Design Leetcode - Online Coding Judge

Practiced On: 8th feb 2025

Time Taken: 65 Mins (1 Hour 5 Mins)

Gave this mock in evening time, had low energy
so couldn't gave my best
I am sure could have done better if given in morning hours
Need to work on this

Qs for the interviewer

1. Do user need to sign in?
2. How many languages are supported?
3. How many DAU and total users are there on platform.
4. Do we need competition? How many users will be there?

Requirements

Functional Requirements:
1. User should be able to see list of coding problems
2. Users should be able to click a Q(coding problem) & see its details
3. User will be able to write code in their respective language and submit the same
4. User will be able to participate in a competition
5. User will be able to see the leaderboard in competition
6. How is score for leaderboard is decided?

Non Functional requirements:
1. System should have high availability (availability > consistency)
2. System should show near realtime leaderboard
3. System should have low latency for code submissions and get response
4. System should be secure

Entity

1. User > user_id, email_id, last_login, default_lang
2. Problem > id, title, description, testCase [], diff, topic, company
3. User submitted solution > submitId, submittedBy, testCaseResult[], timing, lang, score
4. Competition > id, problem [], date, timing, leaderboard

API design

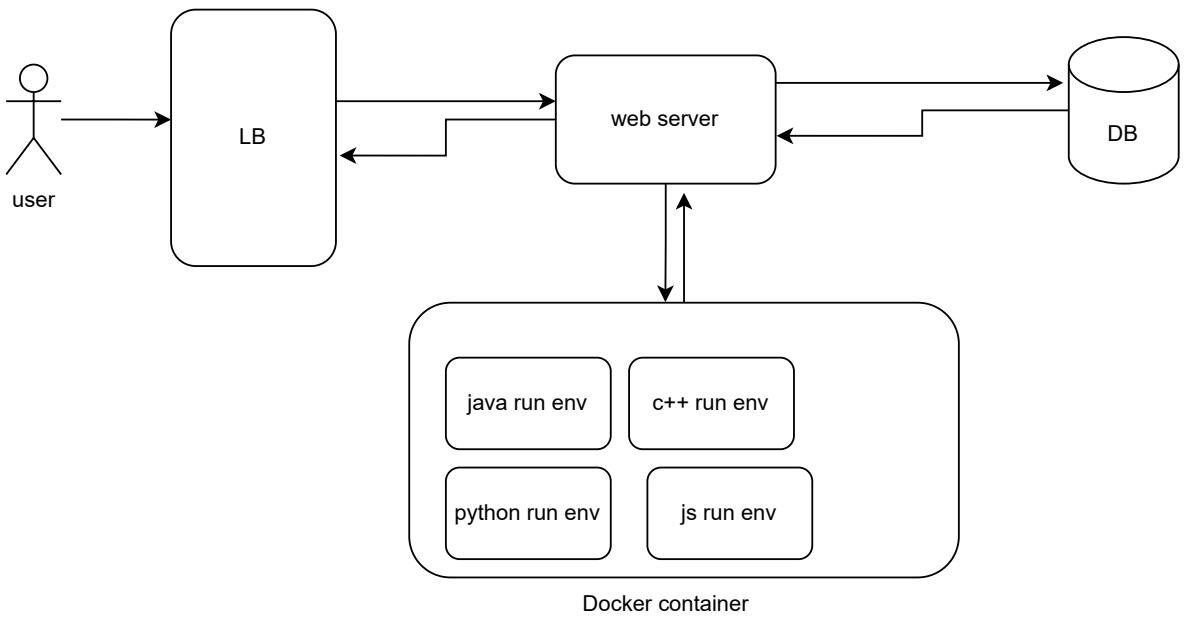
1. GET list of problems
    GET HTTP/allProblems?difficulty={}?topic={}?frequency={}?company={}?page={}
    RESPONSE JSON {
        list<problems>
        // at the point problem obj won't have all detail, only have title,
        //small description, topic, difficulty, frequency and company tag
    }

2. GET a specific problem
    GET HTTP/problem/:id
    RESPONSE JSON {
        title: "",
        description: "",
        sample:"",
        topic, difficulty, frequency, companyTag
    }

3. POST submit a code for a problem
    POST /problem/:id/submission
    body {
        language:"",
        user_id:"",
        code:"" ,
        customeTestCase:""
    }
    RESPONSE JSON {
        response_object{}
    }

