AuraC² – Aura Contest Control

Assignment #3 — System Design & Initial Implementation

CS499A – Graduation Project (A)
First Semester 2025/2026

Team Members:

- **Amr Bani Irshid** – *ID:* 2022901054 – *Email:* 2022901054@ses.yu.edu.jo
- **Omar Qasem** – *ID:* 2022901052 – *Email:* 2022901052@ses.yu.edu.jo

Supervisor: Nawaf O. Alsrehin
Submission Date: 29 / 11 / 2025
University: Yarmouk University
Department: Computer Science
Github Link: https://github.com/Amr-BaniIrshid/AuraC2
YouTube video: https://youtu.be/nij0FWZfTDM

Section 1 — Introduction

AuraC² (Aura Contest Control) is a web-based contest system designed to replace tools like PC² and Codeforces for competitive programming contests inside Yarmouk University. It allows teams to:

- Register/Login using **JWT Authentication**
- Join a contest and solve problems
- Submit code, which is **sent to a Worker via RabbitMQ Queue**
- Automatically executed using **Judge0 API**
- Get the result through **Result Queue + Notification Service**

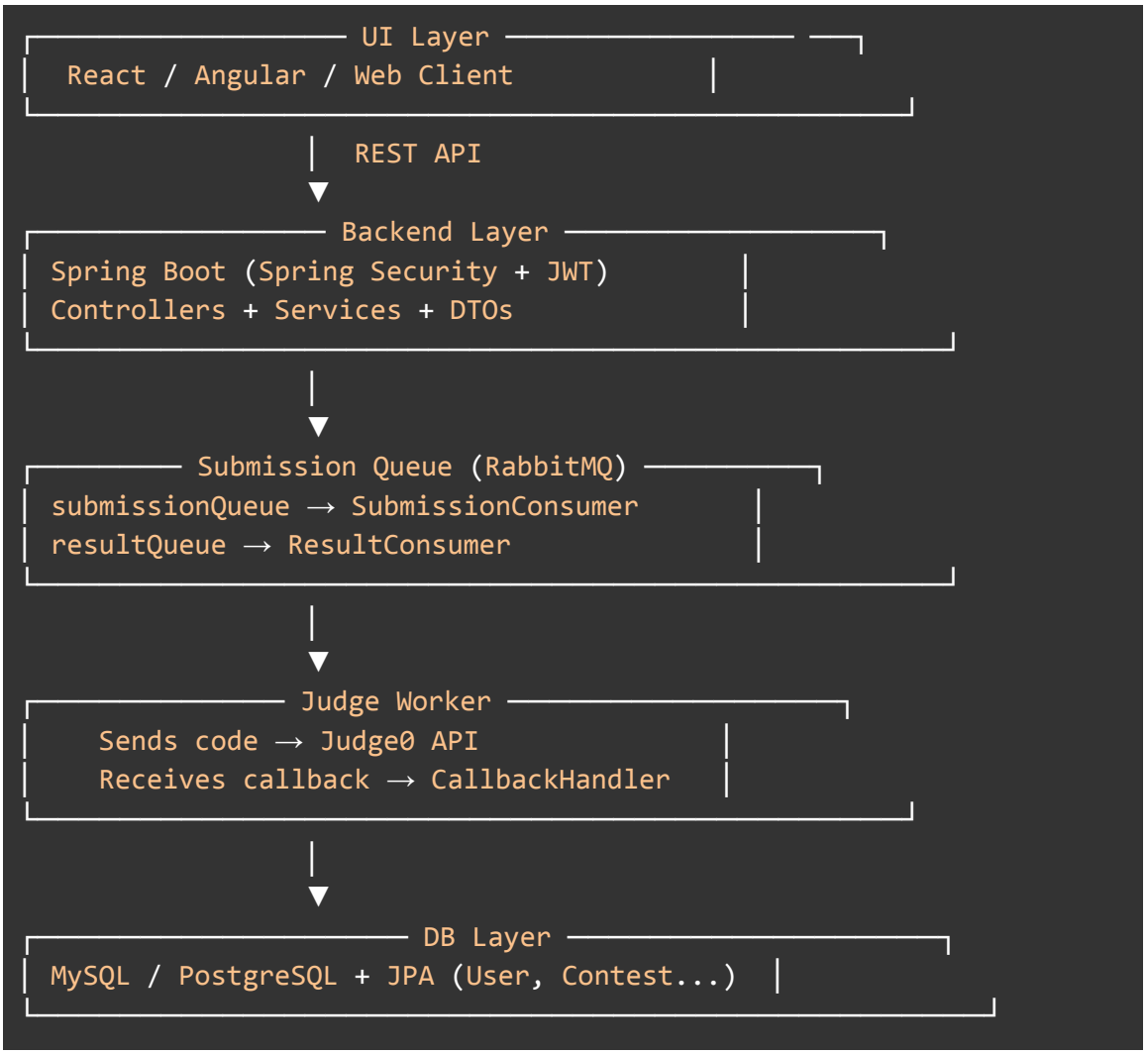
This assignment marks the **transition from analysis to real implementation**.
The focus is on:

