Travlendar+ project YOUR NAMES

Requirement Analysis and Specification Document

Deliverable:	RASD
Title:	Requirement Analysis and Verification Document
Authors:	YOUR NAMES
Version:	1.0
Date:	31-January-2016
Download page:	LINK TO YOUR REPOSITORY
Copyright:	Copyright © 2017, YOUR NAMES – All rights reserved

Contents

Table of Contents 3

List of Figures 5

List of Tables 5

1 Introduction 6
- 1.1 Purpose 6
- 1.2 Scope 6
- 1.3 Definitions, Acronyms, Abbreviations 6
- 1.4 Revision History 6
- 1.5 Reference Documents 6
- 1.6 Document Structure 6

2 Architectural Design 7
- 2.1 Overview 7
- 2.2 Component View 8
- 2.3 Deployment View 9
- 2.4 Runtime View 11
- 2.5 Component Interfaces 11
  - 2.5.1 IUserAuthenticator 13
  - 2.5.2 IProfileManager 13
  - 2.5.3 ITournamentManager 13
  - 2.5.4 IBattleManager 14
  - 2.5.5 ISessionManager 14
  - 2.5.6 INotification 14
  - 2.5.7 Model Interfaces 15
  - 2.5.8 IBattleTeamOrganiser 16
  - 2.5.9 ISubmissionManager 16
  - 2.5.10 ISubmissionListener 16
  - 2.5.11 ISubmissionRetriever 16
  - 2.5.12 IScoringHandler 16
  - 2.5.13 IBattleRepoHandler 16
  - 2.5.14 IGithubAPI 16
  - 2.5.15 IFileManagement 17
  - 2.5.16 API Endpoints 17
- 2.6 Selected Architectural Styles and Patterns 29
  - 2.6.1 3-Tier Architecture 29
  - 2.6.2 Layered Architecture 30
  - 2.6.3 Facade 30
  - 2.6.4 Mediators 31
- 2.7 Other Design Decisions 31
  - 2.7.1 Load Balancing, Firewall and Replication 31
  - 2.7.2 Sandbox Paradigm 31

3 User Interface Design 34

4 Requirements Traceability 35

5 Implementation, Integration, and Test Plan 36

6 Effort Spent **37**

References **38**

List of Figures

1 High-Level Architecture of the System 7
2 Component Diagram 8
3 Deployment Diagram 9
4 Class Diagram (Based on RASD) 12
5 Submission Scoring 33

List of Tables

1 Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, Acronyms, Abbreviations

1.4 Revision History

1.5 Reference Documents

1.6 Document Structure

2 Architectural Design

2.1 Overview

The high-level architectural diagram provided below offers a conceptual overview of the CodeKataBattle (CKB) platform's infrastructure. It delineates the system's division into three primary layers: Presentation, Application, and Data.

The Presentation Layer captures the user interaction with the system via a standard web browser, illustrating the entry point for both educators and students.

The Application Layer is the system's backbone, housing the business logic and core functionalities, including load balancing, application servers, and interfaces for external services such as the GitHub API, Static Analysis Tool API, Email Service, and Notification Service. A dedicated firewall protects this layer, ensuring secure data transactions. It is mainly responsible for handling requests from clients and presentation layer. This layer communicates with the Data Layer, to store and process the data.

The Data Layer is structured to manage persistent data and comprises the Database Management System (DBMS), which supports sharded databases for scalability, and a File Storage system that accommodates various data types, including educator uploads and code submissions. This layer is mainly responsible for data storage and access by querying.

Each component is strategically placed to optimize performance and maintainability, reinforcing the platform's robustness and reliability. The details are discussed in the following sections.

Here, it is important to explain the reasons that led to choice of 3-Tier Architecture. Firstly, this kind of separation of logic helps to improve horizontal scalability. Each layer can be developed and maintained by different software teams. Also, different technologies can be adopted for presentation, application and data layers without affecting each other.

On the other hand, another option can be Microservice Architecture, which is more modular then the 3-layered architecture.It provides higher degree of separation between each part of your application, which leads to even more flexibility and agility than you'd get from a three-tier app. However, as a trade-off, a Microservice Architecture includes more components to deploy and track, which makes developing and maintaining application more challenging because of the higher complexity. In this scenario, even orchestrators and service meshes can be needed. When we think about the CodeKataBattle platform specifically, the scalability of 3-layered architecture is well enough with several servers. A Microservice Architecture might not make sense when a large cluster which maximizes scalability and resilience is not used. Eventhough Microservice Architecture would be better option to scale up and down in a granular way, because of such complexity , it will be excessive for the CodeKataBattle.

