POLITECNICO
MILANO 1863

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

List of Figures

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.

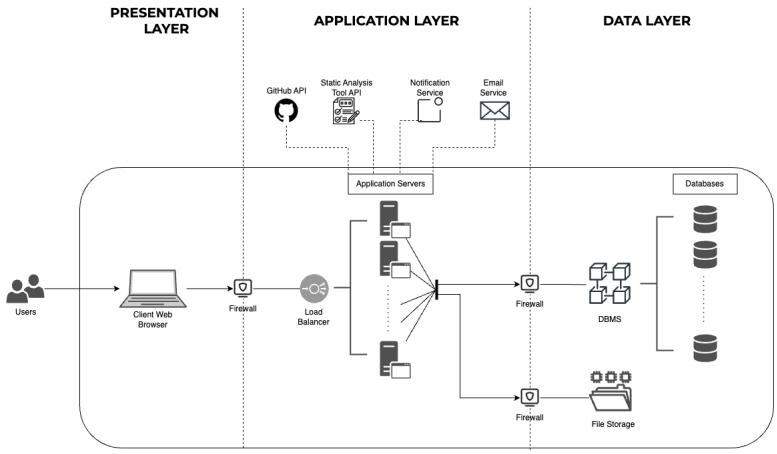

Figure 1: High-Level Architecture of the System

2.2 Component View

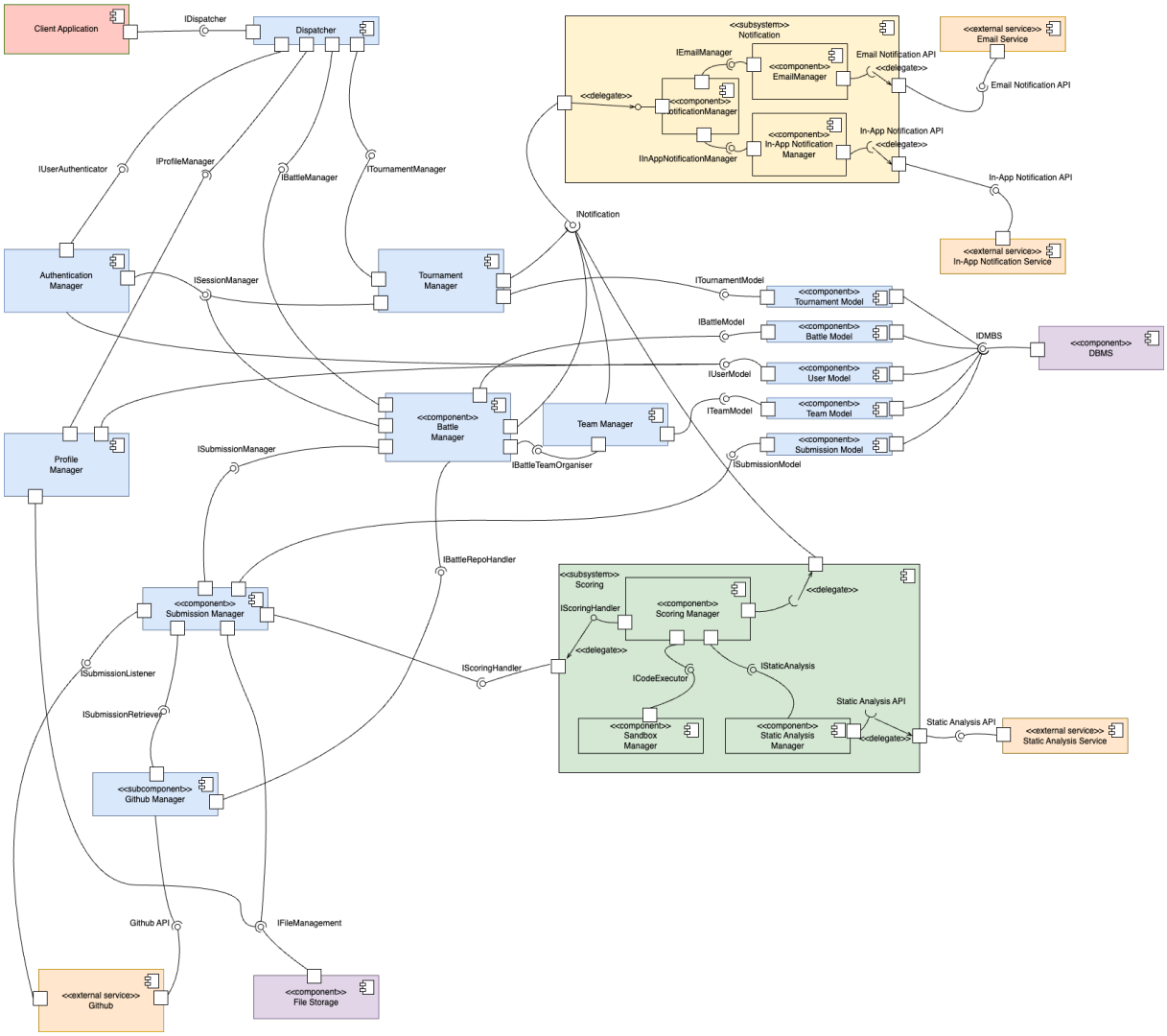

Figure 2: Component Diagram

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.

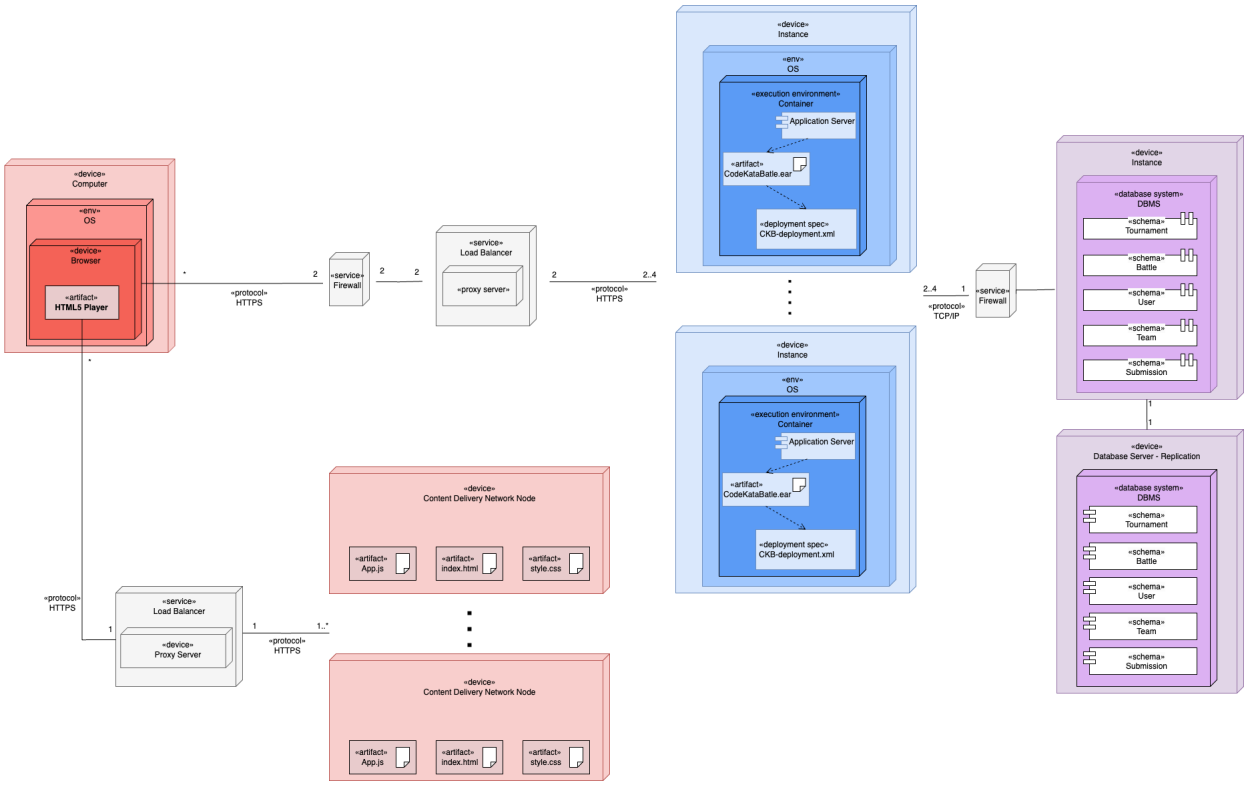

Figure 3: Deployment Diagram

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.