-System Architecture
-Sequence Diagrams
-Database + Class Diagrams
-Core Implementation (30–40% finished)

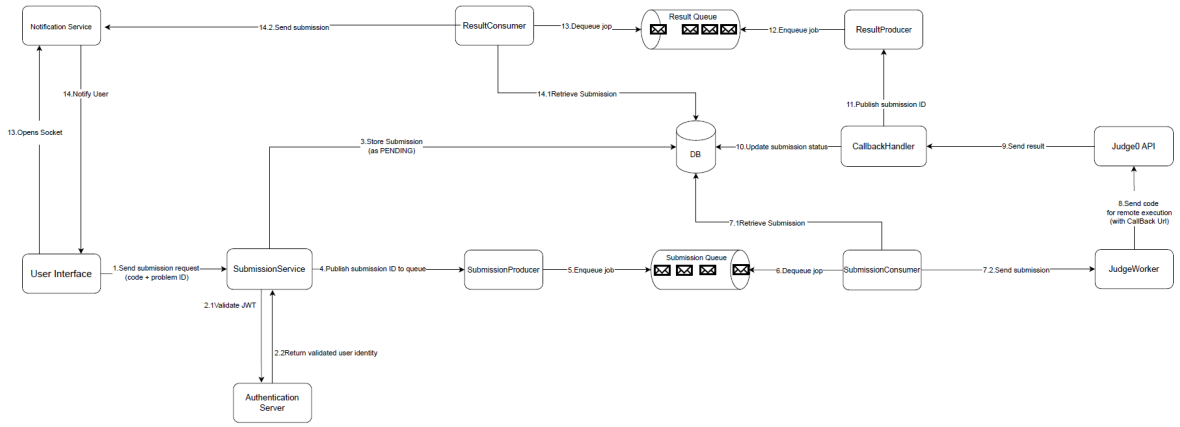
Section 2 — System Architecture Design

2.1 Architecture Diagram

Insert this diagram inside PDF:



🔷 2.2 Full Submission Flow — Final Sequence Diagram



This diagram shows:

- Submission request → JWT validation
- Stored in DB as `PENDING`
- Enqueued using RabbitMQ → JudgeWorker → Judge0
- Result is sent back → DB updated → Notification sent

