A Report on the Development of a Rubik's Cube Timer Application

Santiago Andrés Benavides Coral
20232020036
Engineering faculty
Advanced Programming
Season 2024-III
Universidad Distrital Francisco José de Caldas

CONTENTS

LIST OF FIGURES

A Report on the Development of a Rubik's Cube Timer Application

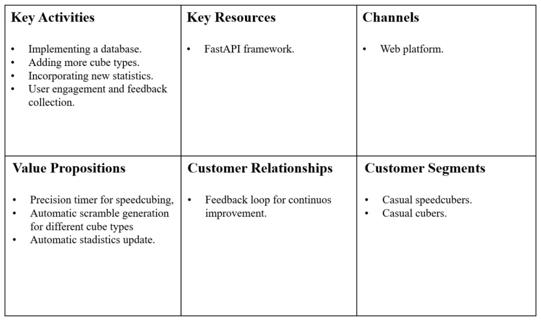
***Abstract*—This report documents the development of a Rubik's Cube Timer application, designed to help users time and record their Rubik's Cube solving sessions. The application features user authentication, solve timing, and statistics tracking. The backend is built using FastAPI, with data persistence in a JSON file, while the frontend is a simple web interface. This report covers the introduction, user stories, functional and non-functional requirements, UML diagrams, conceptual design, system architecture, backend service design, SOLID principles analysis, and conclusions.**

I. Introduction

The Rubik's Cube Timer application is a web-based tool designed for speedcubers to time their solves, track their progress, and analyze their performance. The application provides a user-friendly interface for timing solves, along with features such as user authentication, solve history, and statistics. The backend is implemented using FastAPI, a modern Python web framework, and the frontend is built with HTML, CSS, and JavaScript. This report outlines the development process, including user stories, requirements, design, and implementation details.

II. Business Model Canvas

The following Business Model Canvas outlines the key aspects of the Rubik's Cube Timer application:



Key Activities	Key Resources	Channels
• Implementing a database. • Adding more cube types. • Incorporating new statistics. • User engagement and feedback collection.	• FastAPI framework.	• Web platform.
Value Propositions	**Customer Relationships**	**Customer Segments**
• Precision timer for speedcubing, • Automatic scramble generation for different cube types • Automatic stadistics update.	• Feedback loop for continuos improvement.	• Casual speedcubers. • Casual cubers.

Fig. 1. Business Model Canvas for the Rubik's Cube Timer Application

III. User Stories

The following user stories describe the functionality of the Rubik's Cube Timer application from the perspective of different users:

- **User Registration**
  As a new user, I want to create an account so that I can log in and track my progress over time.
- **Cube Selection**
  As a casual cuber, I want to choose the type of cube so that I can time my solves for different types of cubes.
- **Scramble Generation**
  As a speedcuber, I want to receive a random scramble for my selected cube so that I can practice under standard competition conditions.
- **Start and Stop Timer**
  As a user, I want to start and stop the timer so that I can record the time it takes me to solve the cube.
- **Automatic Statistics Update**
  As a user, I want my statistics to update automatically after each solve so that I can track my performance easily.
- **View Past Performance**
  As a user, I want to see a history of my past solves so that I can measure my improvement over time.
- **Log Out**
  As a user, I want to log out of the app so that my session is securely closed.
- **Error Handling**
  As a user, I want the app to notify me if I make a mistake (e.g., enter the application incorrectly) so that I can correct it quickly and continue using the app.

IV. Functional and Non-Functional Requirements

A. Functional Requirements

- Users can register and log in to the application.
- Authenticated users can time their Rubik's Cube solves.
- Users can view their solve history and statistics.
- The application supports different cube types (e.g., 2x2, 3x3, 4x4).
- Users can generate scrambles for their solves.

B. Non-Functional Requirements

- The application must be secure, with user passwords hashed and stored safely.
- The backend must be scalable and able to handle multiple concurrent users.
- The application should have a responsive and intuitive user interface.
- Data persistence must be ensured, with solves and user data stored reliably.