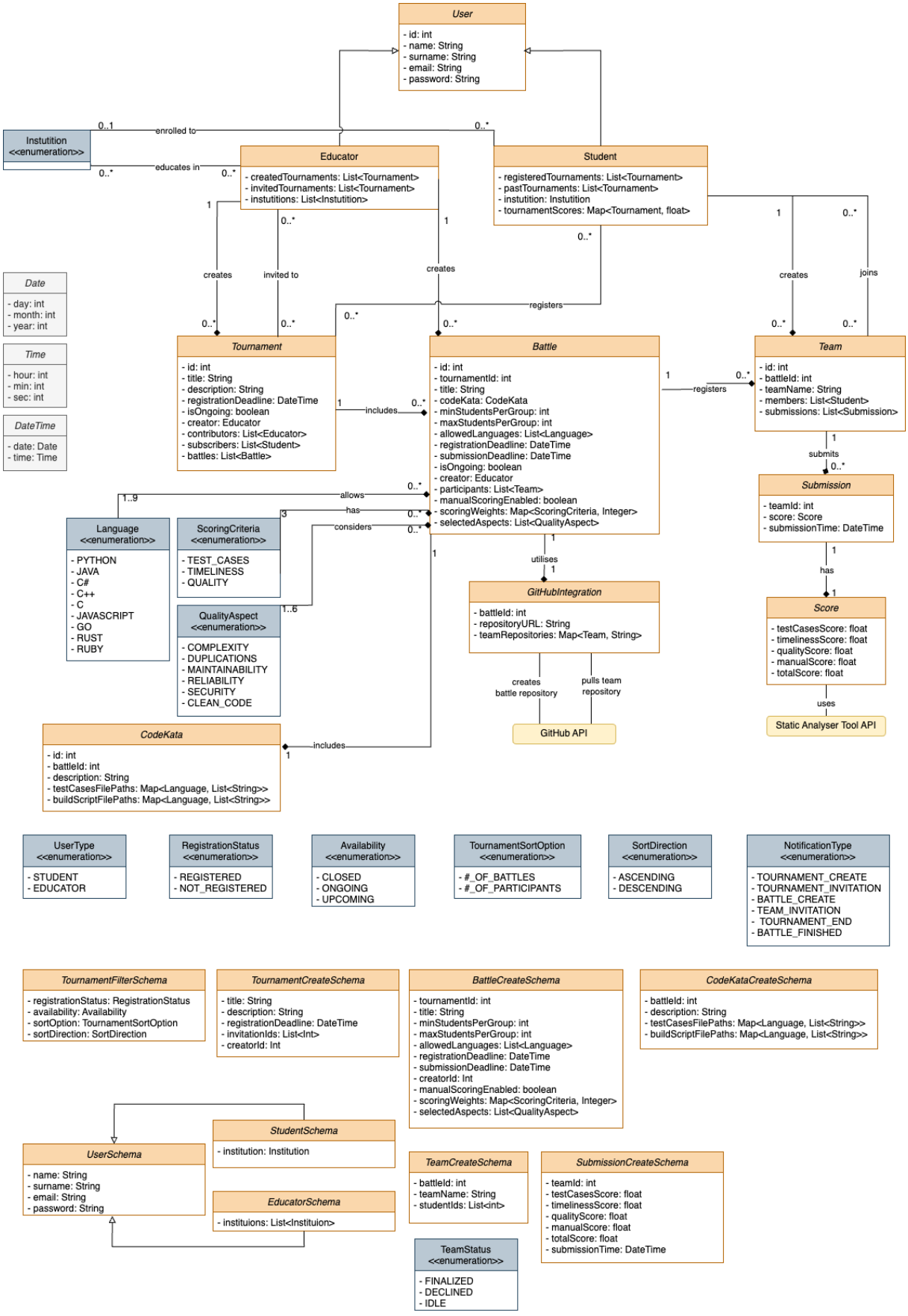

Figure 4: Class Diagram (Based on RASD)

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 CodeKata-Battle, 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:

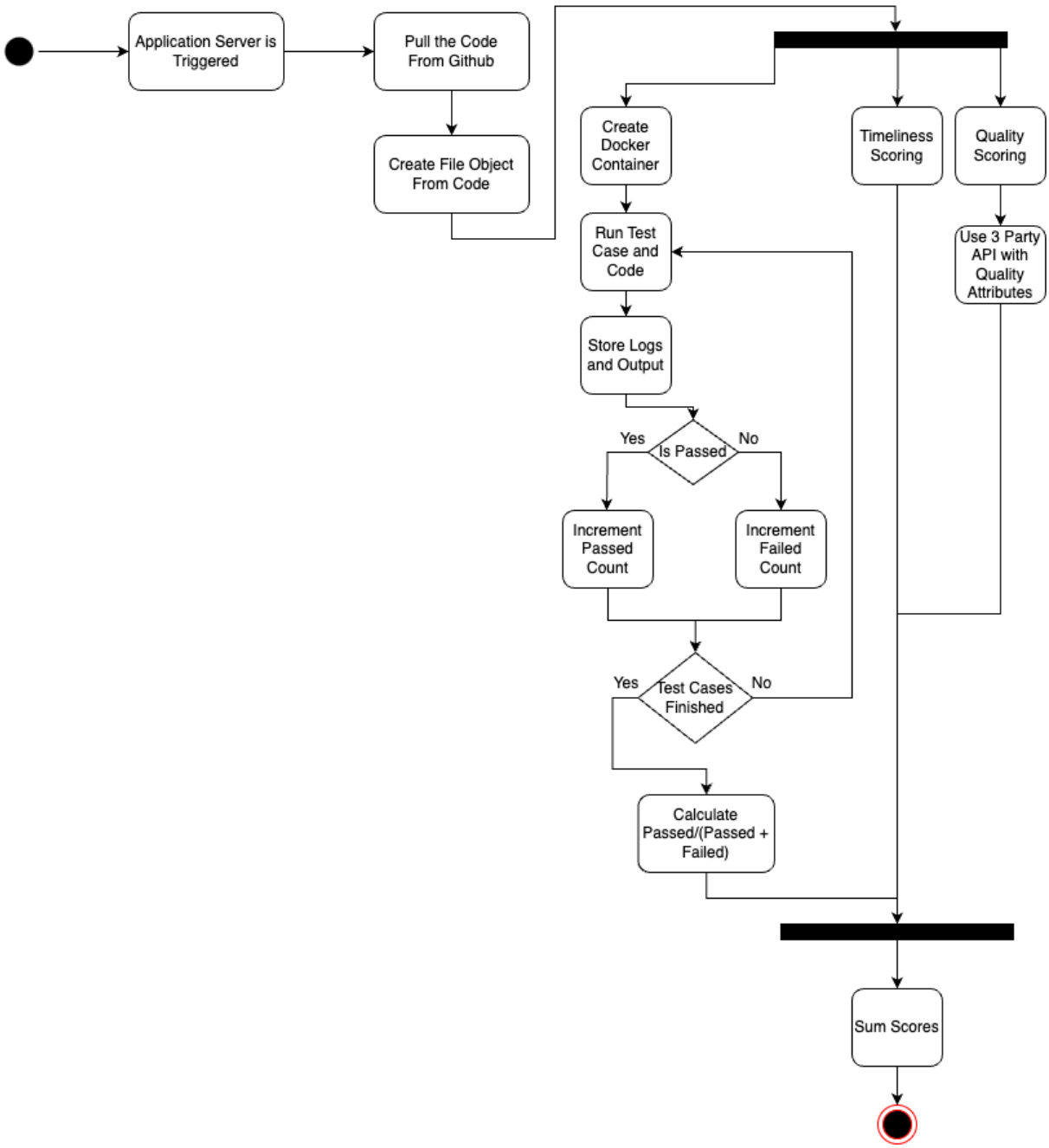

Figure 5: Submission Scoring

3 User Interface Design

4 Requirements Traceability

In this section, we are going to illustrate the mapping of the components introduced in this document on the functional requirements introduced in the RASD. Some closely related requirements are explained together in the tables below.