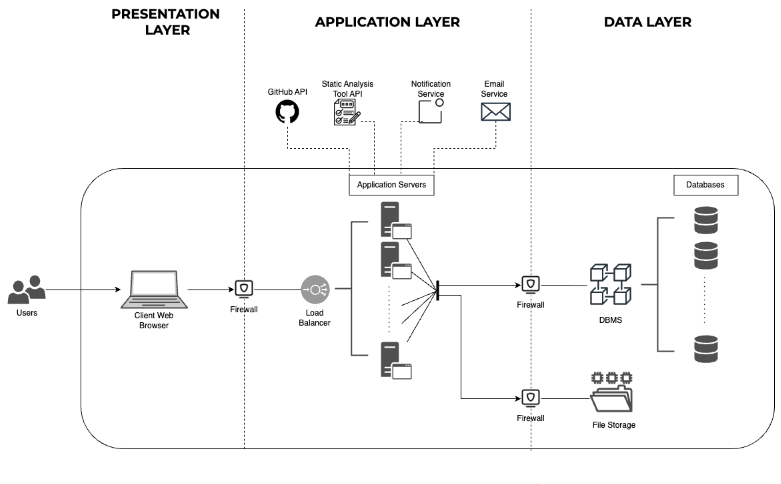

Figure 1: High-Level Architecture of the System

2.2 Component View

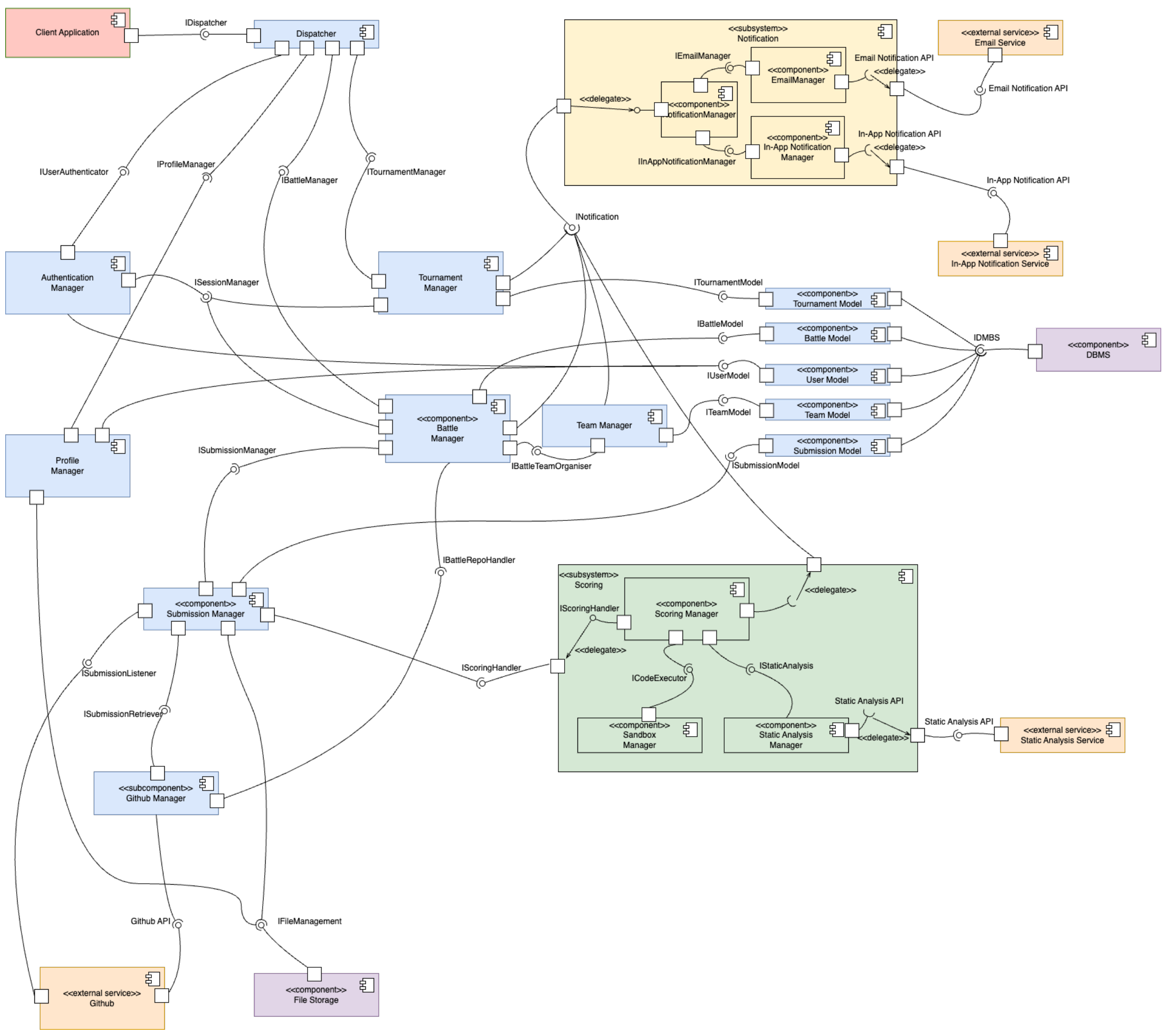

Figure 2: Component Diagram

2.3 Deployment View

Our architecture mainly consists of 3 parts:

- A Static Web Server
- Application Server
- Database Server

Users can interact with website via a browser and a device that has browser support such as a computer, a mobile phone etc. Content Delivery Network is used for the web server which behaves as an entry point to users. It hosts the static and dynamic web content, such as .html, .css, .js, and image files, to users. Using Content Delivery Network has some advantages in terms of performance, reliability and security. CDNs speed up content delivery by decreasing the distance between where content is stored and where it needs to go, reducing file sizes to increase load speed, optimizing server infrastructure to respond to user requests more quickly. Also if a server, a data center, or an entire region of data centers goes down, CDNs can still deliver content from other servers in the network. Moreover, it is also very useful from the security perspective. With their many servers, CDNs are better able to absorb large amounts of traffic, even unnatural traffic spikes from a DDoS attack, than a single origin server.

Then we have our Application Server which is hosted on the cloud with 2-4 instances.

At the Data Layer, we have database server which includes database and DBMS with a firewall.