4. GET leaderboard
    GET HTTP/competition/:id/leaderboard
    RESONSE JSON {
        List<Users> sorted by score # this will be paginated response
    }

High Level Design - draft 1



Flow:

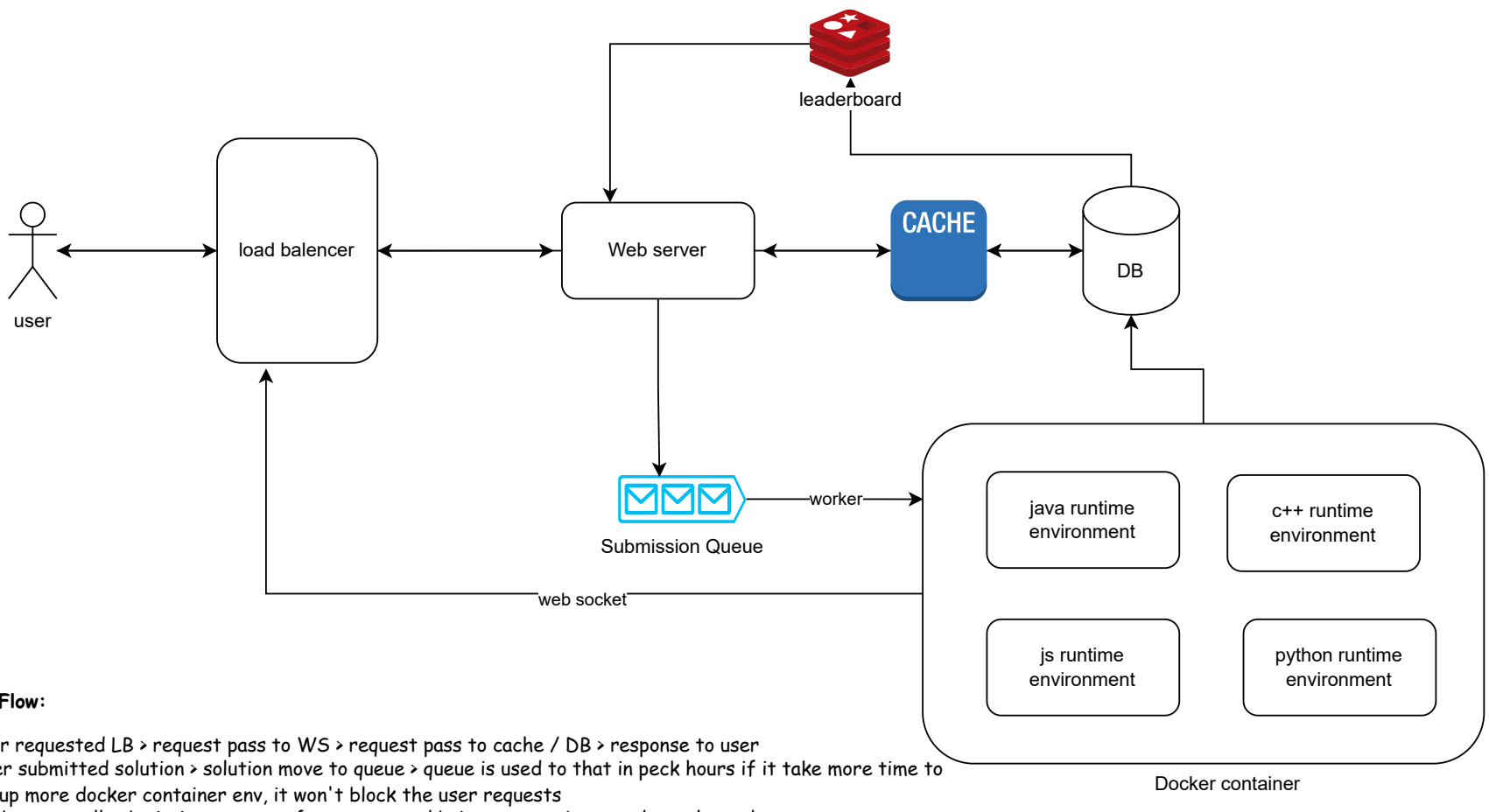
1. User request for /allProblems > passes through LB > request hit WS
2. WB hit database > get all the metadata and Qs for with the filters that user have requested with
3. DB return response to WS > WS return to user > user browser
4. User clicked on a specific problem > id passed to WS > pass to DB > DB return details of the problem

