Online Judge

About the project

An Online Judge platform provides a dynamic and interactive environment for users to enhance their programming skills by solving various coding challenges. Upon landing on the site, users can either log in or sign up to create an account. Once logged in, they are greeted with a comprehensive list of problems, each tagged with its difficulty level and relevant categories.

Users can select any problem to view its detailed description and test their solutions. They can run their code with custom inputs to see how it performs before submitting it for evaluation. The platform rigorously tests each submission against a set of hidden test cases to ensure the solution's correctness and efficiency.

Additionally, the problem page includes a submissions tab, showcasing the last 10 submissions made by all users, along with their respective run times. This feature allows users to gauge the performance of their solutions compared to others.

A leaderboard highlights the top 5 users who have solved the most problems, fostering a competitive spirit among participants. Each user also has a profile page where they can view their personal details and a list of problem titles they have successfully solved.

This Online Judge platform is designed not only to challenge users but also to create a supportive community of coding enthusiasts, helping each other improve and excel in their programming journey.

About frontend

The frontend of the Online Judge website consists of several key pages, each serving a specific purpose to ensure a seamless and engaging user experience:

1. **Landing Page**:
    - **Purpose**: This is the first page users encounter. It provides a welcoming introduction to the platform and features two prominent buttons for logging in and signing up.
2. **Signup Page**:
    - **Purpose**: This page allows new users to create an account. It collects necessary information such as name, email, and password. After successful signup, users are redirected to the problem list page.
3. **Login Page**:
    - **Purpose**: This page enables existing users to access their accounts by entering their email and password. Successful login also redirects users to the problem list page.
4. **Problem List Page**:
    - **Purpose**: This page displays a comprehensive list of available problems, along with difficulty and category tags. It helps users browse and select problems they wish to solve. The page includes three sections: one for the problem list, another showing the last 10

submissions by all users along with their run times, and a section displaying the top 5 users who have solved the most problems. A button at the top right corner leads to the user's profile page.

5. **Problem Description Page**:
    - **Purpose**: When a user clicks on a problem from the problem list, this page opens. It provides a detailed description of the selected problem and allows users to run their code with custom input. Users can also submit their solutions to be tested against hidden test cases.
6. **Profile Page**:
    - **Purpose**: This page displays the user's personal details and a list of problem titles they have solved.

About backend

Backend: Web-APIs

1) /login

Method: POST

Request Body:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

Response Body:

Success (HTTP 200 OK):

```
{
  "message": "Login successful",
  "user_id": 1,
  "name": "John Doe"
}
```

Failure (HTTP 401 Unauthorized):

```
{
```

```
  "message": "Invalid email or password"
}
```

Functionality:

- **Validation: Check if the email and password match an existing user in the database.**
- **Session Creation: If credentials are valid, create a user session and return a success response with the user ID and name.**
- **Error Handling: If credentials are invalid, return an error message.**

2)/signup

Method: POST

Request Body:

```
{
  "name": "John Doe",
  "email": "user@example.com",
  "password": "password123"
}
```

Response Body:

Success (HTTP 201 Created):

```
{
  "message": "Signup successful",
  "user_id": 1,
  "name": "John Doe"
}
```

Failure (HTTP 400 Bad Request):

```
{
```

```
  "message": "Email already exists"
}
```

Functionality:

- **Validation: Check if the email is already registered.**
- **User Creation: If the email is not registered, create a new user in the database.**
- **Session Creation: After successful user creation, create a user session and return a success response with the user ID and name.**
- **Error Handling: If the email is already registered, return an error message.**

3)/problems

Method: GET

Request Body: None

Response Body:

Success (HTTP 200 OK):

```
{
 "problems": [
   {
     "id": 1,
     "title": "Two Sum",
     "difficulty": "Easy",
     "category": "Array"
   },
   {
     "id": 2,
     "title": "Longest Substring Without Repeating Characters",
```

```
        "difficulty": "Medium",
        "category": "String"
      }
    ]
  }
```

Failure (HTTP 500 Internal Server Error):

```
{
    "message": "An error occurred while fetching problems"
}
```

Functionality:

- **Fetch Problems: Retrieve a list of all available problems from the database.**
- **Response Formatting: Return the list of problems in a structured format, including the problem ID, title, difficulty, and category.**
- **Error Handling: If there is an error during the fetching process, return an error message.**