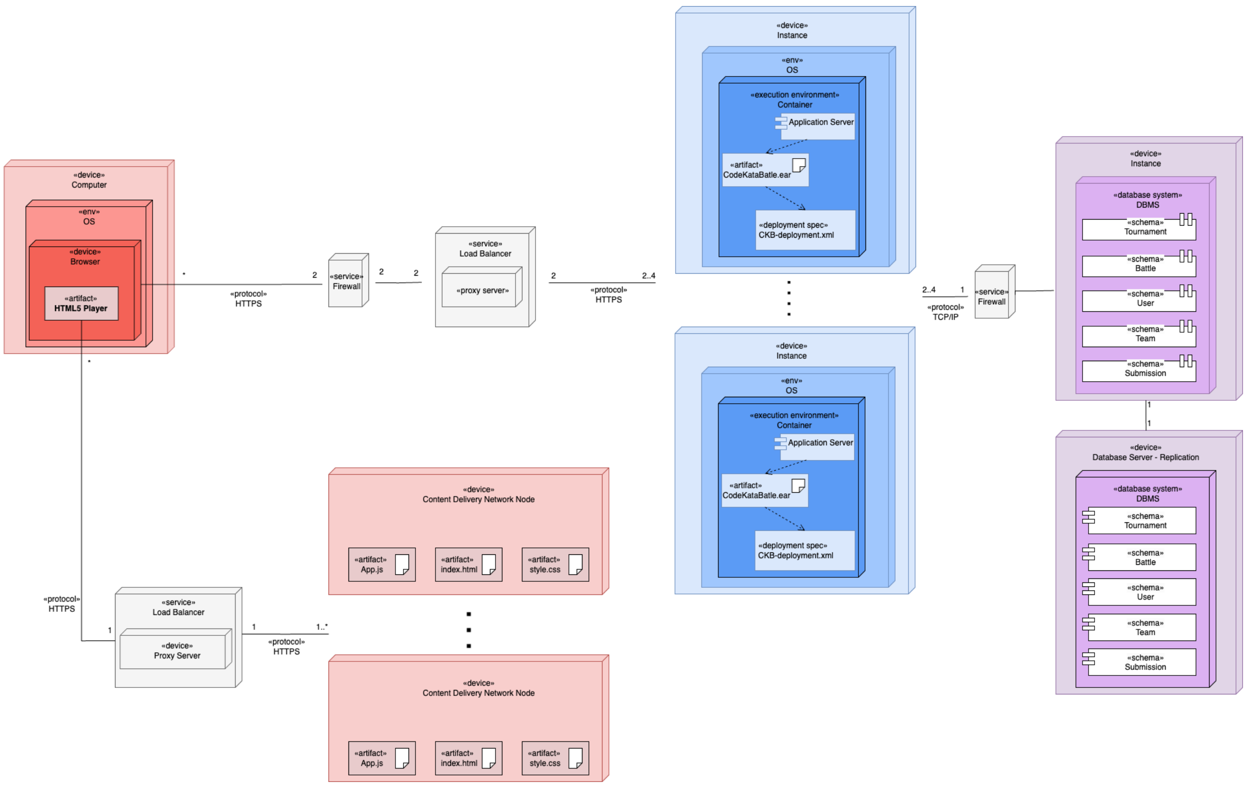

Figure 3: Deployment Diagram

The deployment diagram offers a more detailed view over the hardware and software resources of the application:

User Device: User device can be any device that supports a web browser.

Static Web Content: The static web content of CodeKata Battle is hosted by Content Delivery Network with a Load Balancer distributing traffic and workload. The nodes in CDN can be scaled to very large numbers because it needs low computation power and memory. Geographically distributed nodes can be helpful to provide web interface with great performance in terms of speed. Briefly, The this content is static and all of its code is run on the client's machine,by browser, so there is no need for any logic to be implemented on the CDN side.

Application Server: All the business logic is handled in the Application Server which is hosted on the cloud with minimum 2 and maximum 4 instances. Obviously, these numbers can change in time but ,for an initial design, using 2-4 server instances would be adequate. Also 2 load balancer is used to distributing workload among application server instances. This is the bottleneck of the application so we decided to use 2 of them.

Distributing incoming requests among multiple servers hosted on the cloud can helps us to fulfill some technical constraints:

Reliability: Cloud providers often offer high reliability through redundant resources and infrastructure. If one server fails, others can take over, minimizing downtime. Regular backups and disaster recovery options further enhance reliability.

Availability: High availability is a key feature of these kind of hosting. Multiple instances in the cloud use multiple data centers around the world, ensuring that your services remain accessible even during local outages or disruptions.

Security: Cloud providers invest heavily in security measures, including physical security of data centers, network security, and data encryption. Monitoring tools and firewall helps to sustain a secure application server.

Scalability: Increasing or decreasing the size and amount of the instances can help to deal with changing request loads that servers have to respond to, also with the help of load balancing. Because of the changes in traffic or workload, we would need different computation and memory capacity. You can easily scale your server resources up or down based on demand, ensuring optimal performance without overpaying for unused capacity.

Maintainability: Cloud providers handle hardware maintenance, updates, and patches, allowing us to focus on core business and application development. Also logging and monitoring tools generally works well with this kind of hosting other than in-house hosting or custom solutions we can develop.

Portability: Cloud environments support portability and interoperability. You can move applications and data across different cloud environments or providers with relative ease, avoiding vendor lock-in and allowing for flexibility in deployment choices.

Database: This instance contains database with necessary schemas and database managements system. It is used with a replication It is also has a replicated version of itself, because of our **reliability** and **availability** constraints. By replicating data across multiple nodes or locations, the system can ensure high **availability**. If one node fails, the others can continue to operate, minimizing downtime and ensuring continuous access to data. Also, in the event of a major failure or disaster affecting the primary data center, having replicas ensures that data is not lost and can be quickly recovered.

Firewalls: Firewall services act as a security gatekeeper for a system's business and data layers, screening incoming connections. They enhance security by enforcing rules that either permit or block traffic, safeguarding the system from illicit access or harmful attacks.

Load Balancers: A load balancer is employed to evenly distribute incoming traffic across various instances of an application. This strategy optimizes the use of resources, boosts performance, and maintains high availability. By doing so, the load balancer aids in managing a significant influx of requests, preventing the application from being overwhelmed or suffering downtime, and contributes to overall stability.

2.4 Runtime View

2.5 Component Interfaces

In this section, we explain the interfaces and provided methods and returned objects. Also a class diagram is provided to show the dependencies and relations between the interfaces. Also at the end of the section endpoints provided by the system is listed and explained.
The Class Diagram is the same as Domain Level Class Diagram in RASD except some additional schmes to demonstrate the data structures used in interfaces.

