

# AuraC<sup>2</sup> – 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](mailto:2022901054@ses.yu.edu.jo)
- Omar Qasem – ID: 2022901052 – Email: [2022901052@ses.yu.edu.jo](mailto: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

AuraC<sup>2</sup> (Aura Contest Control) is a web-based contest system designed to replace tools like PC<sup>2</sup> 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**.

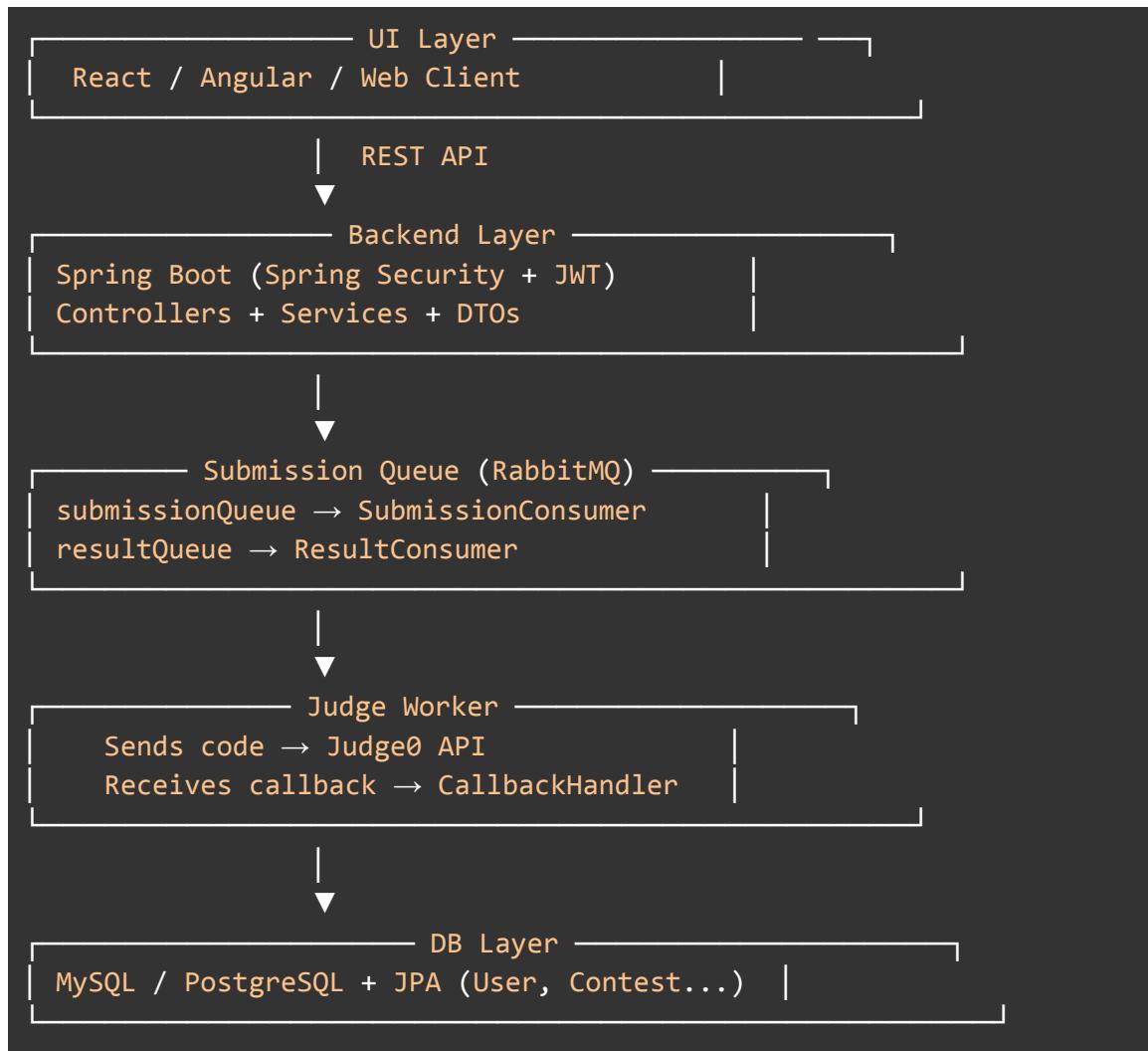
The focus is on:

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

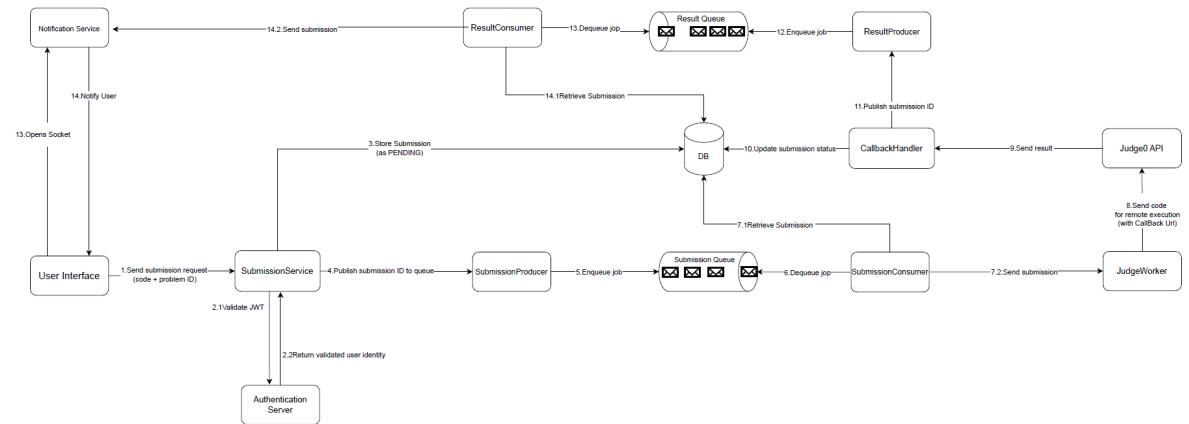
## Section 2 — System Architecture Design