V. UML Diagrams

A. Sequence Diagram

The following sequence diagram illustrates the flow of user registration and solve creation:

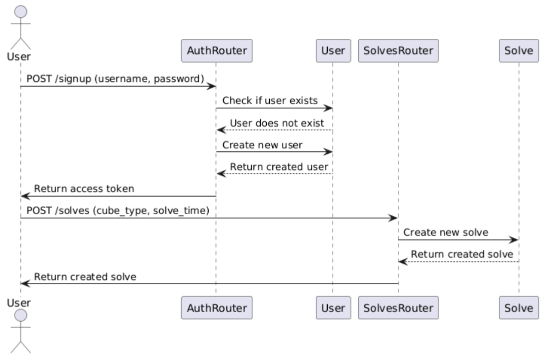


Fig. 2. Sequence Diagram for User Registration and Solve Creation

As shown in Figure 2, the user registration and solve creation process involves several steps.

B. Activity Diagram

The following activity diagram illustrates the flow of user interactions with the timer application:

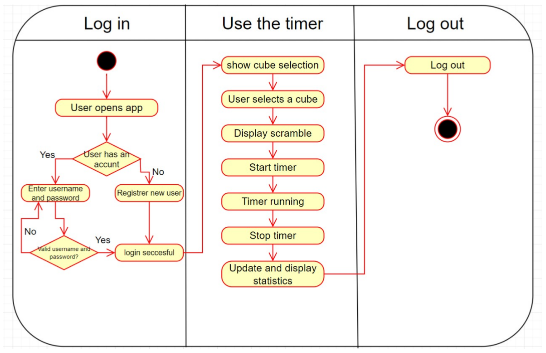


Fig. 3. Activity Diagram for the Timer Application

As shown in Figure 3, the user navigates through different actions, from logging in to using the timer and logging out.

C. Deployment Diagram

The following deployment diagram illustrates the system architecture of the Rubik's Cube Timer application:

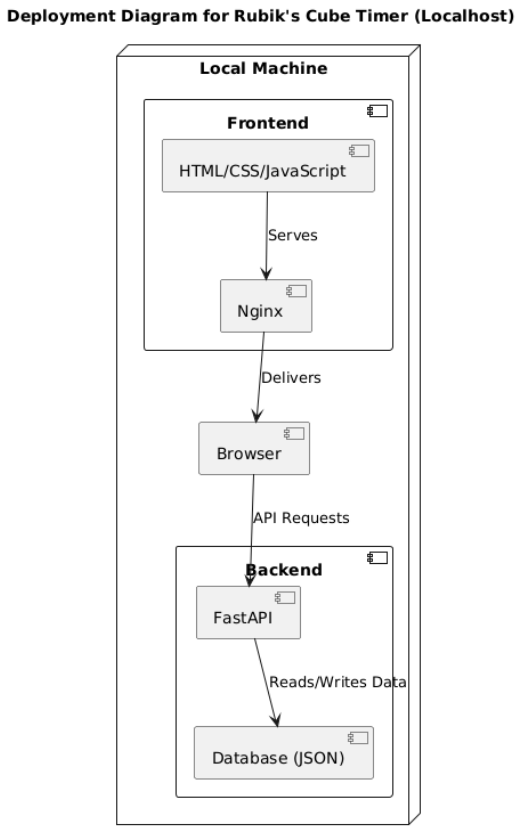


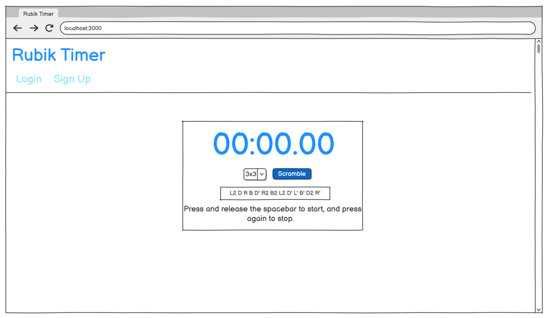
Fig. 4. Deployment Diagram for Rubik's Cube Timer

As shown in Figure 4, the system is divided into frontend and backend components, using Nginx for serving the frontend and FastAPI for backend operations.