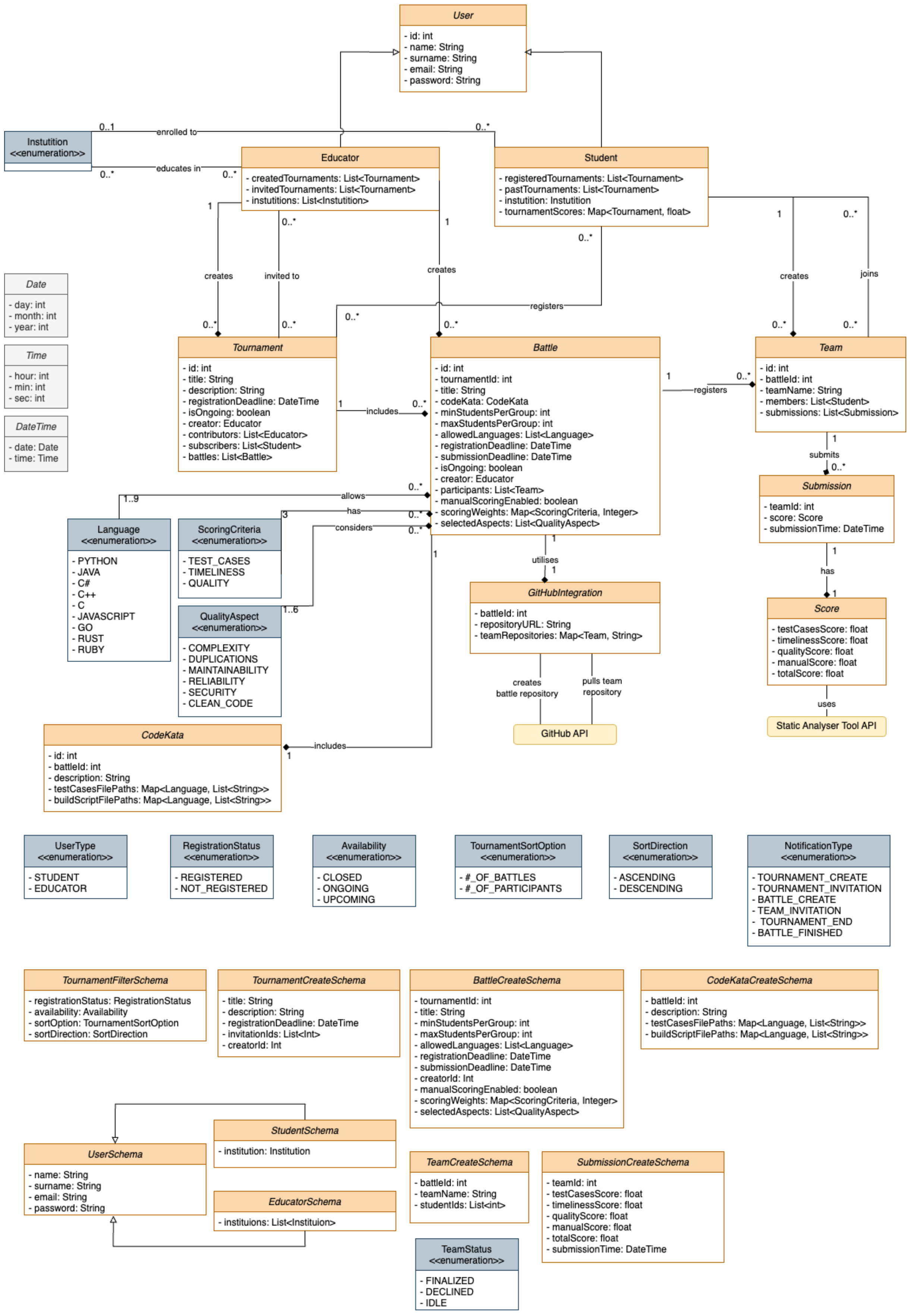

Figure 4: Class Diagram (Based on RASD)

2.5.1 IUserAuthenticator

This Interface contains endpoints related to authentication of users. Endpoints explained in a detailed way at the end of the section. Just signatures are available here.

- Register(email: String, password: String, name: String, surname: String, institutionInfo: List<Institution>, userType: UserType) -> HTTP Response
- Login(email: String, password: String) -> HTTP Response
- Logout()-> HTTP Response
- VerifyEmail(email:String) -> HTTP Response

2.5.2 IProfileManager

This Interface contains endpoints related to profiles of users. Endpoints explained in a detailed way at the end of the section. Just signatures are available here.

- GetMyProfile() -> HTTP Response
- GetProfile(userId: int) -> HTTP Response
- EditProfile(email: String, password: String, newPassword: String, name: String, surname: String, institutionInfo: List<Institution>) -> HTTP Response
- GetEducators()-> HTTP Response

2.5.3 ITournamentManager

This Interface contains endpoints related to Tournament. Endpoints explained in a detailed way at the end of the section. Just signatures are available here.

- GetTournaments(registrationStatus: RegistrationStatus, availability: Availability, sortOption: TournamentSortOption, sortDirection: SortDirection) -> HTTP Response
- GetTournamentInfo(tournamentId: int) -> HTTP Status
- RegisterTournament(tournamentId: int) -> HTTP Status
- GetTournamentBattles(tournamentId: int, minGroupSize: int, maxGroupSize: int, startDate: DateTime, endDate: DateTime, registrationStatus: RegistrationStatus, institution: Institution, languages: List<Language>, educatorIds: List<int>, searchText: String) -> HTTP Response
- GetTournamentLeaderboard(tournamentId: int) -> HTTP Response
- ExportTournamentLeaderboard(tournamentId: int) -> HTTP Response
- CreateTournament(title: String, description: String, registrationDeadline: DateTime, invitedEducatorIds: List<int>) -> HTTP Response
- AnswerTournamentInvitation(tournamentId: int, isAccepted: boolean, educatorId: int) -> HTTP Response
- EndTournament(tournamentId: int) -> HTTP Response

2.5.4 IBattleManager

This Interface contains endpoints related to Battle. Endpoints explained in a detailed way at the end of the section. Just signatures are available here.

- GetBattleInfo(battleId: int) -> HTTP Response
- GetBattleRankings(battleId: int) -> HTTP Response
- RegisterBattle(battleId: int, teamName: String, teamInvitations: List<int>) -> HTTP Response
- AnswerBattleInvitation(teamId: int, isAccepted: boolean, studentId: int) -> HTTP Response
- FinalizeTeam(teamId: int) -> HTTP Response
- DeclineTeam(teamId: int) -> HTTP Response
- GetTeam(teamId: int) -> HTTP Response
- GetTeamScore(teamId: int) -> HTTP Response
- AddManualEvaluation(teamId: int, score: float) -> HTTP Response
- CreateBattle(title: String, description: String, registrationDeadline: DateTime, submissionDeadline: DateTime, languages: List<Language>, testCases: Map<String, List<File> >, buildScripts: Map<String, File>, minGroupSize: int, maxGroupSize: int, percentages: Map<String, int>, isManualScoreEnabled: boolean) -> HTTP Response
- ExportBattleLeaderboard(battleId: int) -> HTTP Response

2.5.5 ISessionManager

This Interface provides session related methods to components. Other components such as **Tournament Manager**, **Battle Manager** etc. They use to retrieve current user from token provided in request header in order to apply business logic related to specific user attributes.