R1	The system shall allow unregistered users to register by providing a unique email, a password, a name, a surname, and institution information.
Authentication Manager	Gets the client request, and directs it to User Repository.
User Repository	Creates the user.
DBMS	Saves the user.
Notification Manager	Triggers Email Manager.
Email Manager	Sends verification email.

Table 1: Mapping on R_1

R2	The system shall allow users who received a verification email to verify their emails.
Authentication Manager	Gets the client request, and directs it to User Repository.
User Repository	Verifies the user.
DBMS	Updates the user.

Table 2: Mapping on R_2

R3	The system shall allow registered users to log in by providing an email, and a password.
Authentication Manager	Gets the client request, and directs it to User Repository.
User Repository	Validates the user credentials.
DBMS	Returns user info.

Table 3: Mapping on R_3

R4	The system shall allow authenticated users to view all tournaments.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Gets all tournaments.
DBMS	Returns all tournaments.

Table 4: Mapping on R_4

R5	The system shall allow authenticated users to filter the tournaments by the registration status and availability.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Gets filtered tournaments using a query.
DBMS	Returns queried tournaments.

Table 5: Mapping on R_5

R6	The system shall allow authenticated users to sort the tournaments by number of participants and number of battles in it.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Gets sorted tournaments using a query.
DBMS	Returns queried tournaments.

Table 6: Mapping on R_6

R7	The system shall allow authenticated users to view a specific tournament.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Gets the tournament by its id using a query.
DBMS	Returns queried tournament.

Table 7: Mapping on R_7

R8	The system shall allow authenticated users to view the leaderboard of the tournament.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Gets the tournament scores of subscribed users and then sorts them.
DBMS	Returns tournament scores.

Table 8: Mapping on R_8

R9	The system shall allow authenticated users to view all battles in a tournament.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Gets the list of battles of the tournament.
DBMS	Returns the list of battles.

Table 9: Mapping on R_9

R10	The system shall allow authenticated users to filter the battles by group size, start date - end date, registration status, institution of the battle creator, allowed programming languages, and the battle creator.
Battle Manager	Gets the client request, and directs it to Battle Repository.
Battle Repository	Gets filtered battles using a query.
DBMS	Returns queried battles.

Table 10: Mapping on R_{10}

R11	The system shall allow authenticated users to search battles by text search.
Battle Manager	Gets the client request, and directs it to Battle Repository.
Battle Repository	Gets searched battles using a query.
DBMS	Returns queried battles.

Table 11: Mapping on R_{11}

R12	The system shall allow authenticated users to view a specific battle and its instructions.
Battle Manager	Gets the client request, and directs it to Battle Repository.
Battle Repository	Gets the battle by its id using a query.
DBMS	Returns queried battle.

Table 12: Mapping on R_{12}

R13	The system shall allow authenticated users to view the rankings of the battle.
Battle Manager	Gets the client request, and directs it to Battle Repository.
Battle Repository	Gets the battle scores of participating teams and then sorts them.
DBMS	Returns battle scores.

Table 13: Mapping on R_{13}

R14	The system shall allow authenticated users to search tournaments with text search by educators, by titles, or by institutions.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Gets searched tournaments using a query.
DBMS	Returns queried tournaments.

Table 14: Mapping on R_{14}

R15	The system shall allow authenticated users to manage their profiles. (a) The system shall allow authenticated users to view their profiles. (b) The system shall allow authenticated users to delete their profiles unless they do not have an ongoing tournament created by themselves. (c) The system shall allow authenticated users to view their settings. (d) The system shall allow authenticated users to edit their settings by name, surname, password, and institution information. (e) The system shall oblige authenticated users to enter their old password during settings editing.
Profile Manager	Gets the client request, and directs it to User Repository.
User Repository	(a) Gets the user profile. (b) Deletes the user. (c) Gets the user settings. (d) Edits the user settings. (e) Edits the user settings.
DBMS	(a) Returns user profile. (b) Removes user. (c) Returns user settings. (d) Updates user settings. (e) Updates user settings.

Table 15: Mapping on R_{15}

R16	The system shall allow educators to create tournaments by providing a title, a description, and a registration deadline.
R34	The system shall trigger the email service to send a notification email for a newly created tournament for all registered users.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Creates the tournament.
DBMS	Saves the tournament.
Notification Manager	Triggers Email Manager.
Email Manager	Sends notification email to all users.

Table 16: Mapping on R_{16} and R_{34}

R17	The system shall allow educators to invite other educators to their tournaments during tournament creation.
R37	The system shall trigger the email service to send an invitation notification for the tournament to the invitee educators.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Creates the tournament.
DBMS	Saves the tournament.
Notification Manager	Triggers In-App Notification Manager.
In-App Notification Manager	Sends invitation notification to invitee educators.

Table 17: Mapping on R_{17} and R_{37}

R18	The system shall allow educators to edit the tournaments that they have created by providing a title, a description, and a registration deadline unless the registration deadline has not passed.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Edits the tournament.
DBMS	Updates the tournament.

Table 18: Mapping on R_{18}

R19	The system shall allow educators to end the tournaments that they have created.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Ends the tournament.
DBMS	Updates the tournament.

Table 19: Mapping on R_{19}

R20	The system shall allow educators to accept or reject the tournament invitation to create battles coming from other educators for a tournament.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Adds the educator to the tournament contributors list.
DBMS	Updates the tournament.

Table 20: Mapping on R_{20}

R21	The system shall allow educators to create battles by providing a title, a description, a registration deadline, a submission deadline, the allowed languages, test cases, build scripts, minimum & maximum group size, and the scoring criteria.
Battle Manager	Gets the client request, and directs it to Battle Repository.
Battle Repository	Creates the battle.
DBMS	Saves the battle.
Notification Manager	Triggers Email Manager.
Email Manager	Sends notification email to users that subscribe to the battle's tournament.

Table 21: Mapping on R_{21}

R22	The system shall oblige educators to upload test case file and build script for every allowed language in battle.
Battle Manager	Gets the client request, and uploads the battle files.
File Manager	Stores the battle files.

Table 22: Mapping on R_{22}

R23	The system shall allow educators to manually evaluate the submissions giving extra point between 0 and 10 after the submission deadline has passed.
Battle Manager	Gets the client request, and directs it to Submission Manager.
Submission Manager	Utilises Scoring Manager, and gives the score to the Submission Repository.
Scoring Manager	Enables educator to evaluate.
Submission Repository	Gives point to the submission.
DBMS	Updates the submission.

Table 23: Mapping on R_{23}

R24	The system shall allow educators to edit the battles that they have created.
Battle Manager	Gets the client request, and directs it to Battle Repository.
Battle Repository	Edits the battle.
DBMS	Updates the battle.

Table 24: Mapping on R_{24}

R25	The system shall allow students to register for tournaments.
Tournament Manager	Gets the client request, and directs it to Tournament Repository.
Tournament Repository	Adds the student to the tournament subscribers.
DBMS	Updates the tournament.