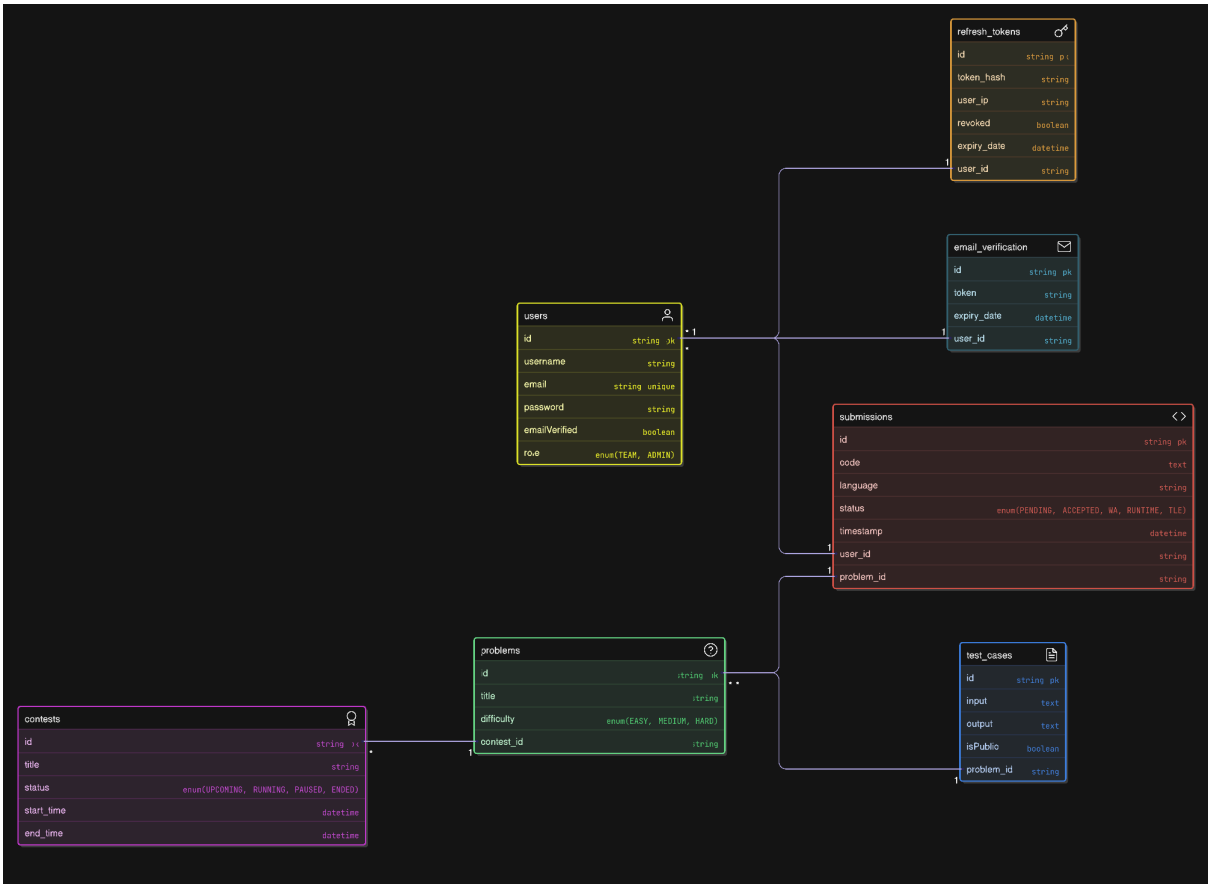
2.3 Use-Case Realization — Example

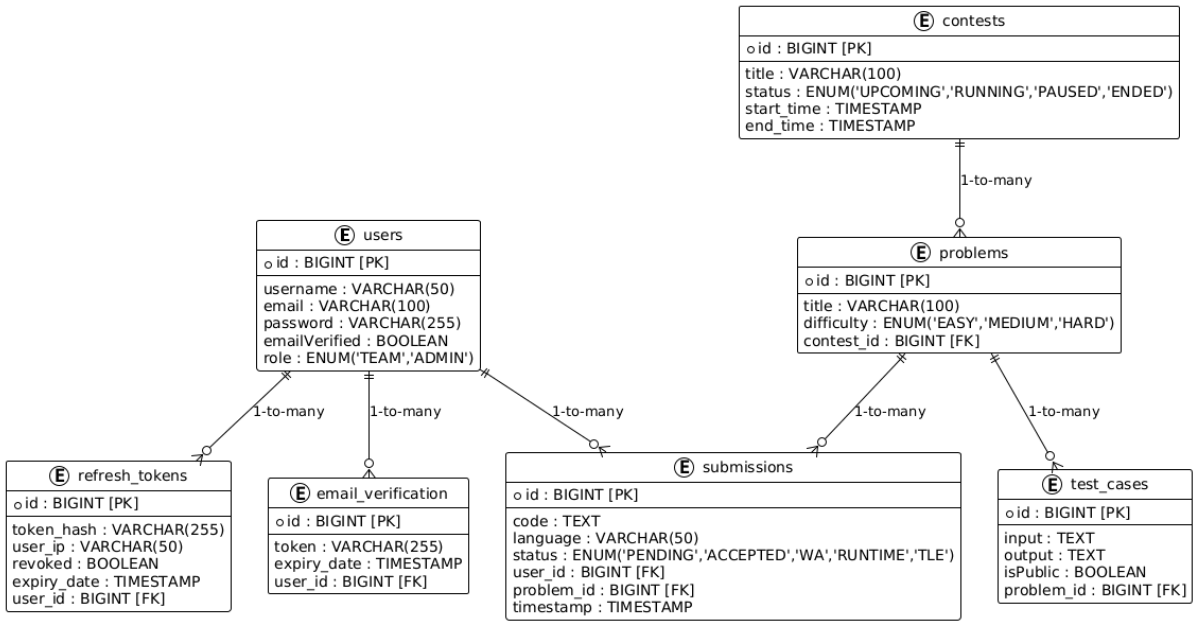
Use-Case: Submit Solution to Problem
Primary Actor: Team User
Goal: Send code → Judge → Receive result