4)/problems

Method: GET

Request Body: None

Response Body:

Success (HTTP 200 OK):

```
{
"problems": [
  {
```

```
      "id": 1,
      "title": "Two Sum",
      "difficulty": "Easy",
      "categories": ["Array", "Hash Table"]
    },
    {
      "id": 2,
      "title": "Longest Substring Without Repeating Characters",
      "difficulty": "Medium",
      "categories": ["String", "Sliding Window"]
    }
  ]
}
```

Failure (HTTP 500 Internal Server Error):

```
{
  "message": "An error occurred while fetching problems"
}
```

Functionality:

- **Fetch Problems: Retrieve a list of all available problems from the database.**
- **Response Formatting: Return the list of problems in a structured format, including the problem ID, title, difficulty, and categories (which can contain multiple tags).**
- **Error Handling: If there is an error during the fetching process, return an error message.**

5)/problem/<id>

Method: GET

Request Body: None

Response Body:

Success (HTTP 200 OK):

```
{
  "id": 1,
  "title": "Two Sum",
  "description": "Given an array of integers, return indices of the two numbers such that they add up to a specific target...",
  "difficulty": "Easy",
  "categories": ["Array", "Hash Table"],
  "examples": [
    {
      "input": "nums = [2, 7, 11, 15], target = 9",
      "output": "[0, 1]"
    }
  ]
}
```

Failure (HTTP 404 Not Found):

```
{
  "message": "Problem not found"
}
```

Functionality:

- **Fetch Problem Details: Retrieve detailed information for a specific problem based on the provided problem ID.**

- **Response Formatting: Return the problem's ID, title, description, difficulty, categories, and example inputs and outputs.**
- **Error Handling: If the problem with the specified ID is not found, return an error message**

6)/submit

Method: POST

Request Body:

```
{
  "user_id": 1,
  "problem_id": 1,
  "code": "your_solution_code_here"
}
```

Response Body:

Success (HTTP 200 OK):

```
{
  "message": "Submission successful",
  "submission_id": 123,
  "status": "Accepted",
  "runtime": "50ms"
}
```

Failure (HTTP 400 Bad Request):

```
{
  "message": "Submission failed",
```

```
  "error": "Compilation error or runtime error details"
}
```

Functionality:

- **Code Execution: Run the provided code against the hidden test cases for the specified problem.**
- **Result Evaluation: Check if the code passes all test cases and determine the runtime.**
- **Record Submission: Save the submission details, including the user ID, problem ID, code, status, and runtime, in the database.**
- **Response Formatting: Return the submission status, ID, and runtime if successful, or return an error message if the submission fails.**

7)/submissions

Method: GET

Request Body: None

Response Body:

Success (HTTP 200 OK):

```
{
  "submissions": [
    {
      "submission_id": 123,
      "user_id": 1,
      "problem_id": 1,
      "status": "Accepted",
      "runtime": "50ms",
      "timestamp": "2024-07-10T12:34:56Z"
    },
```

```
    {
      "submission_id": 124,
      "user_id": 2,
      "problem_id": 2,
      "status": "Accepted",
      "runtime": "75ms",
      "timestamp": "2024-07-10T12:45:30Z"
    }
  ]
}
```

Failure (HTTP 500 Internal Server Error):

```
{
  "message": "An error occurred while fetching submissions"
}
```

Functionality:

- **Fetch Submissions: Retrieve the last 10 submissions made by all users from the database.**
- **Response Formatting: Return the list of submissions, including submission ID, user ID, problem ID, status, runtime, and timestamp.**
- **Error Handling: If there is an error during the fetching process, return an error message.**

8)/leaderboard

Method: GET

Request Body: None

Response Body:

Success (HTTP 200 OK):


```
{
  "leaderboard": [
    {
      "user_id": 1,
      "name": "John Doe",
      "problems_solved": 50
    },
    {
      "user_id": 2,
      "name": "Jane Smith",
      "problems_solved": 45
    },
    {
      "user_id": 3,
      "name": "Alice Johnson",
      "problems_solved": 40
    },
    {
      "user_id": 4,
      "name": "Bob Brown",
      "problems_solved": 35
    },
    {
```

```
      "user_id": 5,
      "name": "Charlie Davis",
      "problems_solved": 30
    }
  ]
}
```

