

Assignment 3

User Stories and Project Design

CPSC 310 - Lab B - Group 2

Benjamin Ward

Cooper Webb

Andrew Park

Farbod Nematifar

Jeremy Gan

Table of Contents

1. USER STORIES	3
2. USER INTERFACE (UI) AND API ENDPOINTS.....	14
3. LLM INTEGRATION:	21
4. DATABASE.....	24
5. TESTING.....	27

1. User Stories

Login/Signup + User Agreement (Epic 1)

Sign up

RGB

- As a user I want to be able to sign up for an account so that I can access the website.

Limitations and Clarifications

- User's must not already have an account to sign up for an account.
- If a user does not enter both their username and password, an error will be displayed.

Definitions of Done

- After users create an account, their info is successfully stored in the database.
- Users are able to login after they signup and access the features that are specific to their account.
- Passwords are securely hashed and stored in the database.

Engineering Tasks

1. Create a back-end table for users that will include (user ID, username, password).
2. Implement a signup page with two text boxes (username and password) and "Sign Up" button.
3. Use node.js and SQLite to connect frontend to database.
4. Write unit tests to test adding in new users.

Estimation of Time: 3.5 hours

1. Create user table: 0.5 hours
2. Implement signup page: 2 hours
3. Write unit testing: 1 hour

Log-In

RGB

- As a user I want to be able to log in to my account and see my personalized data/progress.

Limitations and Clarifications

- Must already be signed up for an account (account must already exist).
- Must enter correct login information.
- If a user does not enter their username and password, an error will be displayed.

Definition of Done

- Users can login with username and password.
- Users go to the user profile page after successful login.
- Users receive an error message if information is incorrect.
- Passwords are securely hashed and matched against stored passwords.

Engineering Tasks

1. Create a simple login-page using react, to allow users to type in necessary information (username, password).
2. Connect the front-end login form with the back-end JavaScript and SQLite to verify user credentials and provide access to the user profile page.
3. Write unit tests for the login functionality, conduct integration testing to ensure back-end front-end connection, and perform end to end testing of login process to make sure it works.

Estimation of Time: 9 hours

- Build the front-end react page: 2 hours
- Ensure front-end back end correctly connected and user able to login: 4 hours
- Test front-end back-end functionalities: 3 hours

Data & Privacy (User Agreement)

RGB

- As a user, I want to ensure that all data I provide is securely stored, so that my data is adhering to PIPEDA and ACM code of ethics and I am protected.

Limitations/Clarifications

- Due to the limitations of our experience and knowledge, the team will try to securely store by ensuring that all steps (within our means and knowledge) are taken.
- This means ensuring that access to our database is limited to only those that are authorized to do so but does not include things like encryption protocols (except for simple md5 hashing of extremely sensitive information such as passwords).

Definition of Done

- Triple test that common attacks such as SQL-Inject doesn't work on our site nor does brute-forcing and also checking that all passwords are actually encrypted on the backend (in the database).

Engineering Tasks

1. Research on the latest vulnerabilities of the database we are using and ensure that we apply any hotfix required to patch them out. (~4 hours).
2. Modify our database to encrypt all passwords (which will include working with our registration module + database tweaks) so that even authorized user that can view all data are unable to view full passwords, and on that note, even reset password will require users to create a new password instead of telling them what their password is. (~ 4 hours)
3. Review PIPEDA and ACM code of ethics and ensure that our platform conforms to its guidelines. (~2 hours)
4. Checkbox for user agreement. The users agree to have their username, password and performance data stored on our database (Add reference to Code of ethics clause).

Estimation of Time: 11 hours

1. Research on the latest vulnerabilities (4 hours)
2. Encrypt all passwords (4 hours)
3. Review PIPEDA and ACM code of ethics (2 hours)
4. Checkbox for user agreement (1 hour)

User Profile (Epic 2)

User Page

RBG

- As a user, I want to be able to view personal information, specifically my username, total time spent on questions, join date, total questions passed, percentage of questions completed, and the total number of tests failing so I can keep track of progress and see how I am doing.

Limitations and Clarifications

- Information on the page should contain personal statistics that show a user's progression with the questions.
- User's personal information such as login username and user ID are viewable.

Definition of Done

- Users can see which questions they have attempted, total time spent, level of difficulty achieved and other pertinent information that is useful.
- Statistics are viewable in a format that is both pleasing to look at (i.e. pie chart showing number of questions completed for each difficulty level) and informative (ie. total hours spent displayed below pie chart).

Engineering Tasks

1. Create front-end display to show progress information from back-end table
2. Create a new database to contain results for each user including information such as question ID, title, difficulty, tags, student ID, and user name
3. Implement back-end logic using TS and SQLite to track results of exercises attempted

Estimation of Time: 9 hours

- Build front-end: 5 hours
- Create database containing information for each user: 2 hours
- Implement back-end logic: 2 hours

Questions (Epic 3)

Question List Page

RGB

- As a student I want to be able to pick from a list of questions so that I can focus on certain topics.

Limitations and Clarifications

- Students should be able to see which questions they have already completed.
- Questions should be filterable by difficulty or topic or tags.
- Questions will have a green background marking complete, or a red background if incomplete or not started.