Step	System Action
1	User calls `/api/submissions` with `code + problemId`
2	SubmissionService validates JWT
3	Store in DB as `PENDING`
4	Publish to RabbitMQ `submissionQueue`
5	`SubmissionConsumer` sends code to JudgeWorker
6	Worker calls Judge0 API
7	Judge0 calls `/api/callback`
8	CallbackHandler updates DB
9	ResultConsumer sends message to Notification Service
10	UI notifies team: **Accepted / Wrong Answer / Runtime Error**

Section 3 — Database Design

3.1 ERD (Entity Relationship Diagram)





3.2 Justification of Design Decisions

- Clear Separation of Layers

Using controllers, entities, and DTOs prepares the project for future service layers and follows **Clean Architecture**.
This prevents code coupling and allows UI / logic / database to evolve independently.

> **Alternative considered:** Monolithic `controller + logic + SQL` in one file.
> **Rejected** because it becomes unmaintainable as features grow.

- Single-Responsibility Entities

Every entity in the code represents **one real concept** only (User, Problem, Contest…).
This follows **Domain-Driven Design (DDD)** and makes testing and refactoring easier.

> **Alternative:** All tokens inside `User` table.
> **Rejected** because it breaks separation of concerns and limits OAuth2 & JWT rotation.

- Enum-Based Validation

Enum values (ContestStatus, Role, Difficulty, TokenType…) enforce **data integrity at DB level**, not only in code.
This ensures the data cannot enter an invalid state even during API misuse.

> **Alternative:** `String status` field.
> **Rejected** because it allows typos and invalid values (“endeddd”, “runs”, etc).

- Submission Entity as Core Link

Future ranking, history tracking, and execution analysis depend on connecting **User + Problem + Status**.
Using a dedicated `Submission` table follows systems like **Codeforces** and **LeetCode**.

Alternative: Only storing results in `Problem`.
Rejected because contest systems require tracking multiple attempts per user.

