



POLITECNICO
MILANO 1863

CodeKataBattle - Annechini Alessandro, Filippi Nicole, Fiorentini Riccardo

Design Document

Deliverable:	DD
Title:	Design Document
Authors:	Annechini Alessandro, Filippi Nicole, Fiorentini Riccardo
Version:	1.0
Date:	20-December-2023
Download page:	https://github.com/RiccardoFiorentini/AnnechiniFilippiFiorentini

Contents

1	Introduction	3
1.1	Purpose	3
1.2	Scope	3
1.3	Definitions, Acronyms, Abbreviations	4
1.3.1	Definitions	4
1.3.2	Acronyms	4
1.3.3	Abbreviations	5
1.4	Revision history	5
1.5	Reference Documents	5
1.6	Document Structure	5
2	Architectural Design	7
2.1	Overview	7
2.2	Component View	11
2.3	Deployment View	14
2.4	Runtime View	17
2.5	Component Interfaces	32
2.6	Selected Architectural Styles and Patterns	36
2.7	Other Design Decisions	38
3	User Interface Design	39
4	Requirements Traceability	46
5	Implementation, Integration and Test Plan	53
5.1	Implementation plan	53
5.2	Component integration and testing	55
5.3	System testing	60
6	Effort Spent	62
7	References	63

1 Introduction

1.1 Purpose

This Design Document (DD) is the continuation of the Requirements Analysis and Specification Document (RASD). The purpose of this Design Document is to translate the conceptual framework defined in the RASD into a physical, robust system architecture. It serves as a guide for software developers, system architects, and stakeholders involved in the construction and evolution of the CKB platform. By examining the system's internal structure, modules, and data flow, this document elucidates the technical details necessary for turning the features into functional components.

This DD not only elaborates the architectural choices made during the design phase but also explains the reasoning behind these decisions. It addresses how the system will handle key functionalities, ensuring scalability, maintainability, and security. In addition, the document explores the specifics of data storage, processing, and the integration of external tools to provide a general view of the technical basis.

In few words, the Design Document acts as a bridge between the conceptualization of the CodeKataBattle platform and its effective realization. Developers and technical stakeholders can use this document to comprehend the system's architecture, enabling them to craft a software that aligns with the defined requirements.

1.2 Scope

The CodeKataBattle (CKB) platform is a software that redefines collaborative coding experiences for students and educators. CKB offers coding competitions termed "Code Kata Battles", providing skill development and promoting a healthy competition among student teams.

This platform's principal objective is to facilitate collaborative learning in software development. Educators have the capability to craft tournaments in which battles are added by that tournament's creator and one or more other educators, whose permission needs to be granted. Students subscribe to tournament to join contained battles, they can do that in teams or by themselves. Code Kata Battles are real-time coding challenges constituted by a software project that includes also test cases and build automation scripts, they also have a registration deadline, a submission deadline, a description and some constraints about participating teams.

CKB real-time collaboration feature enables students to collectively address coding exercises, but to do this all users need a GitHub account to work on the project. For every battle a GitHub repository is created by the platform, and the link is distributed to all subscribed teams, students can work on the project pushing a new commit into the main branch of battle's repository, and GitHub Action immediately informs the CKB platform through proper API calls. The automated evaluation mechanisms provided by the system, evaluate factors like functional correctness, timeliness, and code quality, providing instant feedback to participants, but in addition to that after the battles' submission deadline educators can manually evaluate every team's work, ensuring a comprehensive valuation

of students' performances.

In addition, the platform not only notifies participants about new battles' creation and new tournaments' creation, but also about the final ranking at the conclusion of battles. The tournament's ranking is the sum of all battles' score, is important that every user subscribed to the platform can see not just the list of all tournaments, but also their ranking.