Definition of Done

- Students can see available questions organized by category on the question menu.
- Students can click on a question to attempt it.
- Students can see which questions they have/have not completed.

Engineering Tasks

1. Create a new database table to store attributes for each question (exercise ID, title, difficulty, tags, and completion status, and studentID).
2. Develop a front-end display for the title of each question, along with its status (done or not done) using react (scrollable selection pane).
3. Implement back-end logic using TS and SQLite to track the problems that are done (all tests pass) or incomplete/not attempted.
4. Connect the front-end to the back-end. Write tests for individual components so that questions load and are present, demonstrate status, and when clicked, students are taken to the question page.

Estimation of Time: 15 hours

- UI development for creating a scrollable pane that contains all of the questions, colored background will indicate completeness: 2 hours
- Write the individual questions that users will answer (Ex. 10 questions): 5 hours
- Ensure connection between front-end back-end is correct and works as expected: 3 hours

Question Attempt Page

RGB

- As a student I want to be able to complete a programming comprehension exercise by seeing the given question on the screen, alongside a textbox where I can input my answer, explain my interpretation of the function in plain English and press “Submit” to get a response and see whether I am correct.

Limitations and Clarifications

- Students should be able to view the question (function) and submit their response.
- Return an ERROR message when a student attempts to copy the given code snippet into the textbox.

Definition of Done

- Students can view the question.
- Students can enter their interpretation.
- Students can click a “Submit” button to send their answer.

Engineering Tasks

- The LLM model will be given a **system message** that will clarify that users are not allowed to input JavaScript code. This will prevent users from copying and pasting code.
- Generate the UI to display the function associated with the question. This function will be pulled from the back-end.
- Generate the UI for user submission text box and submission button.
- Associate a back-end endpoint for the specific question #id. This will be triggered by the “Submit” button. This will pass the data from the user submission text box,
- When the student's answer is submitted it is sent to an LLM api to return a LLM generated code response for testing.

Estimation of Time: 9 hours

- Ensure LLM model checks for invalid input responses: 2 hours
- UI (Function Display, Submission Box, Submission Button): 3 hours
 - Sending user submission to OpenAI’s GPT4o API for result: 4 hours

Question Feedback

RGB

- As a student, I want to get feedback on my attempt (pass/fail), with failures being displayed as tests failed and passes being displayed as tests pass, to demonstrate whether their interpretation is functionally equivalent to the code snippet.

Limitations

- If incorrect, students must be given feedback, the number of tests passed vs. failed.

Definitions of Done

- Students can attempt a question and reliably get response/feedback for their attempt (number of tests passed/failed).
- Students will be prompted to re-submit if their response fails the tests.
- Students will be prompted to return to the question selection screen if their tests passed.

Engineering Tasks

- Write a script to parse the LLM generated code for the markdown block.
- Pass the parsed code to an isolated JavaScript virtual machine to run the test cases
- Write the test cases for each question.
- Map results for the user's submission to the database (username, time-taken, tests passed).

Estimation of Time: 17 hours

- LLM response parsing: 2 hours
- Running the parsed response in an isolated vm on test cases and returning the results: 10 hours
- Writing test cases per each question: 5 hours

Additional Features (Epic 4)

Leaderboard

RGB

As a user I want to be able to see a leaderboard to see how I rank compared to other students.

Limitations and Clarifications

- User always appears on leaderboard (if they don't rank, they go in the bottom at "0" questions)

Definition of Done

- Users are to click on leaderboard page and see their position in leaderboard
- Leaderboard includes users username ranked by questions completed

Engineering Tasks

1. Pull top 10 users from Database and pass to front-end
2. Add UI for Leaderboard page: design a page with section for leaderboard
3. Add each user "entry" to appropriate place on leaderboard

Estimation of Time: 7 hours

1. Pull users from Database (2 hours)
2. Add UI for Leaderboard page (3 hours)
3. Add the users to the correct place on the leaderboard (2 hours)

Hint

RGB

- As a user I want to be able to receive a hint when I do not pass a test for a question if I am stuck so that I can be guided in the right direction.

Limitations / Clarifications

- Each hint can only be given after at least 1 attempt at the question.
- Each failed test only has 1 hint.
- Each hint is pre-established and not based on the user's input.
- Users must push hint button to see hint.

Definition of Done

- When a student enters their response and proceeds to fail at least one test, there will be a hint button that appears that students can click on for each failed test that shows one hint as to why the student may be failing that test.
- The hint will suggest why that test may be failing. Hint's are static and mapped to each specific button.

Engineering Tasks

1. Write a hint for each test that may help a user who is failing that specific test.
2. Add a button beside the failed test that is called hint.
3. Program the button to reveal the specific hint corresponding to that failed test.
4. Add the static hint button for each failed test and reveal the specified hint for that test.
5. Do this for all tests, for all questions.

Estimation of Time: 6 hours

- Each of the test cases will have their own specific hint which will be return from back-end: 4 hours
- Button feature added to front-end: 2 hours

Potential Additional Extra Features (Epic 6)

Progress Tracking

RGB

- As a student, I want to be able to get my score average grouped/categorized by type of coding question, so that I can get more practice on specific type of "code structures" that I need improvement on.