- getCurrentUser(token: String) -> User

2.5.6 INotification

This Interface provides methods that enables other components to send notifications via email or in-app notifications. **INotification** abstracts the internal operations related to using notification services and provides simple methods.

- sendEmail(to: String, type: NotificationType, payload: Map<String,String>) -> boolean
  This method sends email to given email with predefined email type related to a template and payload of it.
- sendInAppNotification(userId: int, type: NotificationType, payload: Map<String,String>) -> boolean
  This method sends in-app notification to given user id with predefined notification type related to a template and payload of it.

2.5.7 Model Interfaces

These interfaces are mainly responsible for the query management and communication with DBMS. They provide database query methods for other components.

1. **Tournament Model**
    - GetAllTournaments() -> List<Tournament>
    - GetFilteredTournament(filter: TournamentFilterSchema) -> List<Tournament>
    - GetTournamentById(tournamentId: int) -> Tournament
    - CreateTournament(attributes: TournamentCreateSchema) -> int (Tournament Id)
    - EndTournament(tournamentId: int) -> boolean
    - AddEducator(tournamentId: int, educatorId: int) -> boolean
    - AddStudent(tournamentId: int, studentId: int) -> boolean
2. **Battle Model**
    - GetBattle(battleId: int) -> Battle
    - CreateBattle(attributes: BattleCreateSchema) -> int (Battle Id)
    - CreateCodeKata(attributes: CodeKataCreateSchema) -> int (CodeKata Id)
3. **User Model**
    - CreateStudent(attributes: StudentSchema) -> int (Student Id)
    - CreateEducator(attributes: EducatorSchema) -> int (Educator Id)
    - GetUser(userId: int) -> User
    - UpdateStudent(attributes: StudentSchema) -> int (Student Id)
    - UpdateEducator(attributes: EducatorSchema) -> int (Educator Id)
    - DeleteUser(userId: int) -> boolean
4. **Team Model**
    - CreateTeam(attributes: TeamCreateSchema) -> int (Team Id)
    - FinalizeTeam(teamId: int) -> boolean
    - UpdateStudentStatus(teamId: int, studentId: int, isAccepted: boolean) -> boolean
    - GetTeam(teamId: int) -> Team
5. **Submission Model**
    - CreateSubmission(attributes: SubmissionCreateSchema) -> int (Submission Id)
      This function overwrites the existing submission, more precisely, safe deletes it.
    - GetSubmission(teamId: int) -> Submission
    - GetSubmissions(battleId: int) -> List<Submission>
    - AddManualEvaluation(submissionId: int, score: float) -> boolean

2.5.8 IBattleTeamOrganiser

- UpdateTeamStatus(teamId: int, status: TeamStatus) -> boolean
- CreateTeam(attributes: TeamCreateSchema) -> Team
- GetTeam(teamId: int) -> Team
- AcceptTeamInvitation(teamId: int, studentId: int) -> boolean
- DeclineTeamInvitation(teamId: int, studentId: int) -> boolean

2.5.9 ISubmissionManager

GetSubmission(teamId: int) -> Submission

GetBattleScores(battle: Battle) -> List<Map<Team, int> >

2.5.10 ISubmissionListener

Submission(teamId: int, repoUrl: string) -> HTTP Status
This is an endpoint responsible to be triggered by Github Actions Workflow created by teams. When they made a new commit, they send a request to this endpoint.

2.5.11 ISubmissionRetriever

PullSubmission(repoUrl: String, battleId: int, teamId: int) -> String
This method provides the url of a folder in the file storage system. The pulled repository content is stored in this folder with a naming using battleId and teamId. The component using this method is then responsible for other actions.

2.5.12 IScoringHandler

CalculateTestCaseScore(code: File, testCaseFiles: Map<Language, List<File> >, buildScripts) -> float

CalculateStaticAnalysisScore(code: File, qualityAspects: List<QualityAspect>) -> float

CalculateTimelinessScore(submissionDate: DateTime, battleStartDate: DateTime) -> float

ICodeExecutor

- CreateSandboxEnviroment(language: Language, build
- RunTestCases(code: File, language: Language, testCase: File) -> boolean

IStaticAnalysis

- GetStaticAnalysisScore(code: File, language: Language, qualityAspects: List<QualityAspect>) -> float

2.5.13 IBattleRepoHandler

- CreateBattleRepository(battle: Battle) -> String (Repository URL)

2.5.14 IGithubAPI

This is the external API used to retrieve submission from Github. This Interface is explained later.

2.5.15 IFileManagement

This is the external service to store and manager files stored in the application. Cloud Object Storage is used. This Interface is explained later.

2.5.16 API Endpoints

1. **Endpoint Auth/Register**
   Method: POST
   Request Body:

   - email: String
   - password: String
   - name: String
   - surname: String
   - institutionInfo: List<Institution>
   - userType: UserType

   Response:

   - **200**
     - message: "User is registered successfully"
     - id: int
   - **400**
     - message: "Email exists"

2. **Endpoint Auth/Login**
   Method: POST
   Request Body:

   - email: String
   - password: String

   Response:

   - **200**
     - message: "User is logged in successfully"
     - token: String
   - **400**
     - message: "Credentials are wrong"

3. **Endpoint Auth/Logout**
   Method: POST
   Header:

   - token: String

   Response:

   - **200**
     - message: "User is logged out successfully"
   - **400**
     - message: "Something went wrong"
4. **Endpoint Auth/VerifyEmail**
   Method: POST
   Request Body:

   - email: String

   Response:

   - **200**
     - message: "User is verified successfully"
   - **400**
     - message: "Something went wrong"
5. **Endpoint Profile/GetMyProfile**
   Method: GET
   Header:

   - token: String

   Response:

   - **200**
     - profile: Student
   - **200**
     - profile: Educator
   - **400**

    - message: "Something went wrong"

6. **Endpoint Profile/GetProfile/:userId**
   Method: GET
   Header:

   - token: String

   Response:

   - **200**
     - profile: Student
   - **200**
     - profile: Educator
   - **400**
     - message: "Something went wrong"

7. **Endpoint Profile/EditProfile**
   Method: POST
   Header:

   - token: String

   Request Body:

   - email: String
   - password: String
   - newPassword: String
   - name: String
   - surname: String
   - institutionInfo: List<Institution>