Table 25: Mapping on R_{25}

R26	The system shall allow students to register for battles in which the tournaments that they have registered for.
R27	The system shall allow students to register for battles individually.
R28	The system shall allow students to register for battles by a team having a team name.
Battle Manager	Gets the client request, and triggers Team Manager. With the response coming from the Team Manager, it utilises Battle Repository.
Team Manager	Creates a team. (Regardless of the number of students on the team, it is created. Individual students will be considered as a team of one in the project.)
Team Repository	Creates the team.
Battle Repository	Adds the team to the battle participants.
DBMS	Saves the team, and updates the battle.

Table 26: Mapping on R_{26}, R_{27}, R_{28}

R29	The system shall allow students to invite other students to their team during battle registration.
R35	The system shall trigger the email service to send an invitation notification for the battle to the invitee students.
Battle Manager	Gets the client request, and directs it to Team Manager.
Team Manager	Notifies invitee students.
Notification Manager	Triggers In-App Notification Manager.
In-App Notification Manager	Sends invitation notification to invitee students.

Table 27: Mapping on R_{29}

R30	The system shall allow students to accept or reject the team invitation coming from other students for a battle.
Battle Manager	Gets the client request, and directs it to Team Manager.
Team Manager	Utilises Team Repository.
Team Repository	Adds the student to the team members.
DBMS	Updates the team.

Table 28: Mapping on R_{30}

R31	The system shall allow students to finalize their team registration or decline it.
Battle Manager	Gets the client request, and directs it to Team Manager.
Team Manager	Utilises Team Repository.
Team Repository	Finalises or declines the team.
DBMS	Updates or removes the team.

Table 29: Mapping on R_{31}

R32	The system shall create a repository for a battle after the registration deadline for that battle has passed.
R36	The system shall trigger the email service to send a notification email including the link to the battle repository to the students registered for it.
Battle Manager	Initialises repo creation for a battle.
GitHub Manager	Creates repository, return the URL to the Battle Manager.
Battle Repository	Adds repository URL to the battle.
DBMS	Updates the battle.
Notification Manager	Triggers Email Manager.
Email Manager	Sends notification email to the users that participate in the battle.

Table 30: Mapping on R_{32} and R_{36}

R33	The system shall pull the repository of a team following a trigger from GitHub Actions.
Submission Manager	Triggered by GitHub Actions, utilises GitHub Manager and File Manager.
GitHub Manager	Pulls the repository.
File Manager	Stores the files retrieved from repository.

Table 31: Mapping on R_{33}

R38	The system shall automatically evaluate submissions by scoring criteria. (a) The system shall score the submission with respect to test cases, and test case weight. (b) The system shall score the submission with respect to timeliness, and timeliness weight. (c) The system shall score the submission with respect to quality aspects, and quality aspect weight.
R39	The system shall utilise a Static Analysis Tool to calculate the score in terms of quality aspects.
R40	The system shall create a sandbox environment for each team for the submissions in order to run the codes.
Submission Manager	Gets submission files and send them to evaluation.
File Manager	Retrieves previously stored files from File Storage.
Scoring Manager	Calculates test case scores and timeliness. Gets help from Static Analysis Manager to calculate static analysis score.
Sandbox Manager	Creates sandbox environment, runs the code, returns the output. Kills the environment after usage.
Static Analysis Manager	Calculates static analysis score.

Table 32: Mapping on R_{38}, R_{39}, R_{40}

R41	The system shall automatically update the battle score of a team after the evaluation of the submission.
Submission Manager	Delivers the evaluation result to the Submission Repository.
Submission Repository	Gives score to the submission.
DBMS	Updates the submission.

Table 33: Mapping on R_{41}

R42	The system shall automatically update the battle rankings when a score is updated.
Battle Manager	Retrieves scores to display battle rankings.
Battle Repository	Gets the battle scores of participating teams and then sorts them.
DBMS	Returns battle scores.

Table 34: Mapping on R_{42}

R43	The system shall automatically update the tournament leaderboard at the end of each battle.
Tournament Manager	Retrieves the tournament scores to display the leaderboard.
Tournament Repository	Gets the tournament scores of subscribed users and then sorts them.
DBMS	Returns tournament scores.

Table 35: Mapping on R_{43}

5 Implementation, Integration, and Test Plan

In this section, we briefly explained the planned implementation, integration and testing of system components.

5.1 Implementation

The system is structured around three primary layers: client, business, and data. These will be concurrently developed and later integrated. By doing so, each layer can be tested individually, and after integration, comprehensive system-wide testing can be conducted. We will employ a hybrid strategy, combining bottom-up and thread methodologies, to leverage the advantages of both.

The thread approach aids in creating interim deliverables for stakeholder evaluation, proving extremely beneficial for system validation. Concurrently, the bottom-up strategy facilitates incremental integration, enhancing bug detection by allowing for the testing of subsystems as they progressively evolve through the addition of new modules.

The thread strategy involves pinpointing the system's functionalities and the specific segments of the components (termed as sub-components for practical purposes, though they may not align precisely with design sub-components) that are responsible for these functionalities. Since multiple sub-components collaboratively contribute to a single function, it's crucial to establish a sequence for their implementation, for which the bottom-up approach is particularly useful.

This mixed strategy enables the allocation of different feature implementations to independent development teams working in parallel. However, it's essential to identify any common components beforehand to prevent duplication of efforts in creating the same component or sub-component more than once. This approach not only streamlines the development process but also ensures a more efficient and organized integration of the various system parts.

5.1.1 Client Application

Static parts of Client Web Application ,in other words Frontend, can be implemented without server side implementation. Mockups and UI-UX design can be used for this development. However, dynamic actions rely on REST API to communicate server-side responsible for business logic. So, using documentation, unit test mocking REST API can be used to implement Frontend code.

5.1.2 Application Server

Application Server, in other words Backend, is responsible for the business logic in server side. We can separate some parts of the Application Server to be developed in parallel. The components to be incorporated into the system are derived directly from the system's requirements. The developers can be work separately on these components and integrate and test at the end of their development the components to evaluate the dependency and the interactions between them. The entire business logic can be tested separately from the client logic, which accelerates the development process. Below is a concise summary of the system's components with priorities.

Component	Importance	Complexity
Authentication Manager	High	Low
Profile Manager	Medium	Low
Tournament Manager	High	High
Battle Manager	High	High
Team Manager	Medium	Low
Notification Manager	Low	Medium

Repository Components	High	Medium
Submission Manager	High	Medium
Github Manager	Low	Low
Scoring Manager	High	High
File Manager	Medium	Low

So, briefly, we have to consider the importance and complexity of a component before starting to implement the component. This kind of analysis enables us to estimate time and work cost spent for a certain part of application.

5.1.3 Implementation Plan

1. **Repository Components:** For the application, data access layer has a great importance because all other components mainly rely on data access at some point. Database interaction is crucial because of the CRUD operations. So Tournament, Battle, User, Team and Submission Repository components are implemented firstly in parallel because there is no dependency between them.
2. **Notification Manager:** Even though a notification system is not a crucial part of the system, it's methods widely used by lots of components, so implementing it at early stages can be very helpful to see its usage among other components when implementing them. EmailManager and InAppNotificationManager are another required components for the Notification Component and they are considered within Notification Manager Component.
3. **Authentication Manager:** This component offers 2 interfaces which is related to users' authentication. **Registration, login and logout** functionalities are very fundamental to operate all business logic. Also it provides a session manager interface which is used by other components in which it is important to know who the user is, what type it is and what permissions it has.
4. **Tournament Manager:** Implementation of this component can be at the end if we follow just bottom-up approach. However, one of our goals is to create intermediate products and this component, even if it is a complex component, can helps us to create such an intermediate product. It has also no dependency other than Tournament Repository, Notification Manager and Authentication Manager.
5. **File Manager:** This component mainly responsible to communicate with File Storage Service. It encapsulate the operations such as fetch,read or write file from or to the file system on the cloud. It provides an interface to manage files so it is needed by other components.
6. **Profile Manager:** Unlikely to Tournament, this component has really low complexity without no real dependency other than repositories. By implementing Profile Manager at early stages, we can achieve a intermediate product with less effort and no dependency is needed.
7. **Scoring Manager:** Again this is another components which is responsible to communicate to an external service. Also it provides an iterface to calculate score of a submission. This component also has high priority with its required components such as Sandbox Manager and Static Analysis Manager.
8. **Github Manager:** Even though Github Manager is not fundamental as some main components such as Submission Manager and Battle Manager, implementing it early comes with advantages. It provides interfaces to these components. Also there can be another components using some logic related to Github.