### 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

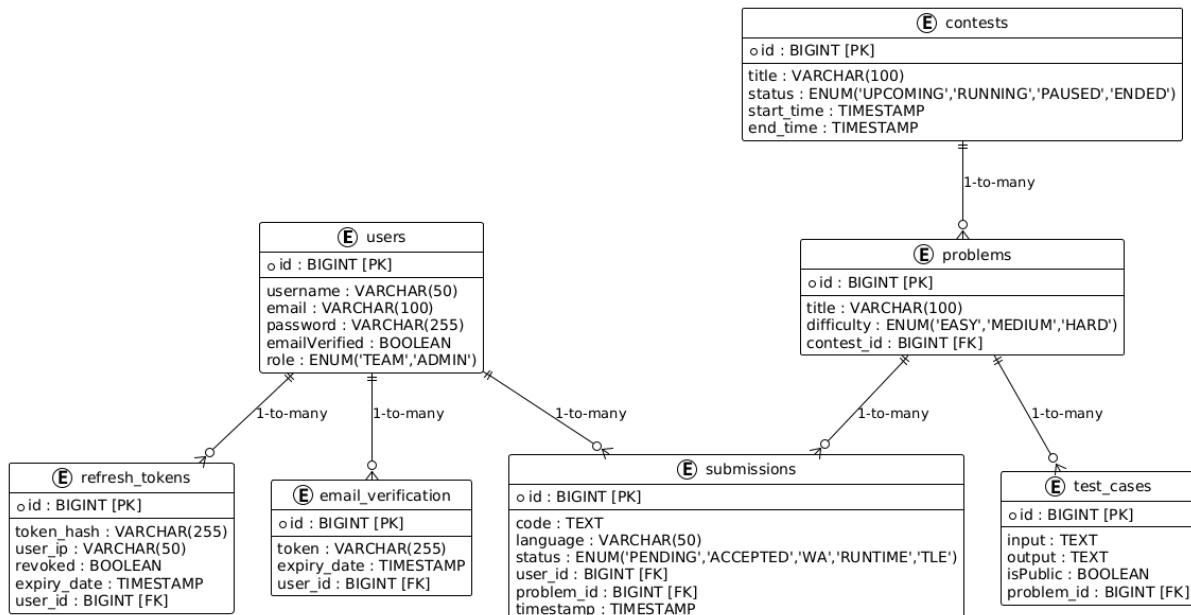
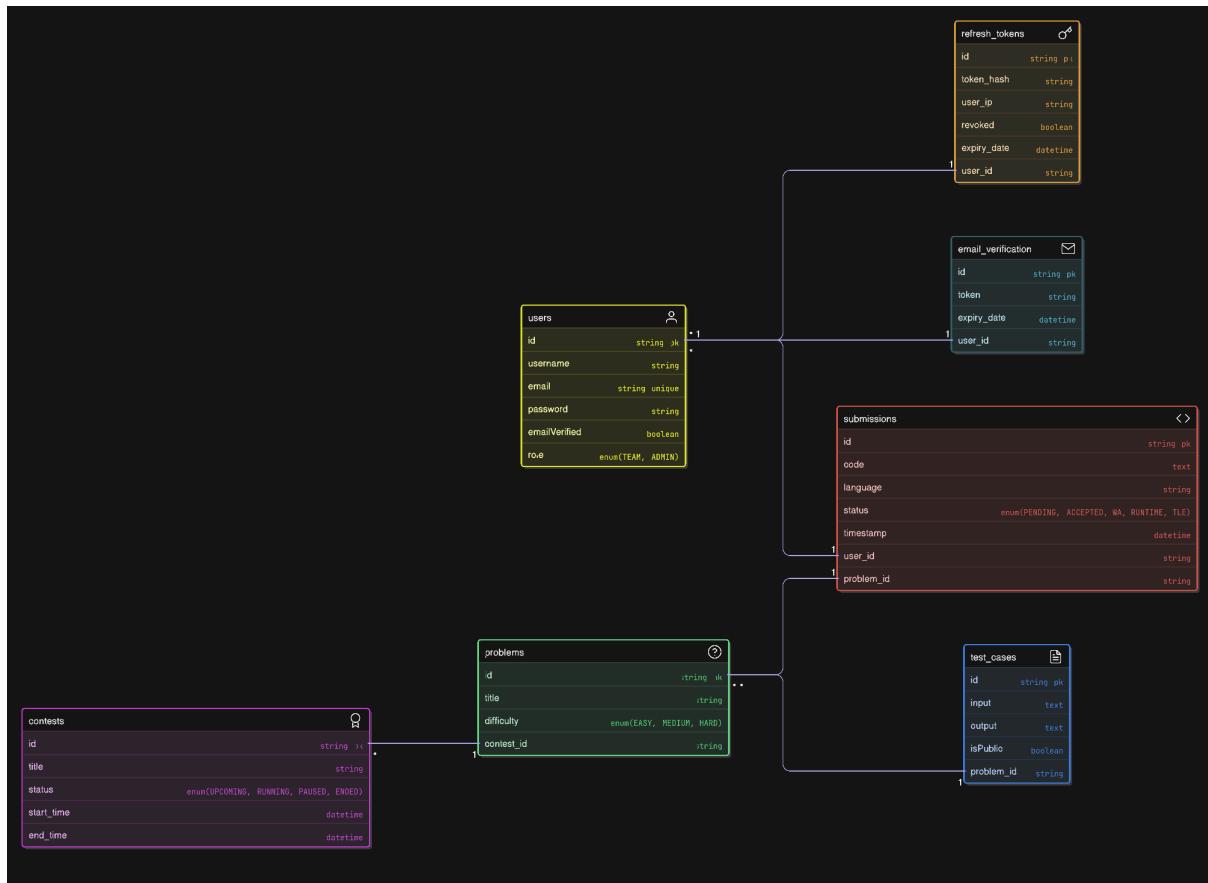
**Primary Actor:** Team User