VI. Conceptual Design

A. Mockups

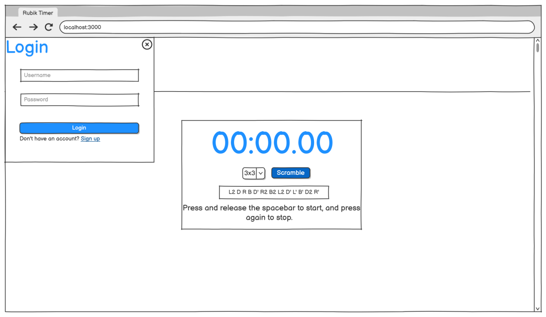
The following mockups illustrate the user interface design of the Rubik's Cube Timer application:



L2 D R B D' R2 B2 L2 D' L' B' D2 R'

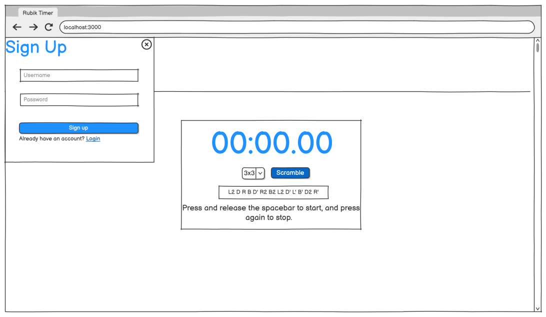
Press and release the spacebar to start, and press again to stop.

Fig. 5. Mockup 1: Home Screen



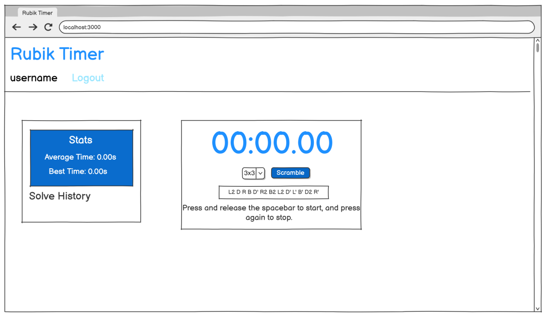
Press and release the spacebar to start, and press again to stop.

Fig. 6. Mockup 2: Login Screen



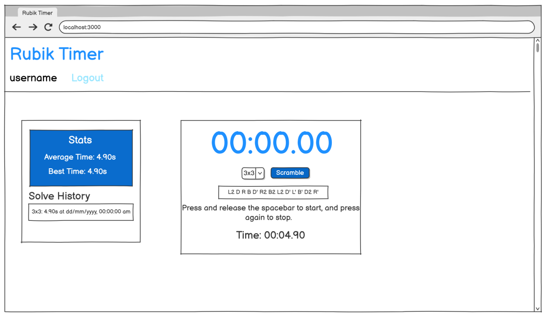
Press and release the spacebar to start, and press again to stop.

Fig. 7. Mockup 3: Sign Up Screen



Press and release the spacebar to start, and press again to stop.

Fig. 8. Mockup 4: User Screen



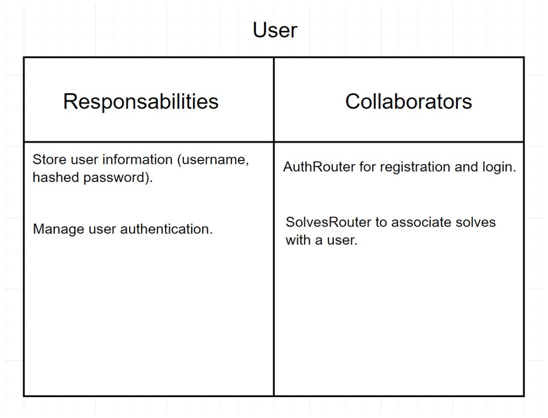
Press and release the spacebar to start, and press again to stop.

Time: 00:04.90

Fig. 9. Mockup 5: Solve History and Statistics Screen

B. *CRC Cards*

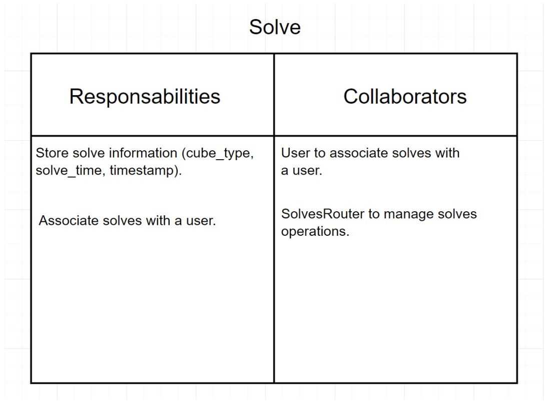
The following CRC (Class-Responsibility-Collaboration) cards describe the key components of the Rubik's Cube Timer application:



User

Responsabilities	Collaborators
Store user information (username, hashed password).	AuthRouter for registration and login.
Manage user authentication.	SolvesRouter to associate solves with a user.

