AURAC² – AURA CONTEST CONTROL: SYSTEM DESIGN AND INITIAL IMPLEMENTATION

Authors

Amr Bani Irshid
Department of Computer Science
Yarmouk University
Irbid, Jordan
2022901054@ses.yu.edu.jo

Omar Qasem
Department of Computer Science
Yarmouk University
Irbid, Jordan
2022901052@ses.yu.edu.jo

Supervisor: Dr. Nawaf O. Alsrehin
Course: CS499A – Graduation Project (A)
First Semester 2025/2026
Submission Date: 29 / 11 / 2025

Github: https://github.com/Amr-BaniIrshid/AuraC2
Video: https://youtu.be/nij0FWZfTDM

Abstract

AuraC² (Aura Contest Control) is a web-based contest management system designed to replace tools like PC² and Codeforces for competitive programming contests inside Yarmouk University. The system supports secure team registration and login using JWT authentication, contest participation, problem management, and automated code judging through a RabbitMQ-based queue and the Judge0 online judge. This report describes the system architecture, the submission flow, the underlying database and class design, and the initial implementation status (about 30–40% complete). It also discusses challenges faced during design and motivates the chosen solutions.

1. Introduction

Competitive programming contests at universities typically rely on external platforms such as PC² or Codeforces, which are not tightly integrated with local authentication, course workflows, or internal infrastructure. AuraC² aims to provide an in-house contest control system tailored to Yarmouk University.

AuraC² is a web-based platform that allows teams to register and log in using JWT-based authentication, join contests, view problems, and submit solutions. Each submission is pushed to a RabbitMQ queue, processed by a worker that communicates with the Judge0 API, and then returned through a result queue, after which the result is stored and delivered to the user via a notification service. This assignment marks the transition from pure system analysis to real implementation of the architecture.

The main focus of this report is:
(1) System architecture.
(2) Submission sequence flow.
(3) Database and class diagrams.
(4) Core implementation progress (30–40%).

2. System Architecture Design

2.1 Overall Architecture

AuraC² follows a layered architecture similar to modern web systems used in online judges.

The UI layer consists of a web client implemented using React or Angular. It communicates with the backend through a REST API over HTTPS.

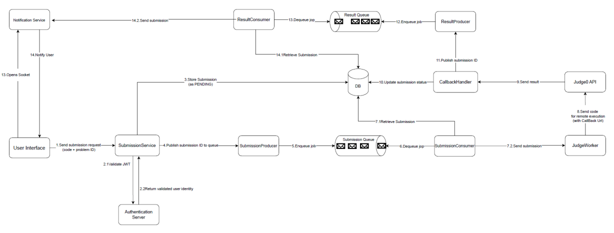
The backend layer is built with Spring Boot and uses Spring Security with JWT for authentication and authorization. The backend exposes controllers that receive HTTP requests, services that contain the business logic, and DTOs (Data Transfer Objects) that decouple API payloads from database entities.

For asynchronous processing, AuraC² uses RabbitMQ as a message broker. Submissions are published to a submissionQueue, from where a SubmissionConsumer retrieves them. Judging results are delivered through a resultQueue, consumed by a ResultConsumer, which then updates the database and triggers notifications.

Judging is performed by a Judge Worker component. This worker sends the source code and execution parameters to the Judge0 API, waits for the callback, and then forwards the final status to the backend through a dedicated callback endpoint. A CallbackHandler is responsible for updating the submission record.

The persistence layer uses a relational database (MySQL or PostgreSQL) accessed through JPA/Hibernate. Entities such as User, Contest, Problem, TestCase, Submission, EmailVerificationToken, and RefreshToken are mapped to tables and persisted transparently.

2.2 Submission Flow



The complete submission flow can be described as follows. A team sends a POST request to /api/submissions with the code, language, and problem identifier. The backend validates the JWT access token and checks that the user is allowed to participate in the contest. A new Submission entity is created with status PENDING and saved to the database.

The backend then publishes a message to the submissionQueue containing the submission's identifier and any required metadata (language, time limit, memory limit, and so on). The SubmissionConsumer, running as part of the Judge Worker process or a separate microservice, consumes this message and prepares a request to Judge0. The worker sends the code and input specification to Judge0 and registers a callback URL.

When Judge0 finishes execution, it calls the /api/callback endpoint with the final status and execution details (ACCEPTED, WRONG_ANSWER, RUNTIME_ERROR, TIME_LIMIT_EXCEEDED, etc.). The CallbackHandler updates the Submission entity in the database to reflect the new status and stores execution metadata if required. Finally, a message is pushed to the resultQueue. The ResultConsumer listens for these messages and notifies the UI (e.g., via WebSocket or a polling API), so that the team sees the result in real time.