9. **Submission Manager:** This component is one of the main components in the application which provides Submission related methods. It is very critic to use because it also provides an endpoint to be triggered by Github.
10. **Team Manager**: It provides interface related to team operations. Needed from Battle Manager.
11. **Battle Manager:** This components provides the endpoints about Battles and operates the logic related to Battles. It uses various interfaces from various components so it is good to implement this component at the last stages.

5.2 Integration & Testing

The section describes the integration plan of the different components and subcomponents of the CodeKata-Battle. The provided graphs illustrate the interdependencies between various components and subcomponents. Before any subcomponent is combined with others to create a larger component, it must undergo unit testing. Following this integration, the complete component is then subjected to further testing.

Before proceeding with the integration testing of a component, it is crucial to satisfy two key conditions, each with its distinct benefits. The first condition is that the component's interface must encompass all the functionalities detailed in the component interface diagram. This ensures that the component adheres to the predefined design specifications, which is vital for maintaining consistency and predictability in the system's overall behavior. Adhering to these specifications also facilitates easier integration with other components, as each piece is designed to fit seamlessly within the broader system architecture.

The second prerequisite is that the component must successfully clear all unit tests. Unit testing plays a critical role in verifying that each individual part of the component functions as intended in isolation. This process helps in identifying and rectifying any bugs or issues at an early stage, significantly reducing the risk of defects in the later stages of development. Successful unit testing guarantees a higher level of reliability and stability in the component, which in turn contributes to the robustness of the final integrated system. By ensuring that each component is thoroughly tested and fully functional before integration, the overall quality and performance of the system are enhanced, leading to a more efficient, reliable, and effective product.

As you can see the diagrams below, Driver is used as unit test to mock the not-finished components. This part is crucial during the integration test written here should be passed.

First integration to be handled is integrating **Repository Components** to DBMS in order to sustain a working data access layer. To enable other components to handle data this integration is very crucial. This is a prelimanary integration with DBMS. Unit Testing is used to mock the other components being under development. This Driver is mainly responsible to mock repository calls from the components using them. Moreover, it checks the validity of the results.

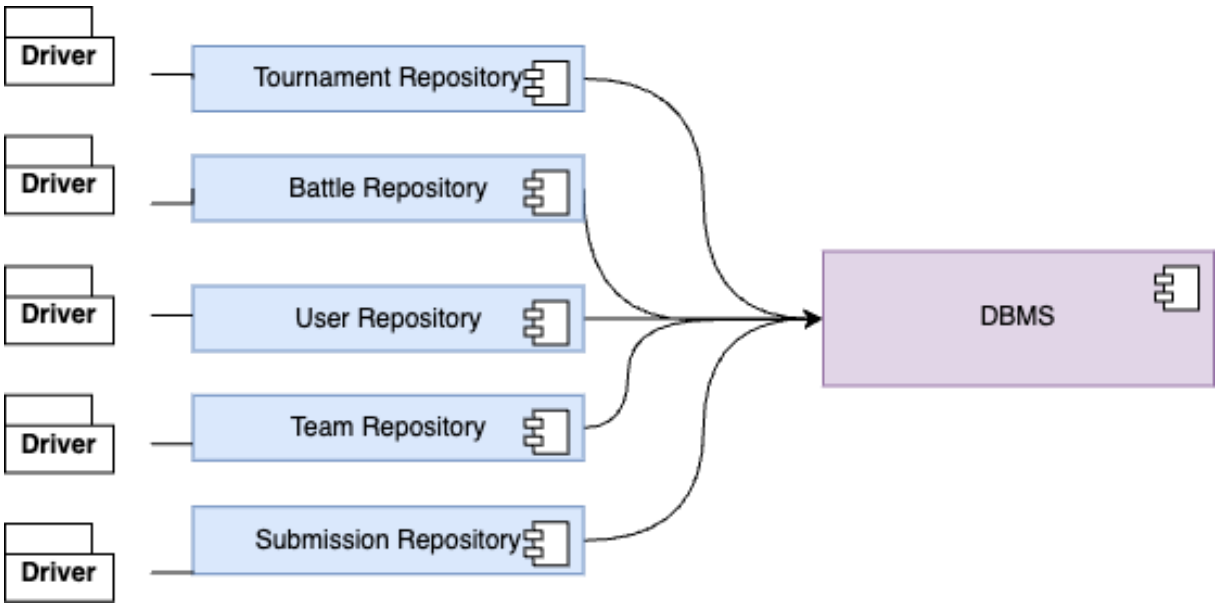

Figure 6: Integration of Repository Components

Notification subsystem is integrated then to be used by other following components. It is very crucial here, after implementing EmailManager and InAppNotificationManager according to the external APIs, NotificationManager is implemented because there exists a dependency between NotificationManager and other two components. Testing is done as explained above with mocking usage of notification according to INotificationManager interface documentation.

