

ECE 421 Project 3 Design

Mackenzie Malainey Lora Ma
mmalaine@ualberta.ca lora@ualberta.ca

Benjamin Kong
bkong@ualberta.ca

April 2022

Contents

1	Server Backend	1
1.1	REST Endpoints	1
1.1.1	User Authentication	1
1.1.2	Match Records	3
1.2	Backend Stack	5
1.3	Admin CLI	6
2	Web Client	6
3	Local Setup	6
3.1	Notice	6
3.2	Default Config	6
3.3	Instructions	6
4	Key Design Considerations	6
4.1	Computerized Board	6
4.2	Computerized Opponents	7
4.3	Interface Components	7
4.4	Exception Handling	7
4.5	Development Tools	7
4.6	UI Design Patterns	7
5	Known Issues	7
5.1	User Authentication	7
5.2	General UI	8
5.3	Leaderboards	8
5.4	Match Record Database	8

1 Server Backend

1.1 REST Endpoints

1.1.1 User Authentication

POST /api/v1/user/login/

Description

Logs in a user.

Request Body Format

Form Data

Request Body Data

user_id = USER_PROVIDED_USERNAME

password = USER_PROVIDED_RAW_PASSWORD

Response Cookies

ADD user_auth_token

Response Status

200 - If successful

401 - If user_id and password do match an existing user

404 - If request body is malformed

Known Issues

Attempting to log in to another account when already logged will automatically log out the other user from the server's perspective. However this might pose a security concern (especially if user specific data gets cached in the future)

Currently accepts and processes requests without any method of form encryption (dangerously insecure)

POST /api/v1/user/logout/

Description

Logs out the currently logged in user.

Request Cookies

CONTAINS user_auth_token

Response Cookies

CLEAR user_auth_token

Response Status

200 - If successful

404 - If header doesn't contain cookie

Known Issues

Need to verify this removes the cookie in browser

GET /api/v1/user/verify/

Description

Verifies that the client's auth cookie matches a known user

Request Cookies

CONTAINS user_auth_token

Response Status

200 - If successful

404 - If header doesn't contain cookie

Response Body Type

JSON

Response Body

Username of authenticated user

Known Issues

Need to verify this removes the cookie in browser

POST /api/v1/user/register/

Description

Registers a new user.

Request Body Format

Form Data

Request Body Data

user_id = USER_PROVIDED_USERNAME

password = USER_PROVIDED_RAW_PASSWORD

Response Cookies

ADD user_auth_token

Response Status

200 - If user_id and password match an existing user

401 - If user_id and password do not match an existing user

404 - If request body is malformed

Response Cookies

ADD user_auth_token

Response Status

200 - If user_id and password match an existing user

404 - If header doesn't contain cookie

Known Issues

Attempting to log in to another account when already logged will automatically log out the other user from the server's perspective. However this might pose a security concern (especially if user specific data gets cached in the future)

Currently accepts and processes requests without any method of form encryption (dangerously insecure)

Assumes client enforces proper password requirements

1.1.2 Match Records

POST /api/v1/user/records/add

Description

Registers match data for the current user

Request Body Format

JSON

Request Body Data

```
"start_time": START_TIME_IN_SECONDS_FROM_EPOCH_UTC_TIME,  
"game_id": {"Connect4", "OttoToot"},  
"cpu_level": {"Easy", "Medium", "Hard"},  
"duration": DURATION_IN_SECONDS,  
"result": {"Win", "Loss", "Tie"}
```

Request Cookies

CONTAINS user_auth_token

Response Status

200 - If successful

401 - If user_auth_token does match an existing user

404 - If request body is malformed

Known Issues

Does not verify "start_time". Probably best to remove this field and use server time to log time upon recording.

GET /api/v1/user/records

Description

Retrieves all the match data from the current user sorted by most recent matches first

Request Cookies

CONTAINS user_auth_token

Optional Request Query Parameters

limit (default = 10)

Number of records to return at once

offset (default = 0)

Number of records to skip (for pagination)

before

Returns matches that happened before (UTC timestamp in seconds)

after

Returns matches that happened after (UTC timestamp in seconds)

sort_by (default = starttime)

Value to sort by, can be either duration or starttime

asc
Sort direction, defaults to false unless using **sort_by=duration**

filter
Only returns elements that match the filter specification (see examples for more info)

Response Status

200 - If successful
404 - If header doesn't contain cookie

Response Body Format

JSON

Response Body

List of match records

Known Issues

Does not support any form of filtering or sorting other than listed

Handles case where user doesn't exist by returning empty list instead of an error status

Does not apply a maximum or minimum on limit values

Should include a count of how many records there are in total

GET /api/v1/games/records

Description

Retrieves all the match data from the current user sorted by most recent matches first

Request Cookies

CONTAINS **user_auth_token**

Optional Request Query Parameters

limit (default = 10)
Number of records to return at once

offset (default = 0)
Number of records to skip (for pagination)

before
Returns matches that happened before (UTC timestamp in seconds)

after
Returns matches that happened after (UTC timestamp in seconds)

sort_by (default = **starttime**)
Value to sort by, can be either **duration** or **starttime**

asc
Sort direction, defaults to false unless using **sort_by=duration**

filter

Only returns elements that match the filter specification (see examples for more info)

Response Status

200 - If successful

Response Body Format

JSON

Response Body

List of match records

Known Issues

Does not have max and min values for `limit`

Should include a count of how many records there are in total

1.2 Backend Stack

The backend is implemented using `rocket(v0.5.0)` for the backend server framework. Through `rocket`'s database connection pool library we used `diesel` as the backend database library which interfaces with a `sqlite3` database.

