Agreed

Commented [12]: Should the value of the property be chosen from a restricted set of possible values (displayed when the property is selected) or entered ...

Commented [13]: Yes, usability suggests that the user should select from a list allowed values

Commented [14]: Is okay to keep images in a folder and use named import from a js file?

Commented [15]: You are free to use the method you prefer for loading images.
(named imports are hard-coded and seem boring to ...

Commented [16]: If the user makes a correct guess that eliminates all the still-enabled items in the screen except the secret item, and at this point the only thing ...

Commented [17]: The user can get more/less than 1 point as score, in that situation.
The score (if the user wins) is computed as $n_items -$...

Commented [18]: To clarify further, each “wrong guess” makes you lose 1 point. The final “select” does not count as a guess, and terminates the match.

At the end of the match, a score is assigned:

- 0 points if the player loses;
- an integer non-negative score if the player wins, computed as $(n_items - n_guesses)$, where n_items is the initial number of items in the board. If the difference is negative, a score of 0 is assigned.

From the main page of the application, any anonymous user can select a difficulty level, start the game, play the full match, and receive a score (that will not be stored nor remembered).

From the main page of the application, logged-in users can select a difficulty level, start the game, play the full match, and the result of their match is remembered and added to the user's personal history.

A logged-in user may access a page with the history of his/her played matches, with a list of completed matches, showing for each of them: date, difficulty, secret item, score. Also the user's total score (the sum of all scores obtained by that logged-in user) must be displayed on the same page.

The organization of these specifications in different screens (and possibly on different routes) is left to the student.

Project requirements

- 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 project must be implemented as a React application that interacts with an HTTP API implemented in Node+Express. The database must be stored in a SQLite file.
- The communication between client and server must follow the "two servers" pattern, by properly configuring CORS, and React must run in "development" mode with Strict Mode activated.
- The evaluation of the project will be carried out by navigating the application. Neither the behavior of the "refresh" button, nor the manual entering of a URL (except /) will be tested, and their behavior is undefined. Also, the application should never "reload" itself as a consequence of normal user operations.
- The root directory of the project must contain a README.md file, and have two subdirectories (client and server). The project must be started by running the two 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. You may assume that nodemon is globally installed.
- The whole project must be submitted on GitHub, on the same repository created by GitHub Classroom.
- The project **must not include** the `node_modules` directories. They will be re-created by running the "npm install" command, right after "git clone".
- 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 file, so that the npm install command might install them.
- User authentication (login and logout) and API access must be implemented with passport.js and session cookies. The credentials should be stored in encrypted and salted form. The user registration procedure is not requested.

Commented [19]: I have a question with the score functioning. If I am playing in the hard difficulty, the only way to win with a score of 0 is by having 36 guesses that represent 36 values on different properties. The unique item has a probability of almost 100% to appear before that. This means that this game mode is always going to be won by using a lot of guesses that gives me the unique item, and without effort gives me more points than the others. Could it be better that the reduction of points is done in other way? Like a percentage way.

Commented [20]: I agree that the scoring is awkward. Let's use it anyway, for its simplicity

Commented [21]: Perfect, thank you

Commented [22]: Should we handle the event in which a logged user starts a game but doesn't finish it (by tracking it with a 0 score) or can we assume that any started game will be completed?

Commented [23]: Interesting question. There is no specific requirement on this point, you are free to make your choice.

I would say that an incomplete match will leave no trace in the history (not even 0), or leaves a special value (different from 0, to mark its semantic difference).

Commented [24]: In fact, the text mentions "completed" matches to go into the history

Commented [25]: Does the total score mean all scores of list accumulated

Commented [26]: It means the sum of the scores of that player

Database requirements

- The project database must be implemented by the student, and must be pre-loaded with 36 items, at least 4 properties, and at least 3 registered users, of which two already played some games, and one never played a game.

Commented [27]: Should we also provide a way for new users to register?

Commented [28]: absolutely not, it's not a requirement (you can put the users directly in the database)

Important note

- As mentioned before, the identity of the secret item must **never** be sent to the browser (except at the end of the game), therefore the APIs must be designed taking into account this requirement.

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 o the exchanged objects
 - b. A list of the database tables, with their purpose
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
3. Overall:
 - a. A screenshot of the **screen for playing the game**. The screenshot must be embedded in the README by linking two images committed in the repository.
 - b. Usernames and passwords of the users.

Submission procedure

To correctly submit the project, you must:

- **Be enrolled** in the exam call.
- Use the provided link to **join the classroom** on GitHub Classroom (i.e., correctly **associate your GitHub username with your student ID**) and to **accept the assignment**.
- **Push the project** in the **main branch** 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, and it is a git 'tag', nor 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
```

Alternatively, you may insert the tag from GitHub's web interface (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 on a clean 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 install; npm run dev)
(cd server ; npm install; nodemon index.js)
```

Ensure that all the needed packages are downloaded by the npm install 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.

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 case of import and require() statements.