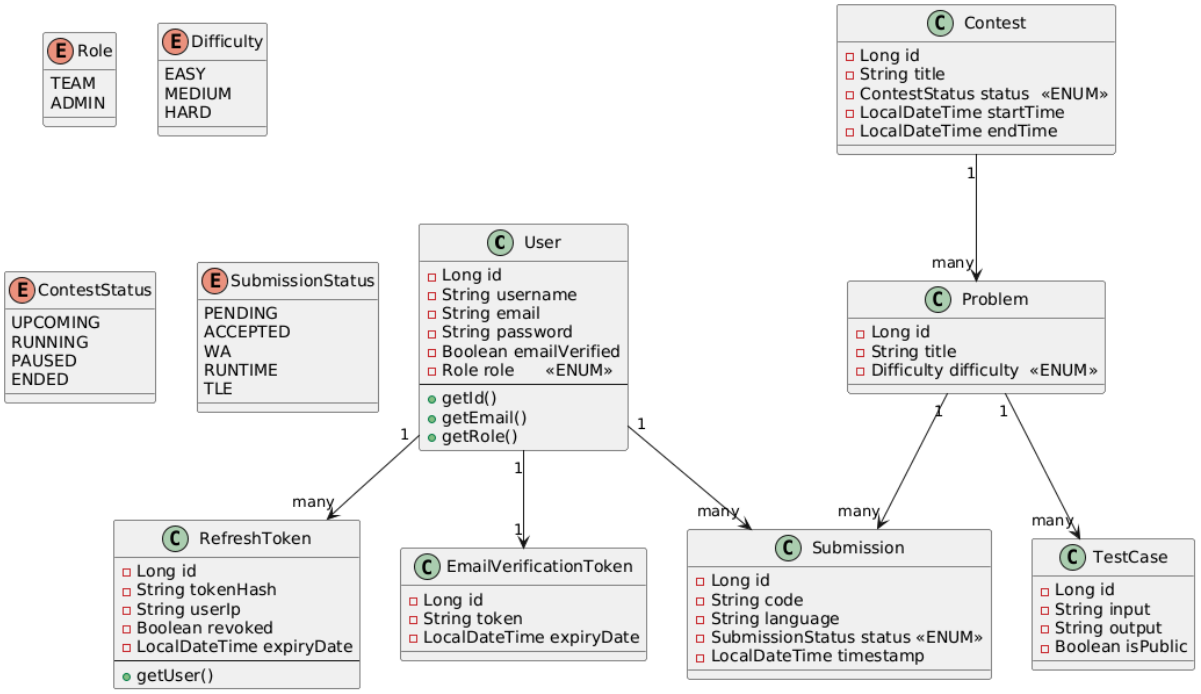
- Security Token Design

Using separate `EmailVerificationToken` and `RefreshToken` entities supports:

1- JWT rotation 2- Revocation 3-Account activation 4-Role-based access control

Section 4 — Class & Module Design

4.1 Class Diagram



4.2 Submission Module Structure

```
SubmissionController     → REST API (POST/GET)

SubmissionService        → Logic / Save Pending

SubmissionProducer       → Publish to submissionQueue

SubmissionConsumer       → Dequeue & send to JudgeWorker

JudgeWorker              → Calls Judge0 API

CallbackHandler          → Updates DB

ResultProducer           → Publishes to resultQueue

ResultConsumer           → Notify UI (WebSocket)
```

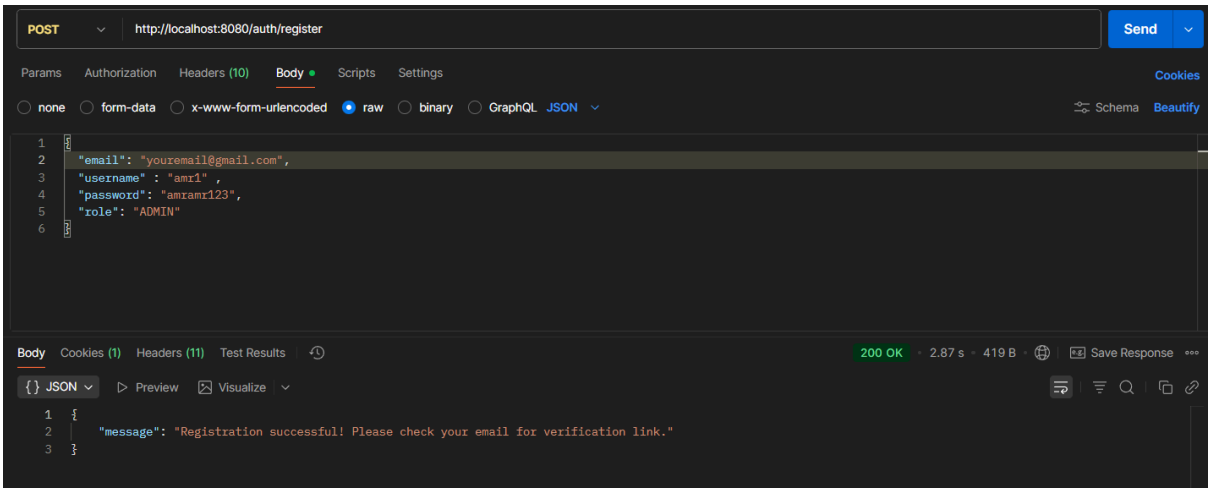
Section 5 — Initial Implementation (30–40% DONE)