Limitations

- Categories of "code" are pre-determined and students need to recognize these categories. For example: sorting, linked list insertion, etc.

Definition of Done

- Have an overall view of categories of code attempted and average score of each category.

Engineering Tasks

1. Ensure that each code in the "databank" belongs to a category.
2. To make it clear to students/users, each category shown alongside each question [Can make it a click-to-reveal to prevent spoiler/hint, or reveal after submission, to-be-decided]
3. Backend: `CategoryX_Questions ++; if(right) categoryX_Right ++; Display: categoryX_Right/CategoryX_Questions, per category.`

Estimation of Effort: 25 hours

- Ensuring each code in databank belongs to a category (~1 hour if already pre-assigned, else depending on size of databank/questionset, could take up to hours)
- Frontend: Make it clear to student/users/each category shown, fetch it from backend (~12 hours of front/back-end coding)
- Backend: In conjunction with previous, query to update backend in conjunction to frontend(ajax) ~12 hours of front/back-end coding)

Password Recovery

RGB

- As a student, I want to be able to recover my password, so that I can regain access to my account if I've somehow forgotten my password.

Limitations

- Must already have a registered account on the platform and also have access to the email the account was used to register with.

Definitions of Done

- Able to recover password via email or at least reset the password.

Task & Estimation of Effort (25 hours)

1. Decide on an auth system such as a Firebase or OAuth 2.0 (~1 hour)
2. Study the API and protocols for integrating chosen auth system (~12 hours)
3. Integrate into our base code and test for usability (~12 hours)



Figure 1: Users will login by typing in their username and password into the text boxes and selecting “Sign In”. If users do not have an account, they can select the “Join now” button to be redirected to the “sign up” route.

HTTP Method

Parameters

Returns

Sign up Page (signup endpoint)

● Placeholder Title

Sign in

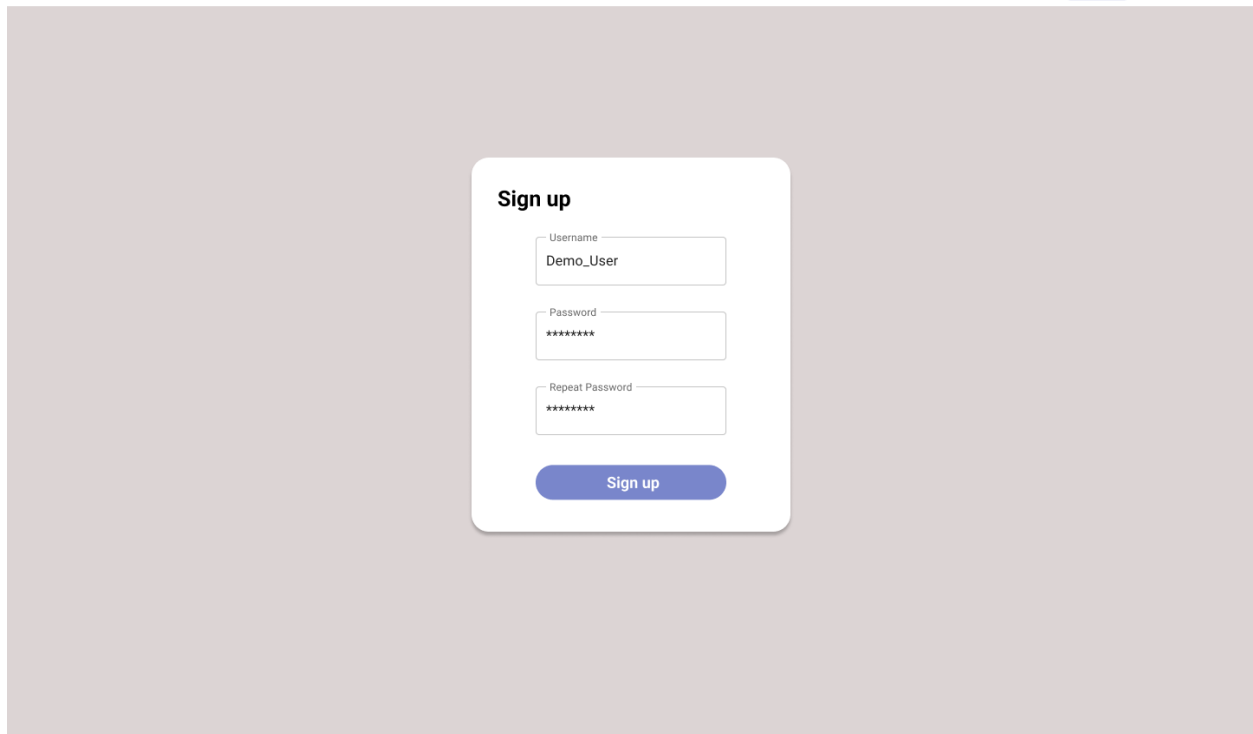
A screenshot of a web application's sign-up page. The page has a light gray background. In the center, there is a white rounded rectangle containing the sign-up form. The form is titled "Sign up" in bold black text. It contains three input fields: "Username" with the text "Demo_User", "Password" with "*****", and "Repeat Password" with "*****". Below these fields is a blue button with the text "Sign up". In the top right corner of the page, there is a blue button with the text "Sign in".