5. User codes the solution and submits
6. code come to WS
    > it won't be safe to run / execute an external code on our own WS
    > for security we can run the code on VM > but VM are not lightweight, so we can use docker container
    > code will be executed in a docker container env with diff test case
    > response is returned back to WS > return back result to user

Design Deep Dive

1. What kind to database to keep > As we need high availability and also will not be using much of the ACID properties we can have a no-SQL database
2. We can add cache before the DB to dec latency
3. To increase scalability one can add more servers
4. Competition: For completion, we need a leaderboard feature that can be a redis cache using sorted set

Final High Level Design



Final Flow:

1. User requested LB > request pass to WS > request pass to cache / DB > response to user
2. User submitted solution > solution move to queue > queue is used to that in peck hours if it take more time to bring up more docker container env, it won't block the user requests
3. worker can pull submission requests from queue and bring up containers and run the code
4. user's can be connected via web socket with container > response will be send back to user
5. Competition > user can vie leaderboard via redis cache that is updated by DB as user submits a solution

Design review by ChatGPT

Strengths:

1. **Well-Defined Functional & Non-Functional Requirements**
    - You have clearly outlined the key features (problem listing, submissions, competitions, leaderboard).
    - Non-functional aspects like **high availability, low latency, and security** are considered.
2. **Good API Design**
    - The API design follows RESTful principles, making it easy to use and extend.
    - Pagination for problem listing and leaderboard is a good choice for scalability.
    - Using query parameters for filtering problems is effective.
3. **Security Considerations for Code Execution**
    - Running user-submitted code in **Docker containers** instead of directly on the web server is a great approach to security.
    - You acknowledge the **resource-heavy nature of VMs** and optimize with containers.
4. **Scalability and Performance Considerations**
    - **Load balancer** to distribute requests.
    - **Submission queue** prevents request blocking during peak times.
    - **Redis sorted set** for leaderboard caching reduces DB reads.
5. **WebSocket for Real-time Updates**
    - Using **WebSockets** to keep users updated on submission results improves UX.

Areas for Improvement:

1. Database Choice & Structure

- **You mentioned using NoSQL for high availability, but do all entities fit this model?**
    - **Problems & Leaderboard:** NoSQL (e.g., MongoDB, Redis) makes sense.
    - **Submissions & Users:** These may benefit from **a relational DB** (e.g., PostgreSQL, MySQL) because:
        - Ensuring data integrity (e.g., unique email per user).
        - Querying past submissions efficiently.
    - Consider **polyglot persistence**:
        - **User, Submissions → SQL (ACID needed).**
        - **Problems, Leaderboard → NoSQL (Fast Reads).**

2. Leaderboard Updates

- **How frequently does the Redis cache update from the database?**
    - If real-time updates are required, you might need **event-driven processing** using **Kafka** or **Redis pub/sub**.

3. Handling Multiple Languages Efficiently

- **How do you decide which runtime environment to spin up?**
  - If a Java submission comes in, **do you keep Java containers always running, or create them on demand?**
  - Possible solution:
    - **Pre-warm a pool of containers** for frequently used languages.
    - **Use Kubernetes** to scale container instances dynamically.

4. Competition Management

- **How do you handle scoring logic?**
  - Can users submit multiple times?
  - Are scores calculated based on execution time, correctness, or both?
  - How do you handle cheating (e.g., detecting hardcoded solutions)?

5. Submission Processing Optimization

- **Right now, one worker pulls a submission from the queue and processes it.**
  - What if thousands of submissions come in simultaneously?
  - You may need:
    - **Multiple worker nodes** to handle high submission loads.
    - **Priority queues** for contest submissions to process them faster.

Suggested Improvements:

1. **Consider Hybrid DB Approach**
   - NoSQL (Redis/MongoDB) for fast lookups.
   - SQL (PostgreSQL/MySQL) for user/submission data consistency.
2. **Optimize Container Execution Strategy**
   - **Pre-warmed containers** for popular languages.
   - **Auto-scaling workers** using Kubernetes.
3. **Refine Leaderboard Handling**
   - Use **event-driven updates (Kafka, Redis pub/sub)** for real-time ranking.
   - Store **historical leaderboards** for past competitions.
4. **Improve Submission Processing Scalability**
   - Introduce **worker auto-scaling**.
   - Use **priority-based queueing** for contest submissions.