Implemented Features

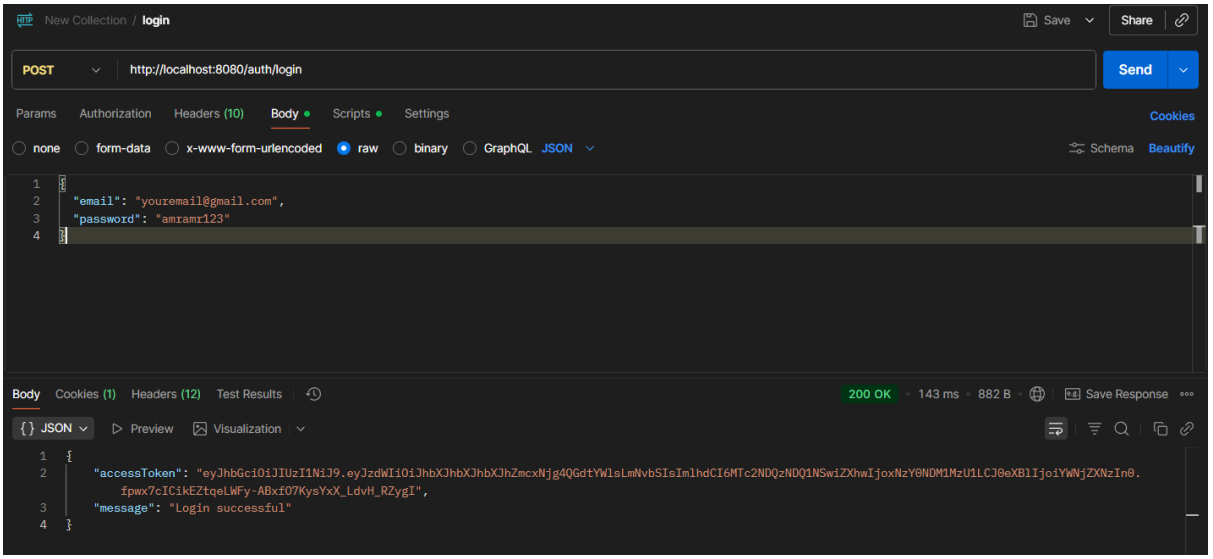
Feature	Status	Proof
JWT Login / Register	Done	Tested via Postman
Email Verification	Done	Gmail SMTP
Refresh Token in Cookie	Done	`secure`, `httpOnly`
Contest Lifecycle	Done	Start, Pause, End
Problems Management	Done	CRUD
TestCases	Done	Linked to Problem
Submission Flow	50% Done	API + Queue config ready