**Goal:** Send code → Judge → Receive result

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

# 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.

---

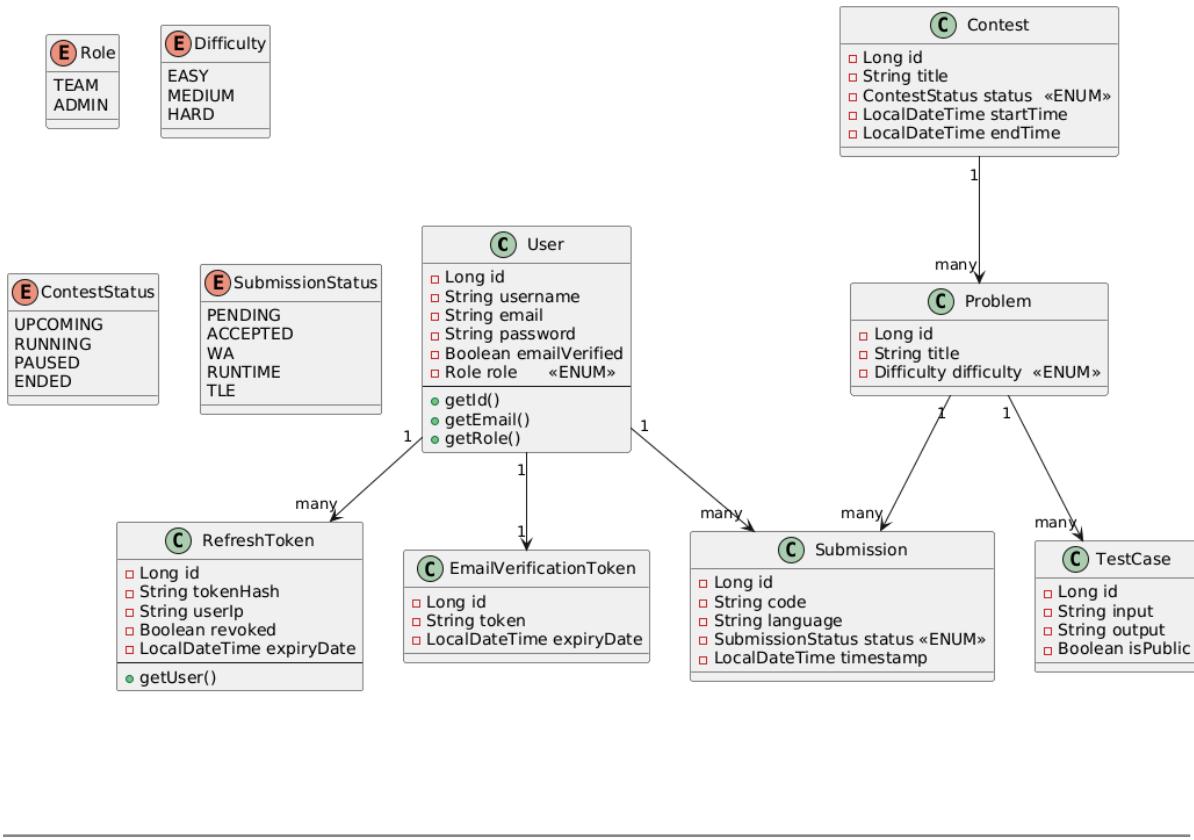
### - Security Token Design

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

The screenshot shows a Postman interface with the following details:

- Method: POST
- URL: <http://localhost:8080/auth/register>
- Body tab selected, showing raw JSON input:

```
1 {
2   "email": "youremail@gmail.com",
3   "username": "amz1",
4   "password": "amzamz123",
5   "role": "ADMIN"
6 }
```

- Response status: 200 OK
- Response time: 2.87 s
- Response size: 419 B
- Response content type: application/json
- Response body (JSON):

```
1 {
2   "message": "Registration successful! Please check your email for verification link."
3 }
```

Login → receive accessToken + refreshToken

## Contest creation (POST /api/contest)

The screenshot shows a POST request to `http://localhost:8080/api/contest`. The request body contains JSON data for a contest:

```
1 {
2     "title": "JCPC 2025",
3     "description": "Yarmouk Programming Contest",
4     "startTime": "2025-01-09T10:00:00",
5     "durationMinutes": 180
6 }
7
```