Figure 2: Users will register for an account by typing in their username and password into the text boxes and selecting “Sign Up”.

HTTP Method

Method	URL
POST	api/signup/

Parameters

Type	Params	Param Type	Description
BODY	username	string	Username entered in login screen
BODY	password	string	Password entered in login screen

Returns

Params	Param Type	Description
session_id	string	Session_id for the frontend state

Questions Page (questions endpoint):

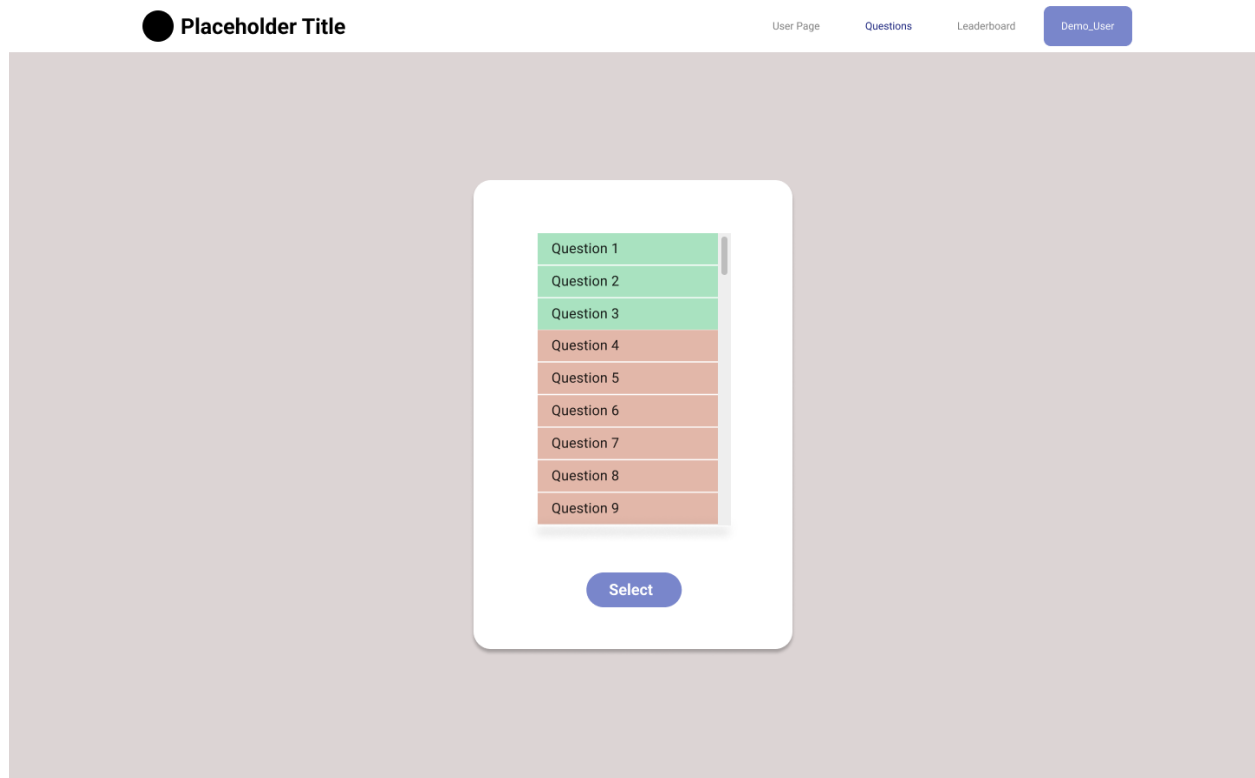


Figure 3: Users can select which questions they would like to select on the question list (green marks complete; red marks incomplete). Once they highlight a question from the list, the user can click “Submit”, to enter the associated questions screen (Figure 4).

HTTP Method

Method	URL
GET	api/questions

Returns

Params	Param Type	Description
question_list	array	Array of each question and their id

Question Page (question submit endpoint):

Placeholder Title

User PageQuestionsLeaderboardDemo User

Question #11

```
1 function foo(a) {
2
3   let b = 0;
4
5   for (let i = 0; i < a.length; i++) {
6     b += a[i]
7   }
8
9   return b;
10 }
11
12
```

Interpretation

This function takes in one parameter, an array. It iterates through the array and calculates its sum. The sum is the final return value.

Submit

Figure 4: Here the associated function for question #11 is displayed on the screen. The user can enter their interpretation of the function within the text field at the bottom and press the “Submit” button. An associated result box as shown in Figure 5 will be displayed when the backend returns a result.

HTTP Method

Method	URL
POST	api/submit/:id

Parameters

Type	Params	Param Type	Description
PATH	id	string	id mapped to the specific question
BODY	Answer	string	Users answer to the question
BODY	session_id	string	Session_id for username
BODY	time_taken	number	Time taken before submission

Returns

Params	Param Type	Description
Test_Cases_Passed	number	Number of test cases passed
Test_Cases_Failed	number	Number of test cases failed

Additional Notes

Based on the http response code, alternate results can be displayed. Below are examples of how different kinds of response codes alter the question result field.