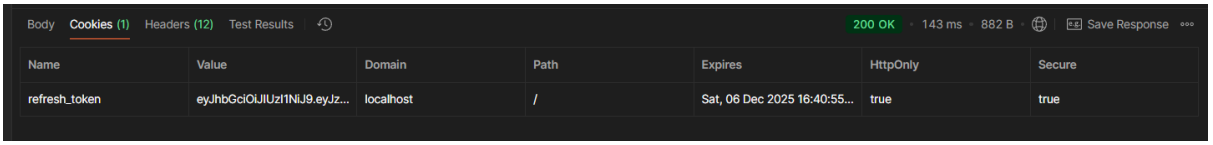
Required Screenshots

Register + Email verification

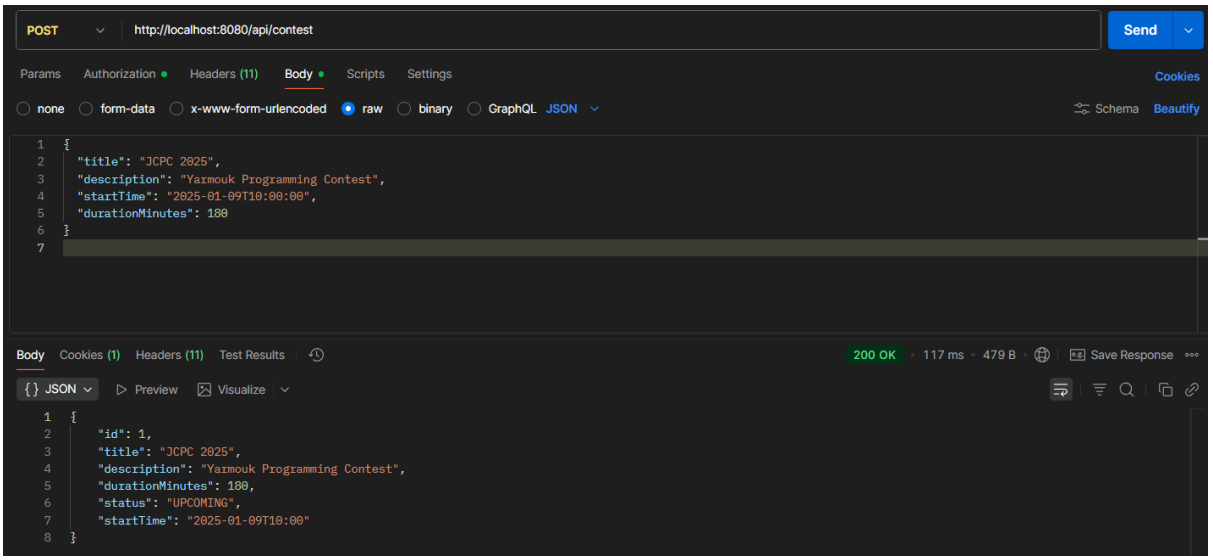


Login → receive accessToken + refreshToken

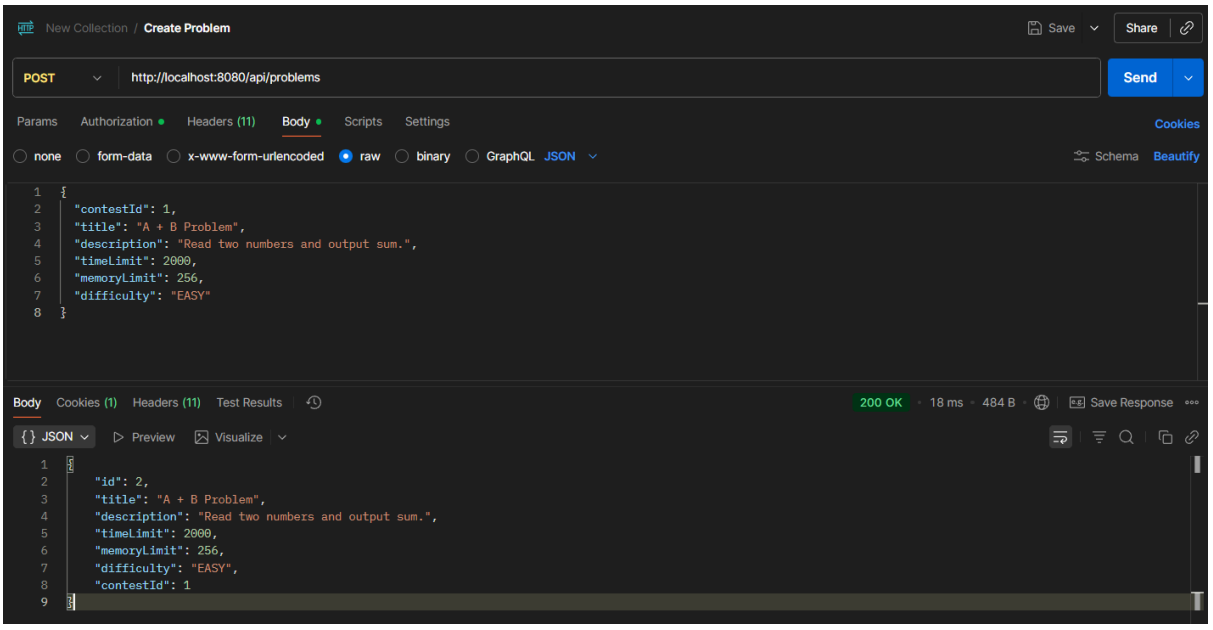




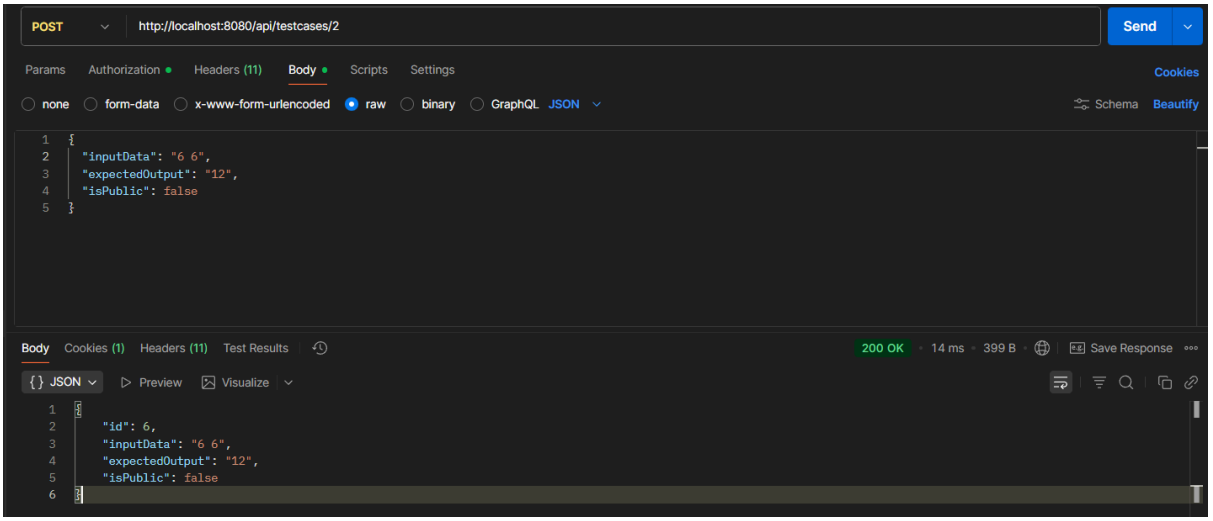
Contest creation (`POST /api/contest`)