The original design was to use `rocket(v0.4.4)` with a `mongodb` database through `rocket`'s database connection pool library. This would allow us to carry over the prior project's database with minimal issue. However we found that some of the dependencies had been removed from `crates.io` and therefore were not able to use `mongodb` with `rocket(v0.4.4)`. Sadly `rocket(v0.5.0)` does not support `mongodb` and we decided it was best to not homebrew a solution together. That is what led us to using `diesel` with a `sqlite3` database. We felt `diesel` was a better option than `rusqlite` with its CLI app to be able to create and run database migrations, embed migrations into the app so that the database could be built on first run as well as the compile time query checking saving a lot of potential headaches during development and for future development and saved on boilerplate code.

1.3 Admin CLI

A local database can be investigated and altered directly using `prj3_cli`. To use, run the CLI and when prompted specify a path for the database you wish to alter. If no database exists at the given path one will be created. Then use the next menu to perform various actions on the database.

2 Web Client

3 Local Setup

3.1 Notice

The tutorial assumes you have the Rust toolchain installed on your system. Also, the install SQLite3 step is not necessary since `sqlite3` is compiled using the library bundled with the `libsqlite3-sys` package, however this does increase the download size for the dependency but for environment consistency is recommended. Database initialization can be completed by running the CLI or running the server with the features mentioned in the tutorial.

3.2 Default Config

The default database path is `$PROJ_ROOT$/localdev.db`

The default port that the local web server serves on is `8080`

The default port that the backend API server serves on is `8000`

IMPORTANT: a proxy is set up for the API calls (see [Trunk.toml](#)) for the local web server that expects the backend API server to be running on the same machine on the port mentioned above. If you wish to change the port or host the backend API server on a different device you will need to update the proxy address in the mentioned config file.

3.3 Instructions

See the [readme](#) for information on setting up the environment and running the server backend and web client.

4 Key Design Considerations

4.1 Computerized Board

Using a computerized board provides several key advantages over a physical board. It allows a player to play against a computerized opponent meaning a player can play and have fun by themselves. It also allows the tracking of games and statistics, allowing players to compare themselves to other players. Using a computerized board also removes the need to manually check if a player has

won or lost. A computerized board also prevents cheating.

In our program, the board state is represented as a vector with length `rows * columns`. We then provide methods that abstract access to the board state as if it were a matrix: for example, to access the slot at row 3 and column 5, we can calculate the index in the vector as `3 * columns + 5`.

4.2 Computerized Opponents

A computerized opponent is essentially code that tries to make smart decisions in order to win (i.e. beat the other player). This is done by attempting to determine the best move from the set of possible moves. This is done by giving each possible move a score by using an algorithm.

4.3 Interface Components

4.4 Exception Handling

In a GUI app exception handling must be taken with care. In a CLI app or script it might be acceptable to simply print out an error message and move on, or just panic on those rare slightly inconvenient to safely handle errors. However, for a GUI app this is much different because we always want to ensure the app remains responsive and that the user has enough feedback from our app to determine what the next rational step should be. Since panicking in a GUI app might cause the app to freeze (especially a web app) this should be avoided and any errors that cannot be fixed (such as failed logout attempts) should be reported to the user, so they are aware that something went wrong and can safely determine what the next step should be.

4.5 Development Tools

Development tools are important to have to be able to debug and verify a system outside of the server and client. We developed an Admin CLI tool to allow for us to create test data or find data in the database to verify the functionality of the server and web app. The CLI does NOT have support for directly testing the server APIs directly, instead other tools such as [Postman](#) were used to verify the API endpoints against test data.

4.6 UI Design Patterns

5 Known Issues

5.1 User Authentication

UI: MISSING No input validation is performed on the login/register forms. Empty forms can be submitted. Has not been found to cause any crashes, just possible security vulnerability.

AUTH: MISSING Username matching is case sensitive.

AUTH: MISSING No validation for usernames or passwords implemented on server.

AUTH: MISSING No validation made to verify when logging in the client doesn't already represent a valid user on server. However this has no impact right now, it is a potential future security vulnerability.

5.2 General UI

UI: BUG General UI issues such as unaligned text, dropdown menus not closing, etc

UI: MISSING No animations for game

5.3 Leaderboards

UI: MISSING No way to filter CPU level or board size (filtering CPU level is implemented server-side though)

5.4 Match Record Database

DATABASE: MISSING Way to classify board sizes for a match

DATABASE: MISSING Handling when a user account is deleted