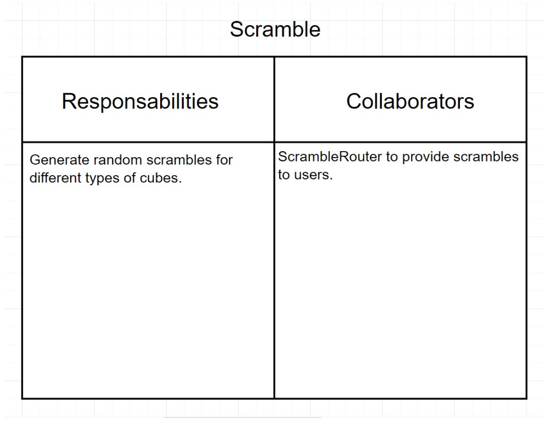
Fig. 10. CRC Card for the User Class



Solve

Responsabilities	Collaborators
Store solve information (cube_type, solve_time, timestamp).	User to associate solves with a user.
Associate solves with a user.	SolvesRouter to manage solves operations.

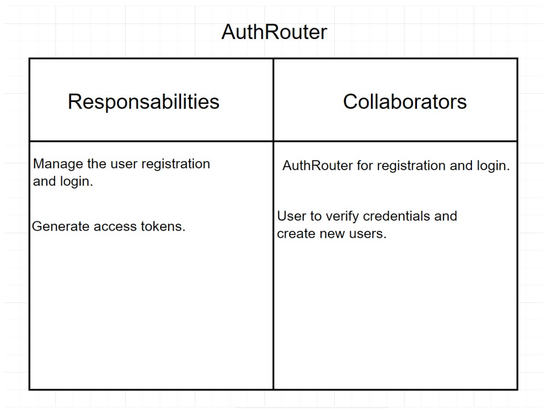
Fig. 11. CRC Card for the Solve Class



Scramble

Responsabilities	Collaborators
Generate random scrambles for different types of cubes.	ScrambleRouter to provide scrambles to users.

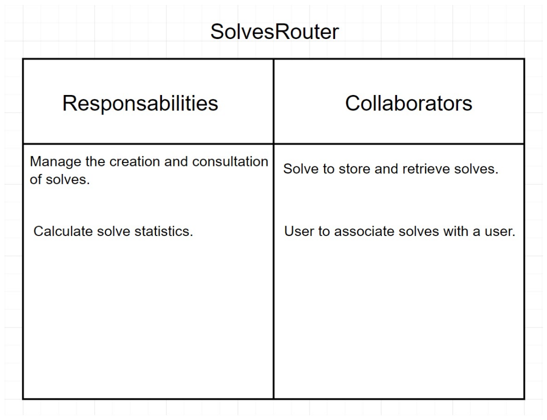
Fig. 12. CRC Card for the Scramble Class



AuthRouter

Responsabilities	Collaborators
Manage the user registration and login. Generate access tokens.	AuthRouter for registration and login. User to verify credentials and create new users.

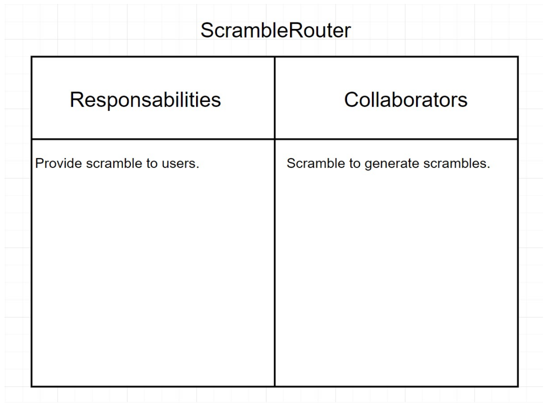
Fig. 13. CRC Card for the AuthRouter Class



SolvesRouter

Responsabilities	Collaborators
Manage the creation and consultation of solves. Calculate solve statistics.	Solve to store and retrieve solves. User to associate solves with a user.