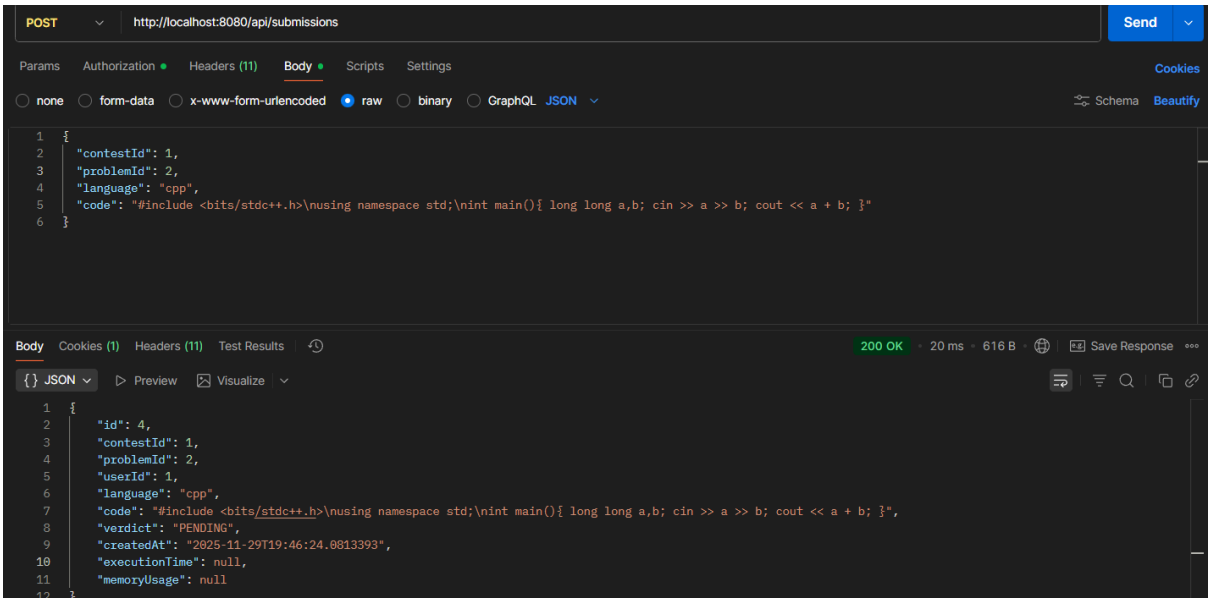
Add problem



Add testcases



Submit code (`POST /api/submissions`)



Example Code Snippet

github link above

Section 6 — Challenges & Solutions

Design Challenge	What was the issue?	Final Design Decision	Why this is a strong solution?
Handling asynchronous code execution	Judge0 API responds late → blocking request	Used **RabbitMQ message queue** instead of direct HTTP	Queue allows **non-blocking execution** + scalable microservice communication
Mapping TestCases to Problems	One problem may contain many test cases → confusion in logic	Used `OneToMany` relation + created clear `problem_id` foreign key	Now the system can **store multiple test cases per problem** and retrieve them cleanly
Identifying which TestCase the user failed on	Submission only stores status (Accepted/WA)	Planned structure: `SubmissionTestCaseResult` table to link which test failed	Makes the grading **explainable + transparent**, like real competitive systems
Connecting User to Contest results	One user participates in multiple contests	Designed `Submission` table to link `user_id + problem_id + timestamp`	Enables **leaderboard + ranking + history tracking**
Avoiding entity leakage in API responses	Entities directly returned by controller	Used **DTO layer** for request/response objects	API is now secure & independent from database structure

Section 7 — Next Steps (Assignment #4)

Planned Feature	Implementation
JudgeWorker → Judge0 Integration	Build HTTP client & mapper
WebSocket Notification	Live UI updates
Submission Status Table	`PENDING
Contest Dashboard	Team Statistics
Frontend Integration	React/Angular