Another important offered functionality is the possibility for the tournament's creator to add some gamification badges, to further reward deserving students. Every badge has a title and a set of rules that need to be respected to obtain the badge. After gaining a badge, this is visible on student's profile by every other user subscribed to the platform. These are the principal goals and key functionalities of the CKB platform, but an in-depth exploration of its design and implementation will follow in this document.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- Students: all students who are subscribed to the platform to learn and participate in tournaments. They need a unique email to subscribe
- Educators: all educators who are subscribed to the platform to create tournaments and battles. They need a unique email to subscribe
- Users: all students and educators who are subscribed to the platform
- Kata: an exercise in karate where you repeat a form many, many times, making little improvements in each. Code Kata is an attempt to bring this element of practice to software development
- Code kata: exercise created by an educator, composed of a description and software components, including test cases and building automation scripts
- Battle: a code kata battle
- Platform: the CodeKataBattle platform, i.e. a system that provides a set of tools, features, and services
- Business Logic: refers to the set of rules, processes, and workflows that define how a system operates to meet its objectives and fulfill its purpose. Here comprehend the core functionalities and rules that govern the interactions, evaluation, and scoring mechanisms within the system

1.3.2 Acronyms

- RASD: Requirement Analysis and Specification Document
- DD: Design Document

- CKB: CodeKataBattle
- GH: GitHub
- GHA: GitHub Action
- DB: Database
- DBMS: Database Management System
- UI: User Interface
- WAF: Web Application Firewall
- IDS: Intrusion Detection System

1.3.3 Abbreviations

- Rn : Requirement number n
- F_n : Functionality number n

1.4 Revision history

- Version 1.0 - 30/12/2023
- Version 2.0 - 26/01/2024
 - Added three system runtime views

1.5 Reference Documents

This document is based on:

- The specification of the RASD and DD assignment given by professors Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at Politecnico di Milano, A.Y. 2023-2024
- Slides of Software Engineering 2 course on WeBeep
- DD sample from A.Y. 2022-2023

1.6 Document Structure

The document is made of seven chapters which describe the architecture of the system, in order to support developers who will implement it.

The first chapter is the introduction of the document, which function is to highlight the purpose and the scope of the document, that briefly recalls RASD introduced concepts; it also gives important information such as definitions, acronyms and abbreviations used within the document, the revision history and the referenced documents.

The second chapter contains a description of the architecture of the system, describing its component view, deployment view, runtime view and the components' interfaces. It talks about selected architectural patterns used and other design decisions.

The third chapter is about user interface design: mockups are provided to give an overview of the user interfaces of the system, coherently to what was already written in the RASD document but adding more details about links between them.

The fourth chapter relates to requirements traceability, it maps the requirements previously identified and defined in the RASD document to the design elements defined in this document, as proof that the design decisions are taken coherently to the requirements, and so the goals can be fulfilled by the designed system.

The fifth chapter describes implementation, integration and test plan that programmers need to respect to develop a correct system: specific subcomponents, their integration and testing is presented here.

The sixth chapter is a report containing the effort spent by each group member.

Finally, the seventh and last chapter contains all the documents' references and softwares used for the drafting of the document.

2 Architectural Design

2.1 Overview

Here the architectural elements that compose the system are explained with their interaction. A description of the replication mechanism chosen in order to make the system distributed is also given. The architectural framework adopted by the CKB platform needs to be robust and scalable. A three-tier application model is used, since it guarantees several of the nonfunctional requirements previously described in the RASD, such as reliability, since having several components which perform the same activity renders the system fault-tolerant, and maintainability, because having a system divided into three layers, any of which independent from the other ones, ensures atomicity between their functionalities. In addition, each tier runs on its own infrastructure, so can be developed individually and simultaneously with the others, giving flexibility and scalability to the system. This architectural paradigm consists of three tiers: the presentation tier, the application logic tier, and the data storage tier.