   Response:

   - **200**
     - profile: Student
   - **200**
     - profile: Educator
   - **400**

  - message: "Email exists"
- **400**
  - message: "Password is wrong"

8. **Endpoint Profile/GetEducators**
   Method: GET
   Header:
   - token: String

   Response:
   - **200**
     - educators: List<Educator>
   - **400**
     - message: "Something went wrong"
9. **Endpoint Tournament/GetTournaments**
   Method: GET
   Header:
   - token: String

   Request Parameters:
   - registrationStatus: RegistrationStatus
   - availability: Availability
   - sortOption: TournamentSortOption
   - sortDirection: SortDirection

   Response:
   - **200**
     - tournaments: List<Tournament>
   - **400**
     - message: "Something went wrong"
10. **Endpoint Tournament/GetTournamentInfo/:tournamentId**
    Method: GET
    Header:

- token: String

Response:

- **200**
  - tournament: Tournament
- **400**
  - message: "Something went wrong"

11. **Endpoint Tournament/RegisterTournament/:tournamentId**
    Method: POST
    Header:

    - token: String

    Response:

    - **200**
      - tournamentId: int
    - **400**
      - message: "Can not register Tournament"

12. **Endpoint Tournament/GetTournamentBattles/:tournamentId**
    Method: GET
    Header:

    - token: String

    Request Parameters:

    - minGroupSize: int
    - maxGroupSize: int
    - startDate: DateTime
    - endDate: DateTime
    - registrationStatus: RegistrationStatus
    - institution: Instituion
    - langauges: List<Language>
    - educatorIds: List<int>
    - searchText: String

    Response:

- **200**
  - battles: List<Battle>
- **400**
  - message: "Something went wrong"

13. **Endpoint Tournament/GetTournamentLeaderboard/:tournamentId**
    Method: GET
    Header:
    - token: String

    Response:
    - **200**
      - leaderboard: List<Map<String, int> >
    - **400**
      - message: "Something went wrong"
14. **Endpoint Tournament/ExportTournamentRankings/:tournamentId**
    Method: GET
    Header:
    - token: String

    Response:
    - **200**
      - leaderboard: StreamResponse(File)
    - **400**
      - message: "Something went wrong"
15. **Endpoint Tournament/CreateTournament**
    Method: POST
    Header:
    - token: String

    Request Body:
    - title: String

  - description: String
  - registrationDeadline: DateTime
  - invitedEducatorIds: List<int>

  Response:

  - **200**
    - tournament: Tournament
  - **400**
    - message: "Tournament can not be created"

16. **Endpoint Tournament/AnswerTournament/:tournamentId**
    Method: POST
    Header:

    - token: String

    Request Body:

    - isAccepted: boolean
    - educatorId: int

    Response:

    - **200**
      - message: "Tournament invitation is answered"
    - **400**
      - message: "Something went wrong"

17. **Endpoint Tournament/EndTournament/:tournamentId**
    Method: POST
    Header:

    - token: String

    Response:

    - **200**
      - message: "Tournament is ended"
    - **400**

    - message: "Something went wrong"

18. **Endpoint Battle/GetBattleInfo/:battleId**
    Method: GET
    Header:

    - token: String

    Response:

    - **200**
        - battle: Battle
    - **400**
        - message: "Something went wrong"

19. **Endpoint Battle/GetBattleRankings/:battleId**
    Method: GET
    Header:

    - token: String

    Response:

    - **200**
        - rankings: List<Map<String, int> >
    - **400**
        - message: "Something went wrong"

20. **Endpoint Battle/RegisterBattle/:battleId**
    Method: POST
    Header:

    - token: String

    Request Body:

    - teamName: String
    - teamInvitations: List<int>

    Response:

    - **200**

    - message: "Registered successfully"
  - **400**
    - message: "Something went wrong"

21. **Endpoint Battle/AnswerBattleInvitation/:teamId**
    Method: POST
    Header:
    - token: String

    Request Body:
    - isAccepted: boolean
    - studentId: int

    Response:
    - **200**
      - message: "Battle invitation is answered"
    - **400**
      - message: "Something went wrong"

22. **Endpoint Battle/FinalizeTeam/:teamId**
    Method: POST
    Header:
    - token: String

    Response:
    - **200**
      - message: "Team is finalized"
    - **400**
      - message: "Something went wrong"

23. **Endpoint Battle/DeclineTeam/:teamId**
    Method: POST
    Header:
    - token: String

Response:

- **200**
  - message: "Team is declined"
- **400**
  - message: "Something went wrong"

24. **Endpoint Battle/GetTeam/:teamId**
    Method: GET
    Header:

    - token: String

    Response:

    - **200**
      - team: Team
    - **400**
      - message: "Something went wrong"

25. **Endpoint Battle/GetTeamScore/:teamId**
    Method: GET
    Header:

    - token: String

    Response:

    - **200**
      - score: Map<ScoringCriteria, int>
    - **400**
      - message: "Something went wrong"

26. **Endpoint Battle/AddManualEveluation/:teamId**
    Method: POST
    Header:

    - token: String

    Request Body:

- score: float

Response:

- **200**
  - message: "Manual evaluation is done successfully"
- **400**
  - message: "Something went wrong"

27. **Endpoint Battle/CreateBattle**
    Method: POST
    Header:

    - token: String

    Request Body:

    - title: String
    - description: String
    - registrationDeadline: DateTime
    - submissionDeadline: DateTime
    - languages: List<Language>
    - testCases: Map<String, List<File> >
    - buildScripts: Map<String, File>
    - minGroupSize: int
    - maxGroupSize: int
    - percentages: Map<String, int>
    - isManualScoreEnabled: boolean

    Response:

    - **200**
      - battle: Battle
    - **400**
      - message: "Battle can not be created"

28. **Endpoint Battle/ExportBattleRankings/:battleId**
    Method: GET
    Header:

    - token: String

Response:

- **200**
  - rankings: StreamResponse(File)
- **400**
  - message: "Something went wrong"

2.6 Selected Architectural Styles and Patterns

2.6.1 3-Tier Architecture

The 3-Tier Architecture is a widely-used design pattern in the development of web-based applications. It divides the application architecture into three distinct layers, each with a specific function, promoting a modular and scalable approach. The three layers are:

Presentation Layer (Client Tier): This is the user interface of the application. It's responsible for displaying user interface elements and processing user input. It communicates with the Business Logic Layer for data processing and operations.