The response status is 200 OK, with a response time of 117 ms and a response size of 479 B. The response body is:

```
1 {
2     "id": 1,
3     "title": "JCPC 2025",
4     "description": "Yarmouk Programming Contest",
5     "durationMinutes": 180,
6     "status": "UPCOMING",
7     "startTime": "2025-01-09T10:00"
8 }
```

## Add problem

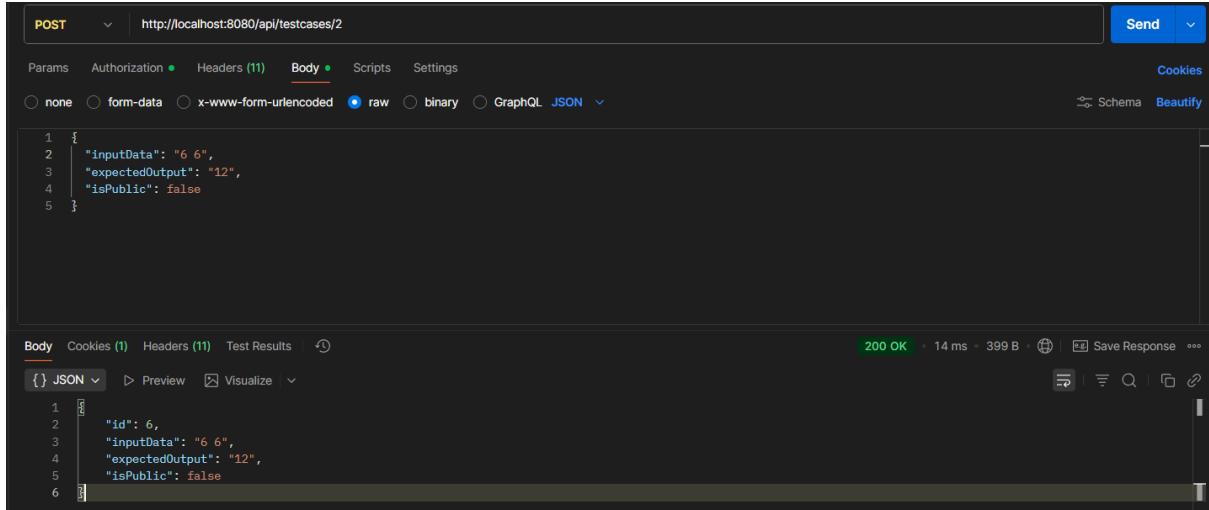
The screenshot shows a POST request to `http://localhost:8080/api/problems`. The request body contains JSON data for a problem:

```
1 {
2     "contestId": 1,
3     "title": "A + B Problem",
4     "description": "Read two numbers and output sum.",
5     "timeLimit": 2000,
6     "memoryLimit": 256,
7     "difficulty": "EASY"
8 }
```

The response status is 200 OK, with a response time of 18 ms and a response size of 484 B. The response body is:

```
1 {
2     "id": 2,
3     "title": "A + B Problem",
4     "description": "Read two numbers and output sum.",
5     "timeLimit": 2000,
6     "memoryLimit": 256,
7     "difficulty": "EASY",
8     "contestId": 1
9 }
```