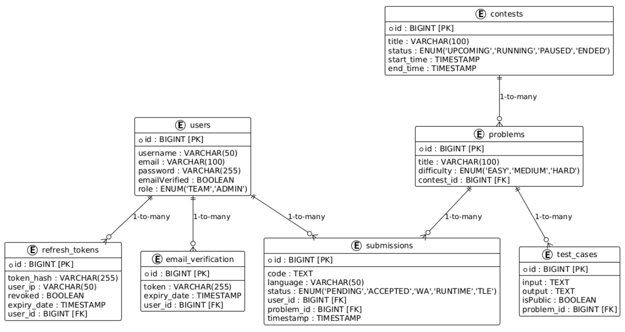
2.3 Use-Case Realization: Submit Solution

Primary actor: team user.
Goal: send code to the judge and receive the result.

Step 1. The user invokes POST /api/submissions with code, language, and problemId.
Step 2. SubmissionService validates the JWT and verifies contest access.
Step 3. The system creates a Submission entity with status PENDING and saves it to the database.
Step 4. The submission identifier and metadata are published to the submissionQueue (RabbitMQ).
Step 5. SubmissionConsumer reads from submissionQueue and forwards the job to JudgeWorker.
Step 6. JudgeWorker calls Judge0 API to create a submission on Judge0.
Step 7. Judge0 executes the code and sends a callback to /api/callback when finished.
Step 8. CallbackHandler updates the Submission in the database with the final status and output.
Step 9. ResultProducer publishes a message to resultQueue or notifies the notification service.
Step 10. The UI informs the team about the result: Accepted, Wrong Answer, Runtime Error, etc.

3. Database Design

3.1 Entity-Relationship Overview



The database model follows a typical competitive programming system. Users participate in contests, contests contain problems, problems have test cases, and submissions connect users to problems in specific contests.

The main entities are:

Users: represent participants and administrators. Fields include id, username, email, password, enabled flag, and role (STUDENT, ADMIN).

Contests: define a contest with title, description, status (UPCOMING, RUNNING, PAUSED, ENDED), start_time, and end_time.

Problems: belong to a given contest, with fields such as title, statement, time_limit, memory_limit, and difficulty (EASY, MEDIUM, HARD).

TestCases: connected to a single problem via a foreign key problem_id, storing input, expected_output, and visibility flags.

Submissions: link a user to a problem and an optional contest context. A submission contains code, language, status (e.g., PENDING, ACCEPTED, WRONG_ANSWER, RUNTIME_ERROR, TIME_LIMIT_EXCEEDED), execution time, and timestamp.

RefreshToken: stores tokens used for JWT rotation and revocation, referencing a user.

EmailVerificationToken: stores tokens used for account activation and email verification.

3.2 Design Justification

Layered separation.

Using controllers, services, entities, and DTOs aligns with Clean Architecture principles. It decouples the HTTP layer from business logic and persistence. This separation avoids tight coupling and allows the UI, backend logic, and database schema to evolve independently.

Single-responsibility entities.

Each entity maps to a clear domain concept: User, Contest, Problem, TestCase, Submission, RefreshToken, EmailVerificationToken, etc. This follows Domain-Driven Design and makes testing, refactoring, and future extensions (such as standings or analytics) easier. Combining too many responsibilities into a single table (for example, storing all tokens inside User) would complicate queries and break separation of concerns.

Enum-based validation.

Critical attributes such as ContestStatus, Role, Difficulty, SubmissionStatus, and TokenType use enums in both the code and database mapping. This enforces data integrity at the database level and prevents invalid states such as "endeddd" or "unknownRole". Even if an API is misused, enum-based constraints protect the data.

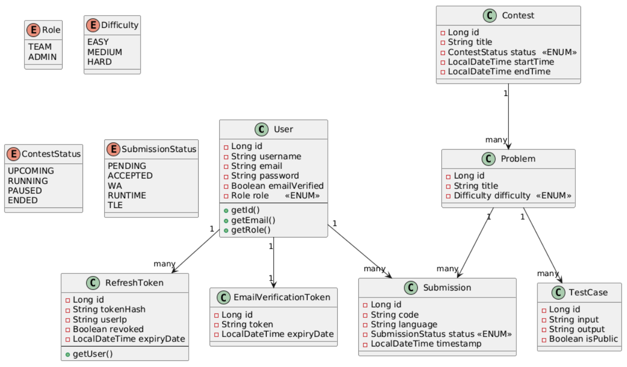
Submission as the core link.

The Submission entity is the central link between User, Problem, and Contest. It allows multiple attempts per problem, historical tracking, and ranking logic similar to Codeforces or LeetCode. Storing results directly in the Problem table would not support multiple attempts and would make ranking impossible.

Security token design.