Business Logic Layer (Application Tier): This layer is the core of the application, handling the business logic. It processes user requests, performs calculations, and makes logical decisions. It communicates between the Presentation Layer and the Data Layer, acting as a mediator for data retrieval and storage.

Data Layer (Data Tier): This layer is responsible for managing the database. It stores, retrieves, and updates data in a structured format. This layer ensures data integrity and security.

Benefits of 3-Tier Architecture:

Modularity: Each layer can be developed and maintained independently. This separation of concerns makes the system more manageable and organized.

Scalability: Each layer can be scaled independently based on demand. For example, you can increase the capacity of the data layer without altering the application or presentation layers.

Flexibility and Reusability: Changes in one layer generally do not affect the other layers. For example, the UI can be redesigned without altering the business logic. Similarly, the business logic can be modified without impacting the database structure.

Improved Security: The separation allows for better security measures. For example, the data layer can be secured independently of the other layers, minimizing the risk of data breaches.

Ease of Maintenance: Individual layers can be updated or repaired without affecting the entire system.

Possible Trade-offs:

Complexity: The architecture can be more complex to design and implement compared to simpler architectures like a monolithic design. This can lead to higher initial development costs and longer development time.

Performance Overhead: The inter-layer communication can introduce latency. For high-performance applications, this might be a limiting factor.

Deployment Complexity: Deploying a 3-tier application can be more complex than deploying a single-tier application, as it may involve setting up and managing multiple servers and environments.

Skill Requirements: The need for expertise in multiple technologies (front-end, back-end, database management) can be higher than in more unified architectures.

Compared to Other Architectures:

Vs. Monolithic Architecture: A monolithic architecture combines all three tiers into a single application. It is simpler to deploy and manage but can become unwieldy as the application grows, and it lacks the modularity and scalability of 3-tier architecture.

Vs. Microservices Architecture: Microservices architecture breaks down an application into small, independently deployable services. While it offers high scalability and modularity, it is more complex in terms of inter-service communication and managing multiple small components.

In summary, the 3-tier architecture offers a balanced approach between modularity, scalability, and manageability. It is well-suited for applications where these aspects are prioritized over the simplicity of deployment and initial development ease.

2.6.2 Layered Architecture

From deployment perspective we have designed Application Layer as a single component, however, from software design perspective it is layered and contains more layers. The application server is divided into Endpoints/Routers, Services (Main Business Logic), and Repositories (Data Access Layer), represents a layered architecture pattern, often used in modern web applications for clear separation of concerns.

1. **Endpoints/Routers:** This layer handles HTTP requests from clients. It's responsible for receiving requests, interpreting them, and directing them to the appropriate service layer for processing. It doesn't contain business logic or data access code; its sole purpose is to route requests to the correct parts of the application.

2. **Services (Main Business Logic):**This is the heart of your application. The service layer contains the core business logic. It processes requests forwarded from the endpoints/routers, applies business rules, and performs operations based on these rules. This layer doesn't directly interact with the database; instead, it communicates with the repository layer to access and manipulate data.

3. **Repositories (Data Access Layer):** The repository layer abstracts the data access logic from the service layer. It provides a collection of methods for accessing and manipulating data in your database or other storage mechanisms. The repository layer's main goal is to isolate the data access logic, making the service layer agnostic of the underlying data source and storage details.

Characteristics of This Architecture:

1. Separation of Concerns: Each layer has a distinct responsibility. Routers handle HTTP routing, services handle business logic, and repositories manage data access. This separation makes the application more maintainable and scalable.

2. Reusability: Each layer can be reused independently. For example, the same service layer can be used with different endpoints, and repositories can be reused across different services.

3. Testability: This architecture makes it easier to test each layer independently. For instance, you can mock the repository layer when testing services.

4. Flexibility: Changing one layer has minimal impact on the other layers. For example, you can change the data access logic without affecting the business logic.

5. Scalability: Each layer can be scaled independently based on the application's needs.

This structure aligns with the Layered Architecture pattern, also known as the n-tier architecture pattern. In web applications, it's common to have a multi-layered architecture like this, which helps in organizing code, improving maintainability, and ensuring the application can grow and evolve over time without becoming too complex or unwieldy.

In summary, this application architecture, with clear divisions between endpoints/routers, services, and repositories, is a well-organized example of a layered architecture. This approach is widely adopted in the development of scalable, maintainable, and testable web applications.

2.6.3 Facade

Definition: The Facade pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

Purpose: Facade is typically used to simplify a complex subsystem or to provide a single entry point to a level of functionality. It doesn't encapsulate the subsystem but provides a simplified interface to it.

When we consider the components in our system such as TournamentManager or BattleManager,this pattern can help us to provide simple methods like GetBattles() or GetTournaments() without going in underlying logic in the application server. Facade simplifies the usage of the Tournament or Battle in this example.

2.6.4 Mediators

The Mediator pattern is a behavioral design pattern in software engineering, used to reduce complex or tight coupling between components or classes. It promotes loose coupling by encapsulating the way different sets of objects interact and communicate with each other. By introducing a mediator object, all communication between different components is centralized within the mediator, instead of being spread across multiple components.
When a Component needs to communicate with another, it sends a message to the Mediator instead of sending it directly to the other Component. The Mediator receives the message and decides how to pass these messages to the appropriate Component or Components. The Mediator might also process or transform the data before forwarding it.

- **Notification Manager:** Notification Manager behaves as Mediator between internal logic of application and Notification Services used to send notification via email or in-app notification. It help us to sustain a flexibility because it is easy to adopt Notification Manager when Notification Service is changed. Otherwise, we have to change every component when solutions used for notification is changed. **Email Manager** and **In-App Notification Manager** also help to link this internal logic to external APIs.

- **Static Analysis Manager:** Static Analysis Manager also acts as a Mediator operating between application server and Static Analysis Tool. Without changing internal components of system, we can adopt to changes of external Static Analysis Tool by just modifying the Static Analysis Manager.

- **Github Manager:** Github Managers is other component acts as a Mediator between internal components and Github as an external service. It is responsible to apply logic related to Github. The changes in the Github environment and Github API do not affect the system thanks to this component. In other words, Github Manager can be easily modified according to changes related to Github without changing other components in the application.