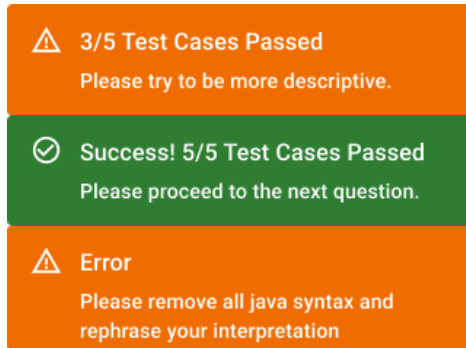


Figure 5: Associated result boxes that will be displayed under the “Interpretation” text field on the question attempt page when a result is returned.

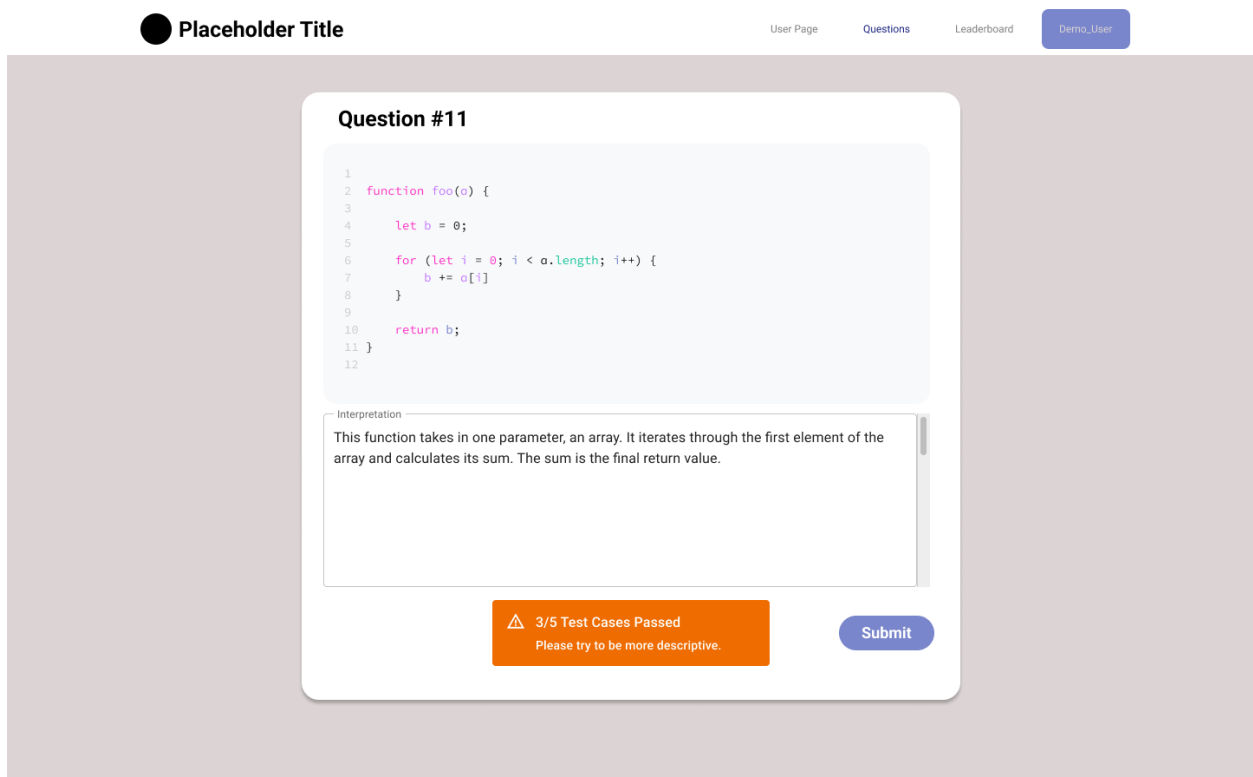


Figure 6: After the user submits their interpretation a result box will appear. As noted above (Figure 5) different result boxes can be displayed based on backend results or errors.

Leaderboard Page (leaderboard endpoint):