Failure (HTTP 500 Internal Server Error):

```
{
  "message": "An error occurred while fetching the leaderboard"
}
```

Functionality:

- **Fetch Leaderboard: Retrieve the top 5 users with the most problems solved from the database.**
- **Response Formatting: Return the leaderboard list, including user ID, name, and the number of problems solved.**
- **Error Handling: If there is an error during the fetching process, return an error message.**

9)/profile/<user_id>

Method: GET

Request Body: None

Response Body:

Success (HTTP 200 OK):

```
{
  "user_id": 1,
  "name": "John Doe",
```

```
  "email": "john.doe@example.com",
  "problems_solved": [
    {
      "problem_id": 1,
      "title": "Two Sum"
    },
    {
      "problem_id": 2,
      "title": "Longest Substring Without Repeating Characters"
    }
  ]
}
```

Failure (HTTP 404 Not Found):

```
{
  "message": "User not found"
}
```

Functionality:

- **Fetch User Profile: Retrieve the user's profile details, including name, email, and the list of problems they have solved.**
- **Response Formatting: Return the user's profile information in a structured format.**
- **Error Handling: If the user with the specified ID is not found, return an error message.**

Database:SQLITE

Table: User Details

- **Columns**:
  - **id**: Integer, Primary Key, Auto-increment
  - **name**: Text, Not Null
  - **email**: Text, Unique, Not Null
  - **password**: Text, Not Null
  - **prob_solved**: Integer, Not Null

Table: Problem

- **Columns**:
  - **id**: Integer, Primary Key, Auto-increment
  - **title**: Text, Not Null
  - **description**: Text, Not Null
  - **difficulty**: Text, Not Null
  - **categories**: Text, Not Null (can store as a comma-separated list or a JSON array)

Table: Submission

- **Columns**:
  - **id**: Integer, Primary Key, Auto-increment
  - **user_id**: Integer, Foreign Key referencing User(id), Not Null
  - **problem_id**: Integer, Foreign Key referencing Problem(id), Not Null
  - **status**: Text, Not Null
  - **runtime**: Text, Not Null
  - **timestamp**: Timestamp, Not Null

Table: TestCase

- **Columns**:
  - **id**: Integer, Primary Key, Auto-increment
  - **problem_id**: Integer, Foreign Key referencing Problem(id), Not Null
  - **input**: Text, Not Null
  - **expected_output**: Text, Not Null

Docker

When using Docker for an Online Judge platform, several security concerns need to be addressed to ensure the platform's integrity and availability. Here are the common security flaws and solutions:

1. Limiting Code Execution Time

- **Flaw**: User-submitted code running indefinitely or taking too long to execute can tie up resources and impact system performance.
- **Solution**: Use Docker's `--ulimit` option to set a time limit for container processes. Additionally, implement a timeout mechanism in your application to kill long-running processes.

```
docker run --ulimit cpu=10 <image>
```

2. Limiting Resource Consumption

- **Flaw**: Unrestricted resource usage by user code can exhaust system resources, leading to denial of service.
- **Solution**: Use Docker's resource limits to constrain CPU and memory usage for each container.

```
docker run --cpus="1.0" --memory="512m" <image>
```

3. Preventing Access to System Files

- **Flaw**: User code accessing or modifying system files can compromise system security.
- **Solution**: Use Docker's filesystem isolation features. Ensure containers are run with limited permissions and use `read-only` file systems when possible.

```
docker run --read-only <image>
```

4. Preventing Malicious Code Execution

- **Flaw**: Executing malicious code can lead to security breaches, data theft, or system damage.

- **Solution**: Use Docker's security options like user namespaces, AppArmor, or SELinux profiles to restrict container capabilities.

```
docker run --security-opt="apparmor=default" <image>
```

5. Ensuring Network Isolation

- **Flaw**: Unrestricted network access can lead to network-based attacks or unauthorized data exfiltration.
- **Solution**: Use Docker's network isolation features to create isolated networks for containers, and limit network communication using firewall rules and Docker's network policies.

```
docker network create --driver bridge isolated_network
```