2.7 Other Design Decisions

In this section we explained further Design Decisions.

2.7.1 Load Balancing, Firewall and Replication

As we mentioned before we use Load Balancer to satisfy constraints related to **Availability**. We are aiming to less downtime by distributing incoming traffic to the running instances. Moreover, using Firewall and HTTPS for all data transactions, our **Security** constraints mentioned in the RASD will be held. Finally, sustaining Database Replication improves the **Sustainability** by backup options and recovery plans.

2.7.2 Sandbox Paradigm

In CodeKataBattle, it is needed to run submitted code via some test cases. To do so, we decided to Sandboxing. The Sandboxing in software engineering refers to a secure, controlled environment where programs can be executed without affecting the host system. This environment strictly controls the resources and permissions available to the program, ensuring that any code run within the sandbox cannot interfere with the system outside of it. This concept is crucial in scenarios where untrusted or untested code needs to be executed safely.

- Secure Execution of Untrusted Code: When users submit solutions to coding problems on CodeKataBattle, their code is executed on the server. Since this code comes from external, untrusted sources, running it directly on the server poses significant security risks.
- Isolation: To mitigate these risks, CodeKataBattle executes user-submitted code within a sandbox. This sandbox environment isolates the code, ensuring that it can't access or manipulate the server's system resources, file system, or network in unauthorized ways.
- Resource Limitation: The sandbox also limits the amount of CPU, memory, and other resources the code can use. This prevents issues like infinite loops or excessively resource-intensive operations from affecting the server's stability.
- Automated Evaluation: Sandboxes facilitate automated testing of submitted code against predefined test cases. This automation is essential for efficiently handling a large number of submissions and providing immediate feedback.
- Consistency in Testing: By running each submission in a standardized environment, you ensure that all code is tested under the same conditions. This is crucial for fairness in judging, as it eliminates variances that could arise from different execution environments.

In summary, sandboxing on a coding challenge platform ensures security, fairness, stability, and scalability. It protects the platform's integrity and the data of its users while providing a fair and consistent environment for evaluating code submissions.

Submission Scoring Algorithm:

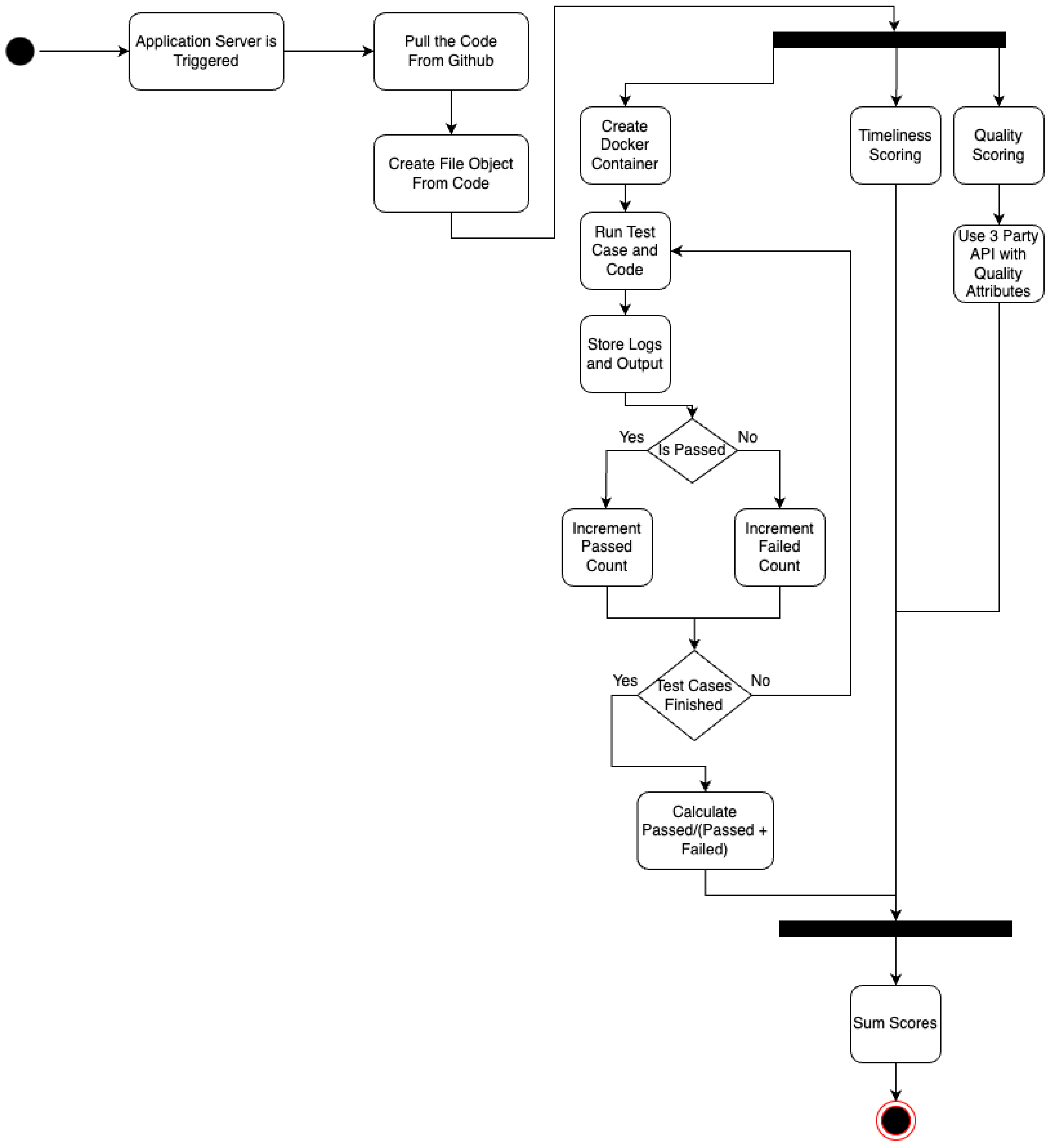

Figure 5: Submission Scoring

3 User Interface Design

4 Requirements Traceability

5 Implementation, Integration, and Test Plan

6 Effort Spent

References