## Add testcases



POST http://localhost:8080/api/testcases/2

Params Authorization Headers (11) Body Scripts Settings Cookies Schema Beautify

Body none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "id": 6,  
3   "inputData": "6 6",  
4   "expectedOutput": "12",  
5   "isPublic": false  
6 }
```

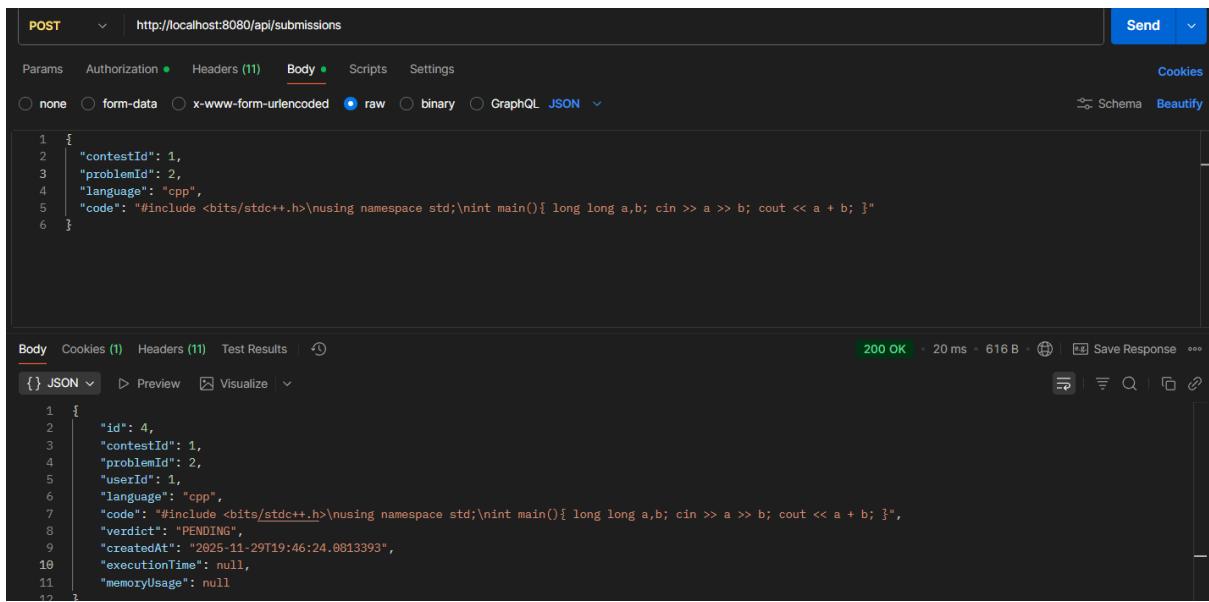
Body Cookies (1) Headers (11) Test Results

200 OK 14 ms 399 B Save Response

{ } JSON Preview Visualize

```
1 {  
2   "id": 6,  
3   "inputData": "6 6",  
4   "expectedOutput": "12",  
5   "isPublic": false  
6 }
```

## Submit code (POST /api/submissions)



POST http://localhost:8080/api/submissions

Params Authorization Headers (11) Body Scripts Settings Cookies Schema Beautify

Body none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "contestId": 1,  
3   "problemId": 2,  
4   "language": "cpp",  
5   "code": "#include <bits/stdc++.h>\nusing namespace std;\nint main(){ long long a,b; cin >> a >> b; cout << a + b; }"  
6 }
```

Body Cookies (1) Headers (11) Test Results

200 OK 20 ms 616 B Save Response

{ } JSON Preview Visualize

```
1 {  
2   "id": 4,  
3   "contestId": 1,  
4   "problemId": 2,  
5   "userId": 1,  
6   "language": "cpp",  
7   "code": "#include <bits/stdc++.h>\nusing namespace std;\nint main(){ long long a,b; cin >> a >> b; cout << a + b; }",  
8   "verdict": "PENDING",  
9   "createdAt": "2025-11-29T19:46:24.0813393",  
10  "executionTime": null,  
11  "memoryUsage": null  
12 }
```

## 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 <b>RabbitMQ message queue</b> instead of direct HTTP	Queue allows <b>non-blocking execution</b> + scalable microservice communication
Mapping TestCases to Problems	One problem may contain many test cases → confusion in logic	Used <b>OneToMany</b> relation + created clear <b>problem_id</b> foreign key	Now the system can <b>store multiple test cases per problem</b> and retrieve them cleanly
Identifying which TestCase the user failed on	Submission only stores status (Accepted/WA)	Planned structure: <b>SubmissionTestCaseResult</b> table to link which test failed	Makes the grading <b>explainable + transparent</b> , like real competitive systems
Connecting User to Contest results	One user participates in multiple contests	Designed <b>Submission</b> table to link <b>user_id + problem_id + timestamp</b>	Enables <b>leaderboard + ranking + history tracking</b>
Avoiding entity leakage in API responses	Entities directly returned by controller	Used <b>DTO layer</b> 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