EmailVerificationToken and RefreshToken are modeled as separate entities instead of embedding their data inside User. This design supports JWT rotation, token revocation, account activation links, and potentially multiple concurrent sessions per user. It is also flexible enough to support future OAuth2 integration.

4. Class and Module Design



4.1 Class Structure

The class diagram mirrors the database entities and the layered architecture:

– User, Contest, Problem, TestCase, Submission, RefreshToken, EmailVerificationToken as JPA entities.
– DTO classes such as UserDto, ContestDto, ProblemDto, SubmissionRequestDto, and SubmissionResponseDto for API communication.
– Service classes (UserService, ContestService, ProblemService, SubmissionService) encapsulating business logic.
– Controllers (AuthController, ContestController, ProblemController, SubmissionController) exposing REST endpoints.
– Security configuration classes for JWT filters, authentication entry points, and access control.

4.2 Submission Module

The submission module is structured as follows.

SubmissionController handles HTTP endpoints for creating and querying submissions (POST/GET). It validates input and delegates to SubmissionService.

SubmissionService contains the core logic for creating submissions, checking permissions, and saving PENDING submissions. It also publishes messages to the queue.

SubmissionProducer publishes submission messages to submissionQueue in RabbitMQ.

SubmissionConsumer listens to submissionQueue, passes jobs to JudgeWorker, and handles error cases.

JudgeWorker interacts with Judge0: it creates submissions on Judge0, handles API requests, and registers the callback URL.

CallbackHandler processes calls from Judge0, updates submissions in the database, and triggers subsequent notifications.

ResultProducer publishes messages about finished submissions to resultQueue.

ResultConsumer listens to resultQueue and notifies the UI, for example using WebSocket or server-sent events.

5. Initial Implementation Status (30–40%)

The following features are implemented and tested:

JWT login and registration.

Users can register and log in using Spring Security with JWT. The endpoints are tested via Postman.

Email verification.

Upon registration, an email containing a verification link is sent using Gmail SMTP. The user account becomes active after clicking the link.

Refresh token in cookie.

Refresh tokens are stored in a secure, httpOnly cookie. This allows safe token rotation without exposing the refresh token to JavaScript.

Contest lifecycle.

Contests can be created, started, paused, and ended through the API. The status field is updated accordingly.

Problem management.

CRUD operations for problems are implemented. Problems are linked to contests and store time and memory limits, difficulty, and statement.

Test cases.

TestCases are linked to their Problem via a foreign key. Test cases can be created, updated, and listed for each problem.

Submission flow (50% complete).

The API endpoints and queue configuration are in place. Submissions are stored as PENDING and published to the queue. The remaining work is the full Judge0 integration and final notification mechanisms.

6. Challenges and Solutions

Handling asynchronous code execution.

Problem: Judge0 responses are delayed and cannot be handled synchronously in a single HTTP request/response cycle.

Solution: Use RabbitMQ queues and a worker-based design instead of direct synchronous HTTP calls. This allows non-blocking request handling and scales horizontally by adding more workers.

Mapping test cases to problems.

Problem: A single problem may have many test cases, which can complicate code paths and queries if not modeled correctly.

Solution: Use a one-to-many relationship between Problem and TestCase, with a clear problem_id foreign key. This design simplifies both storage and retrieval of test cases.

Identifying which test case failed.

Problem: A submission traditionally only stores its final status (ACCEPTED, WRONG_ANSWER, etc.), which does not show which specific test case failed.

Solution: Plan for a SubmissionTestCaseResult table that links a submission to each test case and stores the per-test result. This enables detailed feedback and transparent grading, similar to modern competitive programming systems.

Connecting users to contest results.

Problem: One user can participate in multiple contests and submit multiple solutions for many problems. Contest standings rely on this relationship.

Solution: Use the Submission entity as the central connection between user, problem, and contest, including timestamps. This design supports leaderboards, ranking, and historical analysis.

Avoiding entity leakage in API responses.

Problem: Returning JPA entities directly from controllers couples the API to the internal database schema and may expose sensitive fields.

Solution: Introduce a DTO layer for request and response objects. Controllers convert between DTOs and entities, making the API stable, secure, and independent of internal persistence details.

7. Next Steps (Assignment #4)

The following features are planned for the next assignment:

JudgeWorker–Judge0 integration.

Implement the full HTTP client and mapping for Judge0, including error handling and retries where necessary.

WebSocket notifications.

Add real-time UI updates using WebSocket or similar technology so that teams see submission results instantly.

Submission status tracking.

Introduce a dedicated status table or extend the Submission entity to track state transitions in more detail.

Contest dashboard and statistics.

Build a dashboard for administrators and teams showing contest standings, number of submissions, problem statistics, and per-team performance.

Frontend integration.

Integrate the React/Angular frontend with all backend endpoints, including authentication, contest management, problem browsing, submission visualization, and live updates.