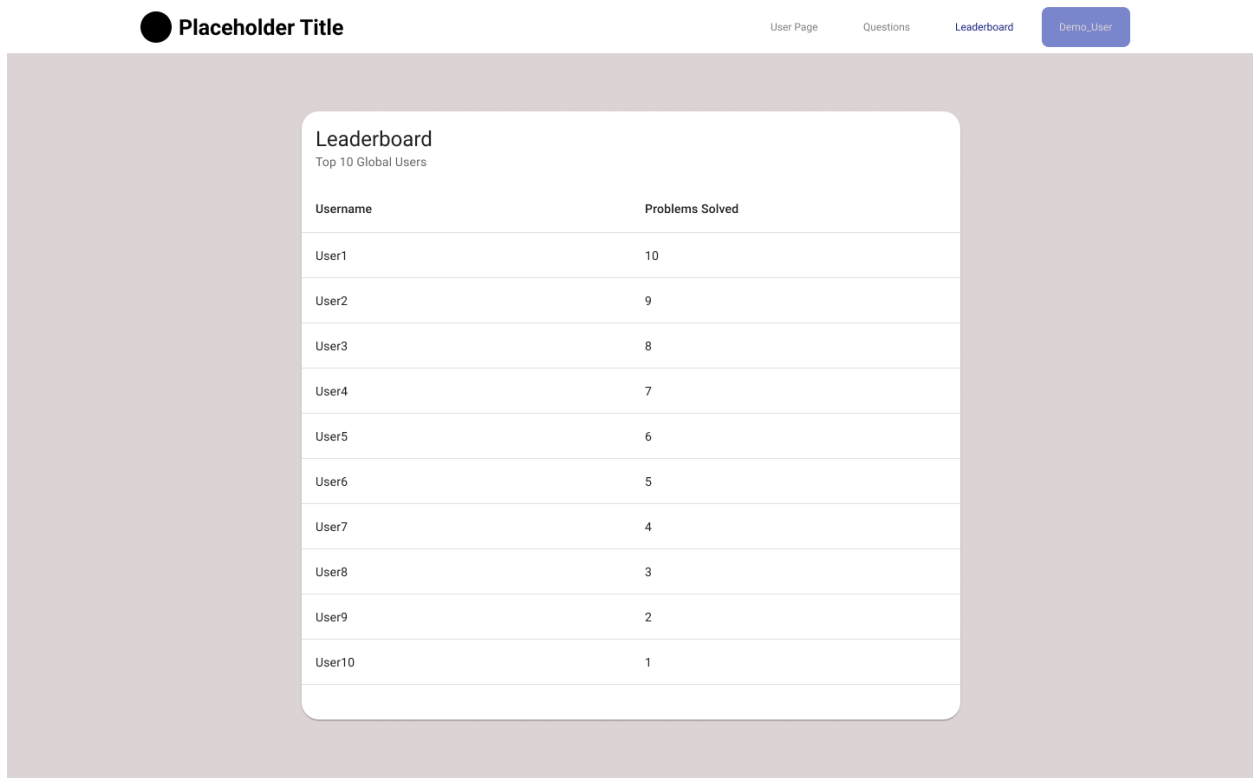


Figure 7: When an authenticated user clicks on the leaderboard page. The top 10 global users will be dynamically rendered. The user will be available to see the top 10 user's usernames and number of problems solved.

HTTP Method

Method	URL
GET	api/leaderboard/

Returns

Params	Type	Description
top10_users	array	Top 10 users and their number of problems solved

User Page (user_statistics endpoint):

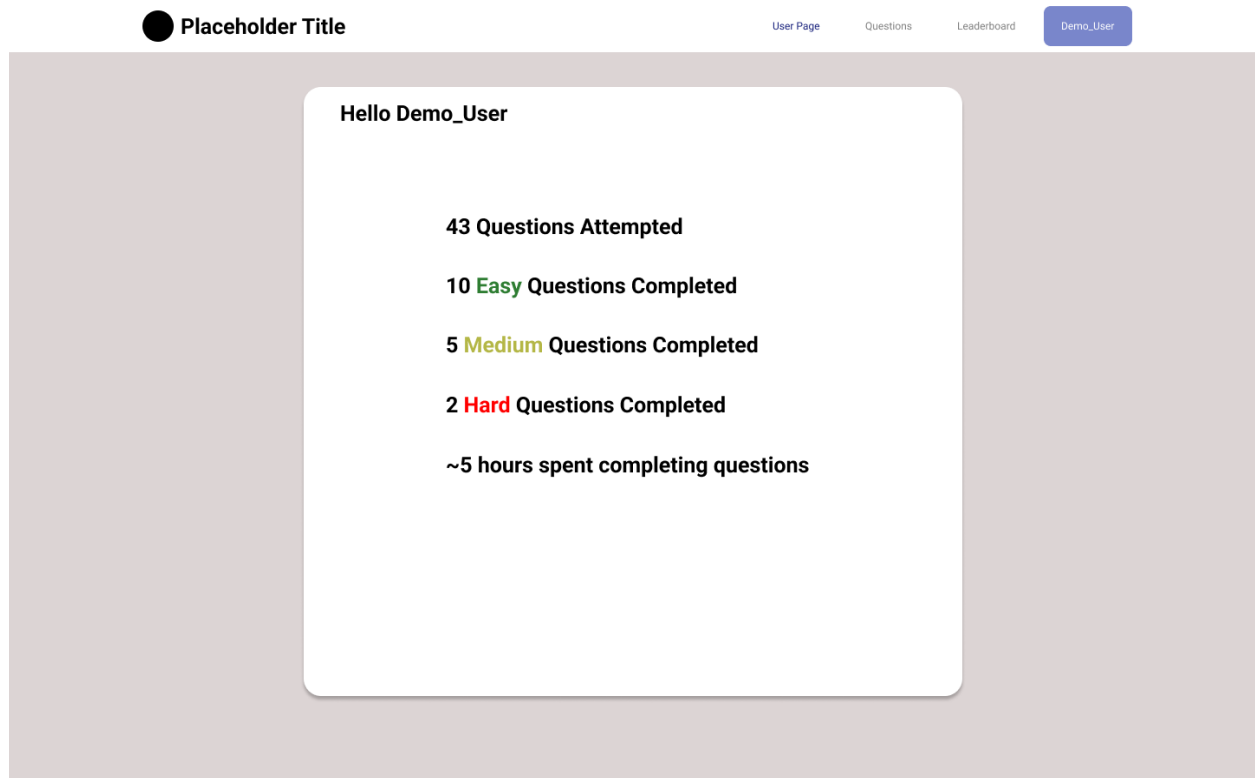


Figure 8: On the user page, the authenticated user will be able to view statistics for their associated account. The user will be displayed the total number of questions attempted, # of easy/medium/hard questions completed and their total amount of time spent.

