High-Level Design Document For OnlineJudge (MERN)

Overview:

Online Judge is an online platform that allows users to solve coding problems, execute and manage their code, submit their solutions, and receive instant verdicts. The platform accommodates multiple programming languages and is designed to scale for competitions where participants work on solving a set of coding questions within a defined time window. The submitted solutions are evaluated against predefined test cases and scores are assigned accordingly.

Functional Requirements:

1) Users should be able to view a list of coding problems.
2) Users should be able to view a given problem, code a solution in various languages.
3) Users should be able to submit their solutions and get instant verdicts.
4) User authentication using JWT.
5) User profile management.
6) Leaderboard.
7) Practice Problems.
8) Evaluation & Scoring of submitted solutions against the test cases.

Non-functional Requirements:

1) Priority on availability and scalability.
2) Security and isolation of code execution.
3) Fast response times (<5 secs).
4) Support for large-scale competitions.
5) Well-tested and easy to deploy using CI/CD pipelines.
6) Regular backups.

Core Entities:

To satisfy our key functional requirements, we will need following entities:

- Problem - Will store problem statements, test cases and the expected output.
- Submission - Will store the user's code submission and the result of running the code against the test cases.
- User - Will store the authentication details of the user.

API or System Interface:

1) **View all problems:** To view the list of problems, we will have a simple GET endpoint that returns the list with only required details from that entity. This will have pagination since we have more problems than should be returned in a single page.

   ```
   GET /problems?page=1&limit=30
   ```

2) **View a specific problem:** This will be another GET endpoint that takes problem ID (we will get this when a user clicked on a problem from the problems list) and returns the full problem details with code stubs. Query parameter for language which can have any default language if not provided, helps in providing code stub in the user's preferred language.

   ```
   GET /problems/:id?language={language}
   ```

3) **Submit a solution:** This will be a POST endpoint that takes a problem ID and the user's code submission and returns the result of running the code against the test cases. For security reasons, we are not passing userId into this endpoint, assuming that user is authenticated and userId is stored in the JWT session.

   ```
   POST /problems/:id/submit
   Body - code: string, language: string
   ```

4) **User authentication using JWT:** This will have two POST endpoints that registers and authenticates the user and returns a JWT token.

POST /auth/login
Body - username / email: string, password: string
Response - token: string

POST /auth/register
Body - username: string, email: string, password: string
Response - token: string

5) **User profile management:** This will have two endpoints, one is GET endpoint that retrieves all the details of the authenticated user's profile, another is PUT endpoint that helps user to update his details like username or password.

GET /user/profile
PUT /user/profile
Body - username: string, password: string

User Schema:

id: string,
username: string,
email: string,
password: string,
profilePicture: string,
followers: ["string"],
following: ["string"],
bio: string,
location: string,
skills: string,
solvedProblems: ["string"],
favoriteProblems: ["string"],
submissionHistory: []
    problemId: string,
    submissionId: string,
    submissionDate: string,
    status: string,
    language: string,

6) **Leaderboard:** This will have a GET endpoint that retrieves the leaderboard rankings with pagination since we have more users than should be returned in a single page.

   GET /leaderboard?page=1&limit=30

7) **Practice problems:** This will have a GET endpoint that retrieves a list of practice problems for the user with pagination since we have more problems than should be returned in a single page.

   GET /practice-problems?page=1&limit=30

8) **Evaluation & Scoring:** This will also have a POST endpoint that evaluates user's code submission and returns the score and details about no. of test cases passed.

   POST /problems/:id/evaluate

Design:

Relative to all other applications like social media (facebook, instagram, whatsapp), youtube, etc. this is a small-scale application. So for this, a monolithic architecture might be more appropriate, as this system can be easily managed as a single codebase and the overhead of managing multiple services is not worth it. We can go for simple client-server architecture for this application.

1) **View all problems:** The server will handle incoming client requests and return appropriate data. This is where we store all the problems in the database such that they are indexed properly to support pagination. I am using a No-SQL database here i.e MongoDB, as we dont need complex queries and i plan to nest the test cases as a subdocument in the problem entity.

Problem Schema:

```
id: string,
title: string,
statement: string,
tags: string,
level: string,
testcases: []
        input: string,
        output: string
codestubs: []
        cpp: string,
        java: string,
        python: string
```

Code stubs for each language will be entered manually by the admin or can be generated automatically given a single language.

2) **View a single problem:** The client will make a request to the API server and the server will return the full problem statement and code stub after fetching it from the database. We can use packages for a coding editor to allow users to code in the browser.

3) **Submit a solution:** When a user submits their solution, we need to run their code against the test cases and return the result. We should be very careful about how we run the code to ensure that it doesn't crash our server.

    - **Run code in API server:** The simplest way to run the user submitted code is to run it directly in the API server. This means saving the code to a file in our local filesystem, running it, and capturing the output.