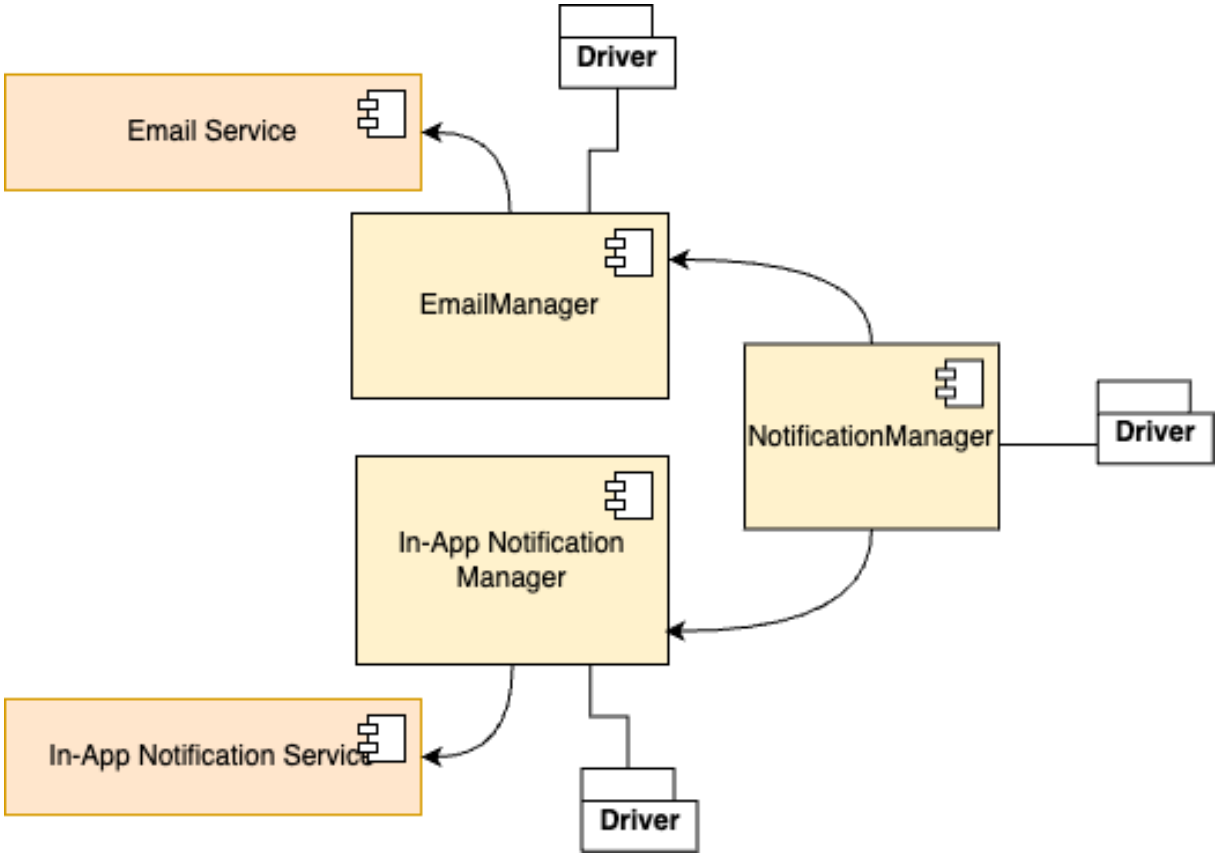

Figure 7: Integration of Notification Subsystem

Authentication Manager is the component responsible for authentication operations. Its integration is done after implementing user repository and notification subsystem. In this stage of development, we are able to implement functions mocked for User Repository and Notification subsystem. Naturally, Unit tests must be passed before and after integration of Authentication Manager Component to these components. At this point, we also have some endpoints to Client Application after integration, so we can do Register, Login, Logout operations end-to-end. Application Server is developing independently, so we will use and test endpoints with the help of some tools such as Postman. At this time, it would be very helpful to start a Postman collection to document the endpoints.

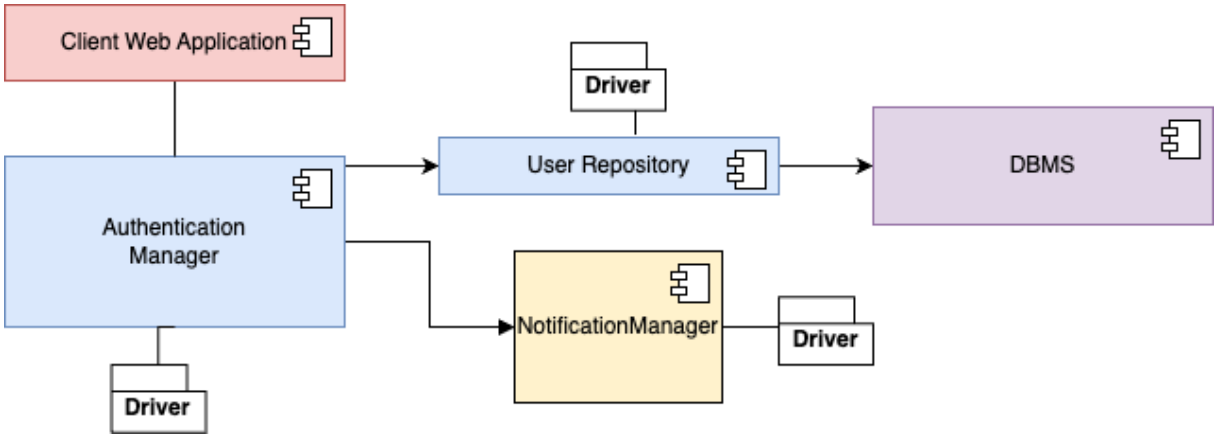

Figure 8: Integration of Authentication Manager Component

Tournament Manager helps us to create an intermediate product because it has very few dependencies. At this stage, we integrated it to developed parts of application. Unit tests must be passed before and after integration. There will be some operations requires to send notification via email and in-app notification. Especially, for Notification Subsystem, implementing this functions must be aligned with unit tests.

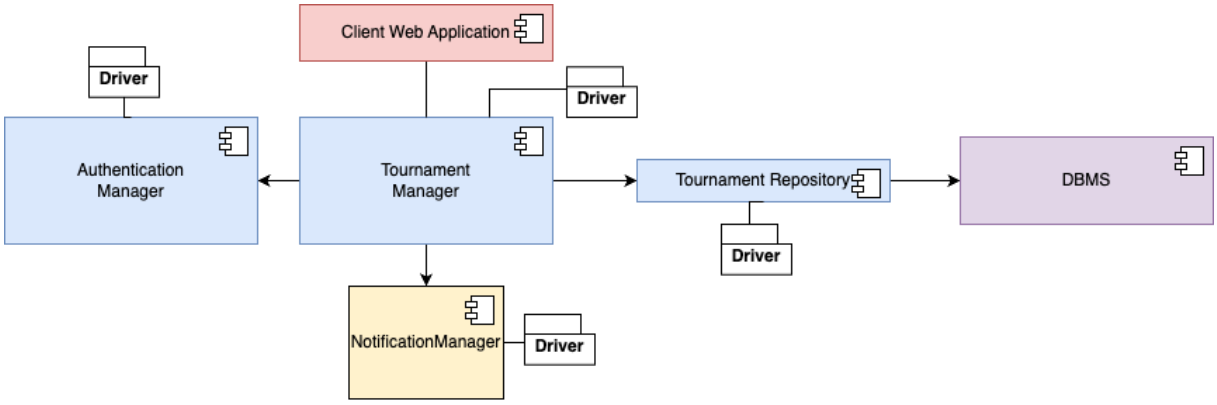

Figure 9: Integration of Tournament Manager Component

File Manager can be developed and integrated to File Storage Service independently. This is also can be done in parallel to other components above.

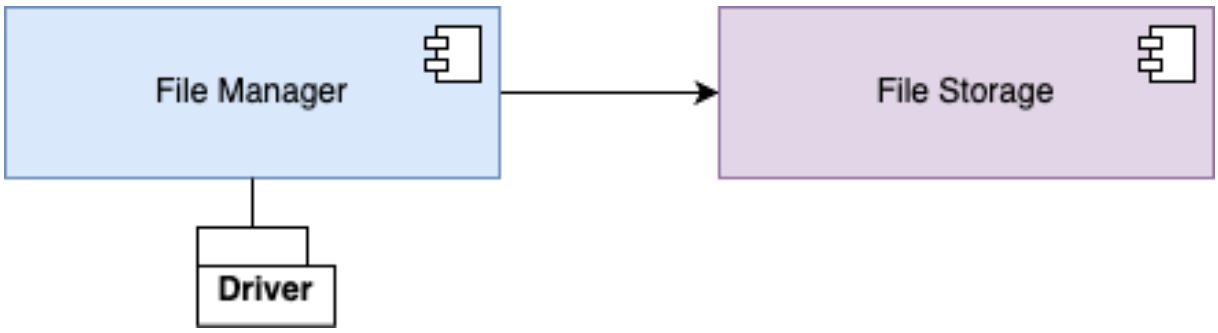

Figure 10: Integration of File Manager Component

Profile Manager is another component to help to create intermediate product with the components Authentication Manager and Tournament Manager. It is integrated to User Repository and File System Manager for the profile specific operations. Also similar to Tournament Manager it uses session information from Authentication Manager and dependent on it. File Manager is integrated to all system via Profile Manager at this stage.