HTTP Method

Method	URL
POST	api/user_statistics

Parameters

Type	Params	Param Type	Description
BODY	session_id	string	Session_id to determine user

Returns

Params	Param Type	Description
Questions attempted	number	Number of questions completed
Easy questions completed	number	Number of easy questions completed
Medium questions completed	number	Number of medium questions completed
Hard questions completed	number	Number of hard questions completed
Total time spent	number	Number of questions completed for user

3. LLM integration:

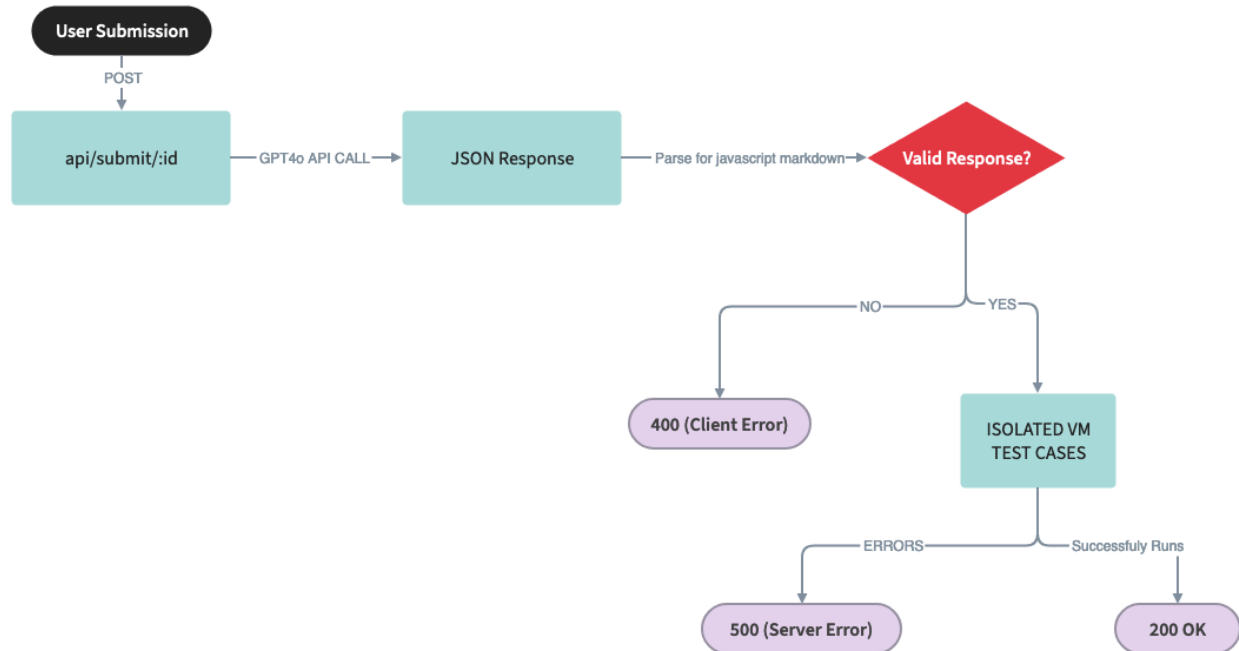


Figure 9: LLM user submission pipeline. Once a client submits a message on the question page (Figure 4), a post message is sent from the frontend to the backends `api/submit/:id` endpoint. Once the backend receives the user message it is sent in the body of a [GPT4o api call](#). Once the API returns a JSON response, it is parsed for the JavaScript markdown block or the substring “ERROR”. If the JavaScript markdown block is successfully detected, the contained JavaScript code will be injected into a virtual machine for testing, utilizing the [virtual machine library](#).

GPT4o API calls:

System message specification:

The system message applied to the api calls prompt the model to detect copy and pasted code from the user. If it detects any JavaScript syntax it will return “ERROR”, which will trigger the frontend to prompt the user to remove all JavaScript syntax.

Example Valid API Message:

```
{
  "model": "gpt-4o",
  "messages": [{ "role": "system", "content": "If the users message contains any javascript code syntax return \"ERROR\". Otherwise only return a single javascript code block of their interpretation with the function name foo." }, { "role": "user", "content": "The function takes one parameter a and returns the parameter plus 1" } ],
  "temperature": 0.7
}
```

Example Valid Message Response:

```
{
  "model": "gpt-4o-2024-05-13",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "```javascript\nfunction foo(a) {\n  return a + 1;\n}\n```"
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
}
```

Example Invalid Message:

```
{
  "model": "gpt-4o",
  "messages": [{ "role": "system", "content": "If the users message
contains any javascript code syntax return \"ERROR\". Otherwise only
return a single javascript code block of their interpretation with
the function name foo."}, { "role": "user", "content": " add(a)
{return a + 1}" }],
  "temperature": 0.7
}
```

Example Invalid Message Response:

```
{
  "model": "gpt-4o-2024-05-13",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": "ERROR"
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
}
```

JSON processing from API call:

Utilizing JavaScript on the backend we will extract the first JavaScript code block from the response by identifying the starting string (``javascript) and the ending string (``) from the markdown block.