      Challenges: Security, Performance, Isolation

- **Run code in VM:** A better way to run user submitted code is to run it in a virtual machine (VM) on the physical API Server. A VM is an isolated environment that runs on top of your server and can be easily reset if something goes wrong. This means that even if the user's code crashes the VM, it won't affect your server or other users.

  Challenges: Resource intensive, Slow at start up, Costly.

- **Run code in container (Docker):** To enhance scalability, we can run user-submitted code in containers. Containers offer an isolated environment for executing code, similar to virtual machines (VMs), but with significantly less overhead and faster startup times. Our approach involves creating a separate container for each supported runtime (e.g., CPP, Java, Python). These containers will install the required dependencies and execute the code within a secure environment. By reusing the same containers for multiple submissions, we can optimize resource utilization and enhance overall performance.

  Challenges: We need to properly configure and secure the containers to prevent users from breaking out of the container and accessing the host system. We also need to enforce resource limits to prevent any single submission from utilizing excessive system resources and affecting other users.

When a user makes a submission, our server will:

- Receive the user's code submission and problem ID and send it to the appropriate container for the language specified.
- The isolated container runs the user's code in a secured environment and returns the result to the API server.
- Server will then store the submission results in the database and return the results to the client.

  But there are a few things we'll want to include in our container setup to further enhance security: Read only system, CPU and memory bounds, explicit timeout.

The main concern here is that we get a sudden spike in traffic, say from a competition or a popular problem, that could overwhelm the containers running the user code. However for our small-scale application as mentioned above, horizontal scaling should be able to handle this load without any issues.

Dynamic Horizontal Scaling Approach: To enhance scalability, we can horizontally scale each of the language-specific containers. By spinning up multiple containers for each language, we can distribute submissions across them. This dynamic scaling can be achieved through auto-scaling groups, which adjust the number of active instances based on real-time traffic demands, CPU utilization, or other memory metrics.

Challenges: If we spin up too many containers, we could end up wasting resources and incurring unnecessary costs. That said, modern cloud providers make it easy to scale up and down based on demand, so this is a manageable risk.

Horizontal Scaling w/ Queue: We can take the same exact approach as above but add a queue between the API server and the containers. This will allow us to buffer submissions during peak times and ensure that we don't overwhelm the containers.

When a user submits their code, the API server will add the submission to the queue and the containers will pull submissions off the queue as they become available. This will help us manage the load on the containers and ensure that we don't lose any submissions during peak times. With the introduction of the queue, the system has become asynchronous. Consequently, the API server can no longer immediately return submission results. Instead, users must poll the server for updates.

We need to add a new endpoint, such as which clients can poll every second to determine if the submission has been processed. If the results are available, they are returned. otherwise, a 'processing' message is provided. While we can also use persistent connections like WebSockets to avoid polling, this approach adds complexity and may not be necessary given the short polling interval (1s).

GET /check/:id

Challenges: While introducing a queue might address some scalability concerns, it could lead to unnecessary complexity. Considering the system's scale, it's possible to handle the load without relying on a queue. Additionally, if we mandate user registration for competitions, we can anticipate peak traffic and proactively scale up containers as needed.

Submission Schema:

```
id: string,
userId: string,
problemId: string,
code: string,
language: string,
result: string,
createdAt: string
```

Running test cases: We don't generate test cases for problems for each language, as test cases will be the same.

Pin -> code == solution.txt
solution.txt == Pout (based on this verdict accepted or rejected will be given)

Choice of Tech Stack:

Frontend:

- **React**: Efficient component-based architecture.
- **Advantages**: Reusable components, fast rendering, rich ecosystem.
- **Challenges**: Initial setup complexity, larger bundle sizes if not optimized.

Backend:

- **Node.js with Express**: Asynchronous, event-driven, suitable for I/O heavy operations.
- **Advantages**: Non-blocking I/O, large community support.

- **Challenges**: Callback hell (manageable with promises/async-await), single-threaded (manageable with clustering).

Database:

- **MongoDB**: NoSQL database, flexible schema.
- **Advantages**: Easy to scale, flexible document structure.
- **Challenges**: Lack of complex transaction support, eventual consistency.

Containerization:

- **Docker**: Containerization platform.
- **Advantages**: Isolation, portability, efficient resource utilization.
- **Challenges**: Security configurations, learning curve.

Authentication:

- **JWT**: JSON Web Tokens for authentication.
- **Advantages**: Stateless, secure, easy to use.
- **Challenges**: Token expiration handling, storage.

CI/CD:

- **GitHub Actions / Jenkins**: Continuous integration and deployment.
- **Advantages**: Automated testing, easy deployment, consistency.
- **Challenges**: Configuration complexity, integration with various services.

Conclusion: OnlineJudge provides a scalable and secure platform for solving coding problems with instant verdict. By leveraging modern containerization techniques and a robust API-driven architecture, it ensures high availability and responsiveness for users, even during large-scale competitions.