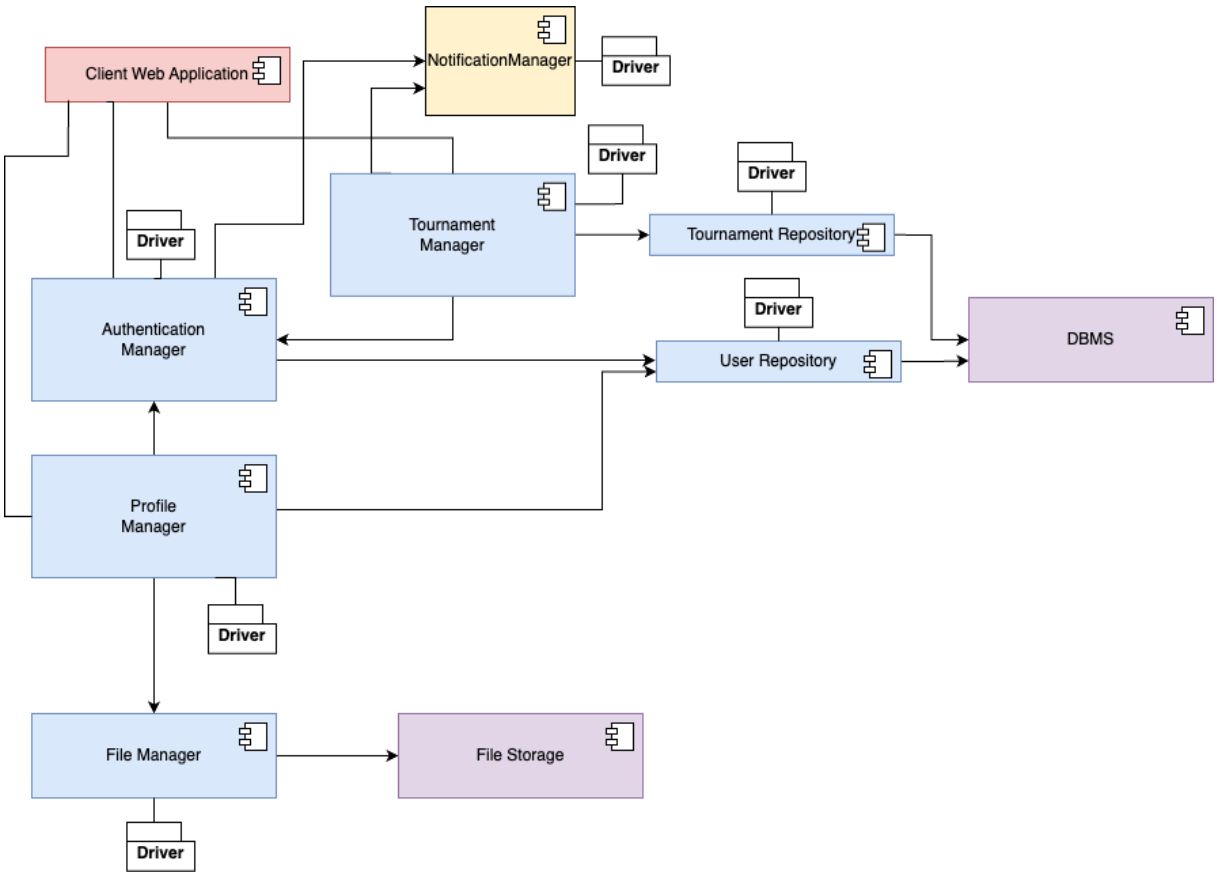

Figure 11: Integration of Profile Manager Component

Scoring Manager and Github Manager can be implemented and integrated in parallel. For Scoring Manager, there are two other components which are responsible to Sandbox Manager and Static Analysis Manager. Firstly, Static Analysis Manager is integrated to 3rd Party API, then it is integrated to Scoring Manager with Sandbox Manager. Also Github Manager is integrated to Github with necessary documentation provided by Github.

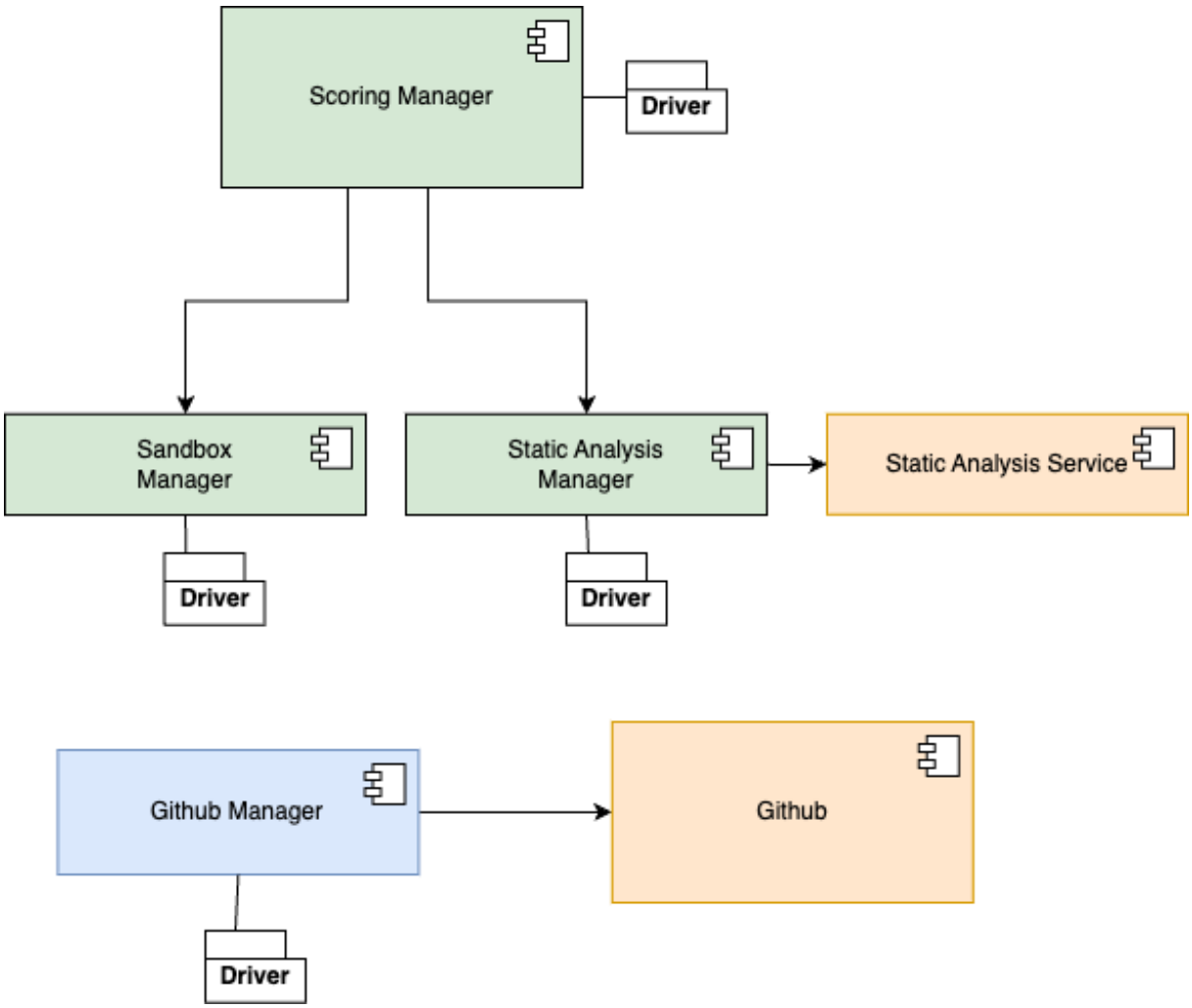

Figure 12: Integration of Scoring and Github Manager

Submission Manager has a dependency over Scoring Manager, Github Manager and File Manager. Its implementation and integration will be held after these components. Also, usage of these components' provided interfaces from Submission Manager should be aligned with unit test written before for these components. Submission Manager is still sperated from the application because mainly Battle Manager uses the interface provided by Submission Manager.