Preprocessed response:

```
````javascript\nfunction foo(a) {\n  return a + 1;\n}\n````
```

Backend processed response:

```
increment(a) {return a + 1;}
```

## 4. Database

For the backend, we've decided use SQLite to store our data. SQLite is a **file-based database** and we've chosen this approach due to the simple nature of the application and the small number of concurrent users that will be using the application.

In terms of the data that will be included, we're currently expecting at least the following tables in the database to be populated.

Table Name	Description
Users	All information about user, eg: UserID, UserName, Password <i>[Note: We're accounting specifically for only 1 type of users: Students]</i>
Questions	The "databank" of questions that we have curated for the application.
Attempts	A collection of all attempts made by students which identifies the unique student, which question, and which attempt including even the time spent on the question and scores

Our team has also chosen to work with the Express framework so this section will cover how we'll utilize Express to make API calls to our SQLite database in order to access, retrieve and edit our data.

In order to do this, we'll first have to make a connection to the database (*assuming that we've properly created and set up an SQLite database along with proper data in our container*). This could be done with something like:

```
// Get the location of database.sqlite file
const dbPath = path.resolve(__dirname, 'db/ourtestdatabase.sqlite')
// Create connection to SQLite database
const test = require('test')({
 client: 'sqlite3',
 connection: {
 filename: dbPath,
 }
})
```

In terms of API calls for our database, we will further utilize Express to do this. We begin by creating a controller.



```

//Import our database
const test = require('./../db')

// Retrieve all users
exports.usersAll = async (req, res) => {
 // Get all users from database
 test
 .select('*') // select all records
 .from('Users') // from 'Users' table
 .then(userData => {
 // Send all users extracted from database in response
 res.json(userData)
 })
 .catch(err => {
 // Send a error message in response [Also a way of testing]
 res.json({ message: `There was an error retrieving users:
${err}` })
 })
}

```

Essentially by utilizing SQLite's native query system we can craft our own API calls (controllers) required to interact with our database depending on what we require for our application on the front end. We then need to create a router that takes in the API endpoint and second, the controller which we've created above.

```

// Create router
const router = express.Router()

// Add route for GET request to retrieve all users
// In server.js, users route is specified as '/users'
// this means that '/all' translates to '/users/all'
router.get('/all', usersRoutes.usersAll)

```

## Database Architecture Template

### Tables

- Users: stores user specific information (username, password)
- Question: stores information on each attempted question (**Question ID**, **User ID**, question, status (pass/fail), total time spent, total errors)
- Attempt: an attempt at a question (**Question ID**, **User ID**, **Attempt ID** time spent, errors, hint, date/time, user answer, LLM generated code, results of test cases, pass/fail, [reason], score

## 5. Testing

The testing suite we will use is Mocha/Chai.

### Login / Signup Testing

When users attempt to login/signup, we need to ensure that their usernames aren't already taken, and that they don't include any forbidden characters. If users attempt this, we have to test that they receive the correct error response back.

Here's a sample test for this:

```
describe('User Creation', () => {
 describe('POST /api/users', () => {
 it('illegal character', (done) => {
 let user = {
 username: 'invalid*user',
 password: 'password123'
 };
 chai.request(server)
 .post('/api/users')
 .send(user)
 .end((err, res) => {
 res.should.have.status(400);
 res.body.should.be.a('object');
 res.body.should.have.property('error');
 res.body.error.should.be.eql('Username contains
invalid characters. ');
 done();
 });
 });
 });
});
```

## **Leaderboard Testing**

We test to ensure that, when a user clicks onto the leaderboard, it's updated with the current high score. Tests will include updating the scores to other users, as well as updating the current users score. Tests will also include checking if a series of questions was done, that another students leaderboard will coincidingly update.

## **API Endpoint Testing**

For the backend we will be testing that all our connections work successfully. For example:

```
describe('Users API', () => {
 describe('GET /api/users', () => {
 it('returns all users', (done) => {
 chai.request(server)
 .get('/api/users')
 .end((err, res) => {
 res.should.have.status(200);
 res.body.should.be.a('array');
 res.body.length.should.be.eql(3);
 done();
 });
 });
 });
});
```

## LLM Response Testing (Questions)

We will do Unit Tests where we pass in “correct” and “incorrect” attempts and test this against the responses that should be given.

For example, if this is our code:

```
function foo(a, b) {
 return a + b;
}
```

We will run several responses which we would accept as correct, and several that would be rejected as wrong, and test the code the LLM returns against test cases.

For example, if we input: ***“In this code block we take two variables, add them together, and return the result,”*** we would expect that the code returned by our LLM successfully passes all tests.

However, if we input: ***“In this code block we are returning the letters a and b.”*** We would expect or all of the tests to fail.

However, if the input was ***“In this code block we are returning the sum of two zeros”*** one test may pass, but the remainder fail.

We also need to do tests to ensure that users aren’t copying and pasting their answers, or just telling the LLM to give them the answer.