Fig. 14. CRC Card for the SolvesRouter Class



ScrambleRouter

Responsabilities	Collaborators
Provide scramble to users.	Scramble to generate scrambles.

Fig. 15. CRC Card for the ScrambleRouter Class

VII. System Architecture

The following class diagram illustrates the structure of the Rubik's Cube Timer application:

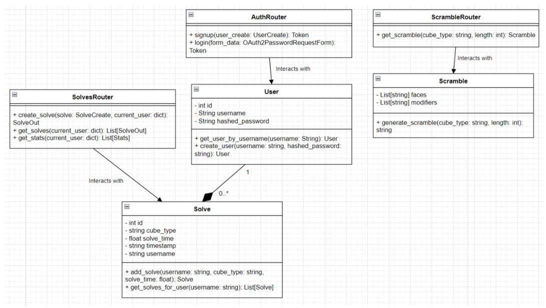

AuthRouter
+ signup(user_create: UserCreate): Token
+ login(form_data: OAuth2PasswordRequestForm): Token
ScrambleRouter
+ get_scramble(cube_type: string, length: int): Scramble
Interacts with
Interacts with
Scramble
- List[string] faces
- List[string] modifiers
+ generate_scramble(cube_type: string, length: int): string
SolvesRouter
+ create_solve(solve: SolveCreate, current_user: dict): SolveOut
+ get_solves(current_user: dict): List[SolveOut]
+ get_stats(current_user: dict): List[Stats]
User
- int id
- String username
- String hashed_password
+ get_user_by_username(username: String): User
+ create_user(username: string, hashed_password: string): User
Interacts with
1
0..*
Solve
- int id
- string cube_type
- float solve_time
- string timestamp
- string username
+ add_solve(username: string, cube_type: string, solve_time: float): Solve
+ get_solves_for_user(username: string): List[Solve]

Fig. 16. Class Diagram for Rubik's Cube Timer

As shown in Figure 16, the system consists of multiple classes and routers interacting to handle authentication, scrambles, and solving records.

VIII. Backend Design: Web Services

The backend of the Rubik's Cube Timer application is designed as a RESTful API using FastAPI. The main components of the backend include:

- **Authentication Service**: Handles user registration, login, and JWT token generation.
- **Solve Service**: Manages the creation, retrieval, and statistics of solves.
- **Scramble Service**: Generates scrambles for different cube types.
- **Database Layer**: Persists user and solve data in a JSON file.

The backend services are designed to be modular and follow the principles of clean architecture, ensuring separation of concerns and ease of maintenance.

IX. Analysis of SOLID Principles Implementation

The implementation of the Rubik's Cube Timer application adheres to the SOLID principles as follows:

- **Single Responsibility Principle (SRP)**: Each class and function has a single responsibility. For example, the ‘db.py‘ file handles database operations, while ‘auth.py‘ manages authentication.
- **Open/Closed Principle (OCP)**: The system is open for extension but closed for modification. New features, such as additional cube types, can be added without modifying existing code.
- **Liskov Substitution Principle (LSP)**: The application does not use inheritance extensively, but where it is used, derived classes can substitute base classes without altering the behavior.
- **Interface Segregation Principle (ISP)**: The backend services are designed with specific interfaces, ensuring that clients are not forced to depend on methods they do not use.
- **Dependency Inversion Principle (DIP)**: High-level modules (e.g., routers) depend on abstractions (e.g., service interfaces) rather than concrete implementations.

X. Conclusions

The Rubik's Cube Timer application successfully addresses the needs of speedcubers by providing a reliable and user-friendly tool for timing solves and tracking progress. The application's backend, built with FastAPI, is modular, scalable, and adheres to SOLID principles, ensuring maintainability and extensibility. The frontend offers an intuitive interface for users to interact with the application. Future work could include adding support for more cube types, improving the user interface, and integrating with external databases for enhanced data persistence.

References

[1] FastAPI Documentation. Available: https://fastapi.tiangolo.com/