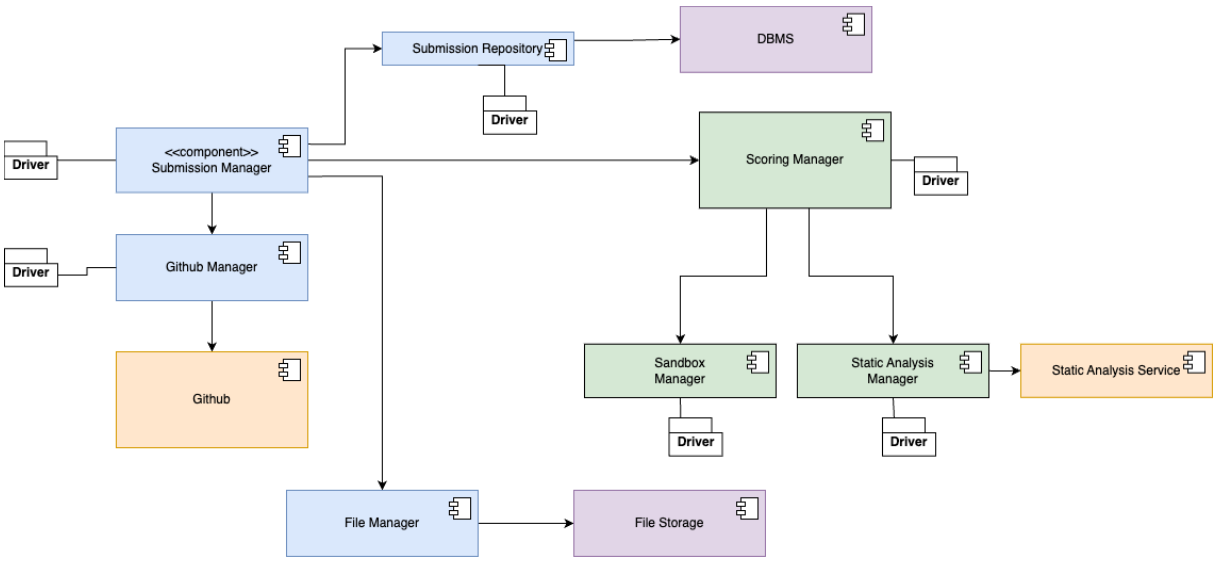

Figure 13: Integration of Submission Manager

At the end Battle Manager, which uses repository components, scoring and Github related actions via Submission Manager and session methods, is implemented and integrated. Of course, Team Manager is firstly implemented and integrated to Battle Manager with the help of some unit test. Then Battle Manager integrated to Submission Manager, Authentication Manager, Repository Components, File Manager and Notification Manager. As you can see, it needs lots of components to be implemented to operate and be integrated to application.

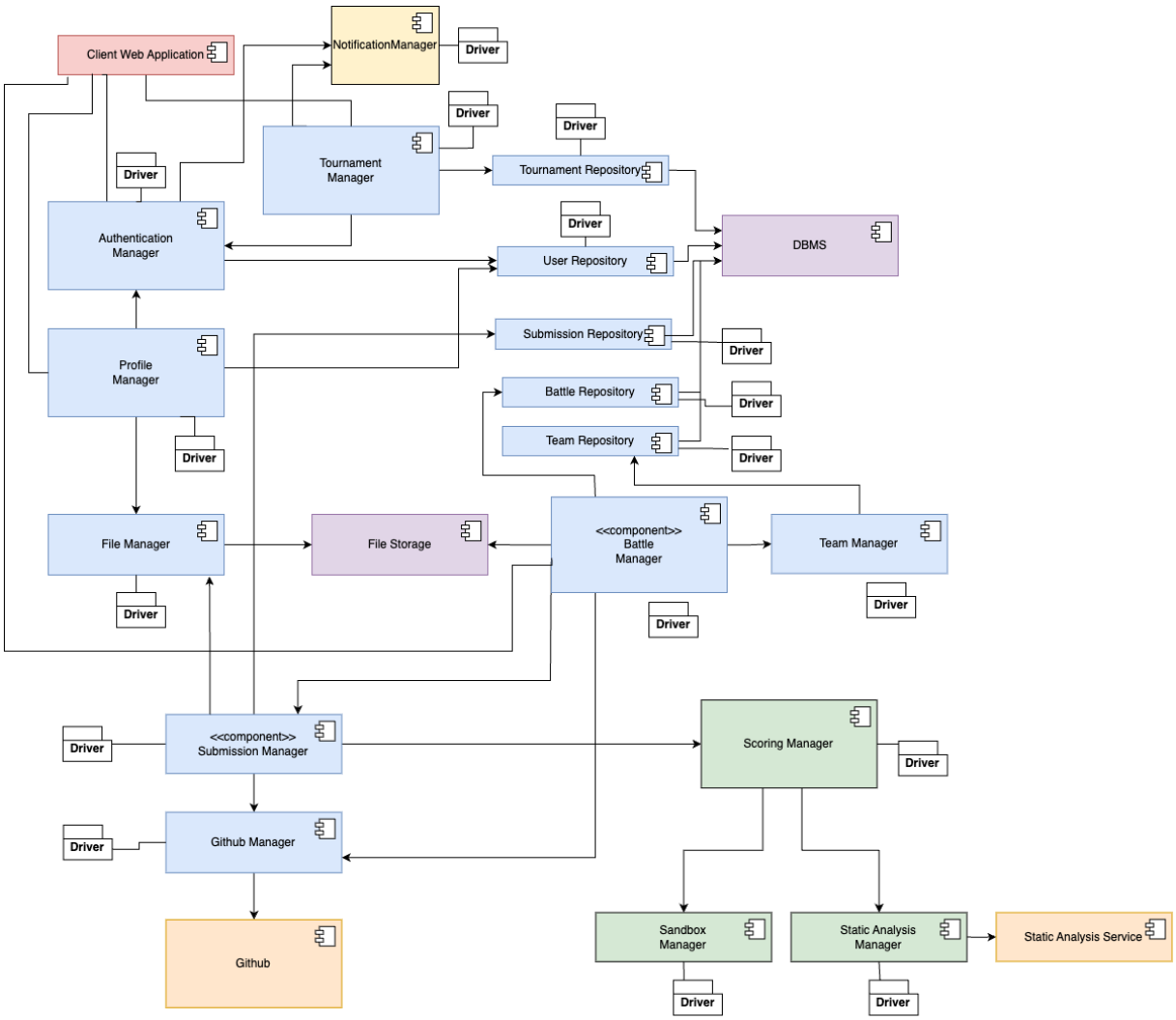

Figure 14: Integration of Battle and Team Manager

5.3 Testing

CodeKataBattle is subjected to a comprehensive list of testing activities, each varying in scope and detail. In the development stage, it is crucial to test every component or module individually to ensure its functionality aligns with expectations.

Given that some components might not function independently, the creation of Driver components becomes necessary. These components simulate the operations of adjacent modules, encompassing both normal and aberrant behaviors, to assess the robustness of each component.

Once individual components have been satisfactorily tested, the next phase involves their integration and the testing of this integrated setup. The CodeKataBattle system employs a blend of bottom-up and thread strategies for this process, as previously discussed.

To comply with requriements discussed in RASD, system must be tests as a whole. This final testing phase confirms that all features are correctly developed and meet both functional and nonfunctional requirements. System testing, being a black-box technique, should involve not just the developers but also the stakeholders.

1. The system should be tested to understand whether all requirements are fulfilled or not.

2. Also, there can be some performance problem related to some inefficient code, algorithm or a bottleneck caused by a design choice. This kind of performance testing is important to simulate workload or traffic in order to find weak aspects of the application in terms of speed, memory, resource utilization etc.

3. In some cases, user can behave differently from stakeholders or developers. To understand such cases, it can be helpful to test user actions with different scenarios, devices etc.

4. Moreover, it is helpful to make stress testing to see system's ability to recover from failures. The objective here is to ensure the system's resilience by pushing it beyond its normal operational capacities and observing how it recovers from failures. This involves overwhelming the system's resources or depriving it of them to test its limits and recovery capabilities.

Whenever new features are introduced to the system, testing is essential to identify any bugs. Additionally, the testing methods outlined above must be employed to ensure the system's proper functioning throughout the implementation phase. This early and continuous testing is crucial for prompt bug detection.

Feedback from users and stakeholders is equally important during system development. Initially, stakeholders should be briefed on the planned functionalities to ascertain if the system aligns with their expectations. Also, intermediate versions of the system should be shared with them for feedback on any issues and see the satisfaction.

This process of receiving regular feedback is vital for the validation of the system throughout its development. It enables developers to promptly identify if the system meets the intended requirements and customer expectations. If discrepancies are found between the system's current state and the stakeholders' expectations, developers can make necessary adjustments. This approach not only ensures the system's relevance and efficacy but also aligns its development closely with user needs and preferences.

6 Effort Spent

References