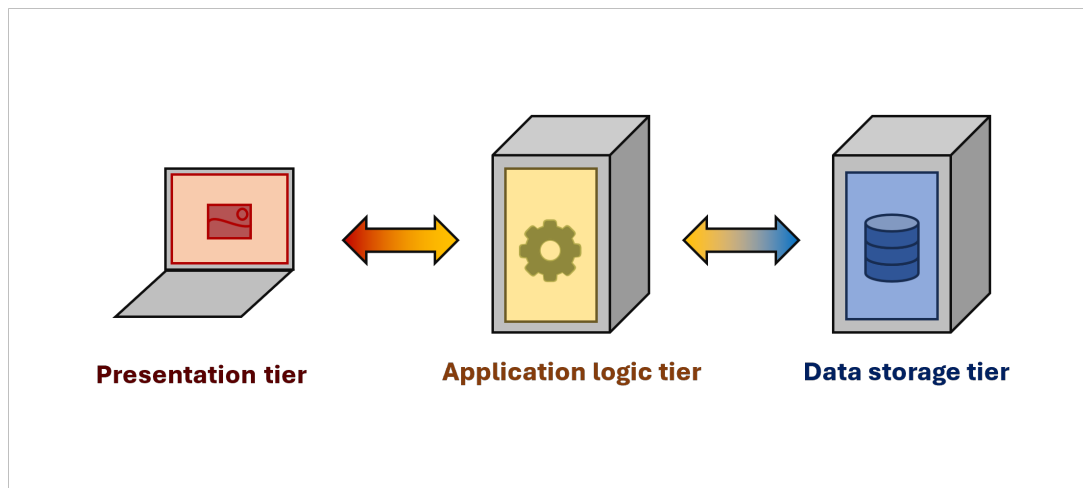


Figure 1: Visual representation of a three-tier architecture

1. **Presentation Tier:** its functions are providing the user interface, and handling user interactions. This tier is client side and interacts directly within users' browsers. All web-based interfaces are executed on the client-side, allowing educators and students to interact with the CKB platform. Since the computation is done locally on the client, the user experience will be efficient and responsive.
2. **Application Logic Tier:** it's positioned on the server-side, and its function is encapsulating the core functionalities of the CKB platform. It contains the business logic, the code execution engines, facilitating communication between the users and the system.
3. **Data Storage Tier:** it's on server-side and it's the foundation of the architecture, where platform's relational databases store, manage and retrieve all necessary

information. To optimize data accessibility and ensure data integrity, a database management system (DBMS) is employed. For security measures, encrypted connections and access controls are implemented, to safeguard users' sensitive information and system data, and against unauthorized access and data breaches.

The CKB platform can be accessed only via web browser, which interacts with the presentation tier. On the server side there are two components situated in the application tier (application server and web server) and one component situated in the data tier (database server). The web server handles the HTTP requests sent by browsers and returns the correct web pages in response, it also need to communicate with the application server via suitable API calls if, to satisfy such requests are necessary some data from the database, this because the web server and the DB are not directly connected.

The application server communicates with the database server that retrieves all the information about students, educators, battles, tournaments, badges, deadlines and so on from the DB. This because the application tier must be able to add, delete or modify data in the data tier using API calls.

In addition, the application server also needs to communicate with an Email service provider, to send e-mails to users, and with the GitHub Action service via some API call, to provide the code written by teams to the system, which evaluates it and updates immediately the ranking.

In the end, the database server acts as an intermediary between the application server and the DBMS, managing connections, processing queries, and ensuring the overall efficiency and reliability of database operations.

Since the system is distributed, so are the web servers, the application servers and the database servers, for this reason three load balancers are inserted into the architecture to dispatch the requests to web servers and application servers: all the requests coming from web browsers are captured by a load balancer that redirects the request to one of the web servers available to handle it. Then the web server manipulates the request and forwards it to the second load balancer, that dispatches requests from web servers to the application server that better elaborates the request in terms of time.

The database should be clustered so that consistency is guaranteed, and the performances are sufficient to fulfill a large number of queries in a reasonable time. The replicated database servers' main function is to manage all data relevant to the application. The third load balancer is interposed between application servers and database servers.

To guarantee a good level of security, a firewall is installed between every tier and between the external network (internet) and the internal network. Its functionalities are monitor, filter, and control incoming and outgoing network traffic based on predetermined and personalized security rules; it works as barriers between a trusted internal network and untrusted external networks that control the flow of traffic and prevent unauthorized access, protecting against various cyber threats and attacks. For other security measures, a Web Application Firewall (WAF) and an Intrusion Detection System (IDS) are inte-

grated into the application tier, the first is necessary to safeguard against common web application threats and enhance overall security, while the second monitors and analyzes system activities, detecting and responding to suspicious behavior that may compromise the platform's integrity.

In the following sections there is a presentation about the components of the systems and their interactions, completed with diagrams and schemes. Then, given the description of the components and their characteristics, an exhaustive discussion concerning the run time specifications of the system are presented with the use of sequence diagrams. Finally, a dependency analysis of the component interfaces is shown.

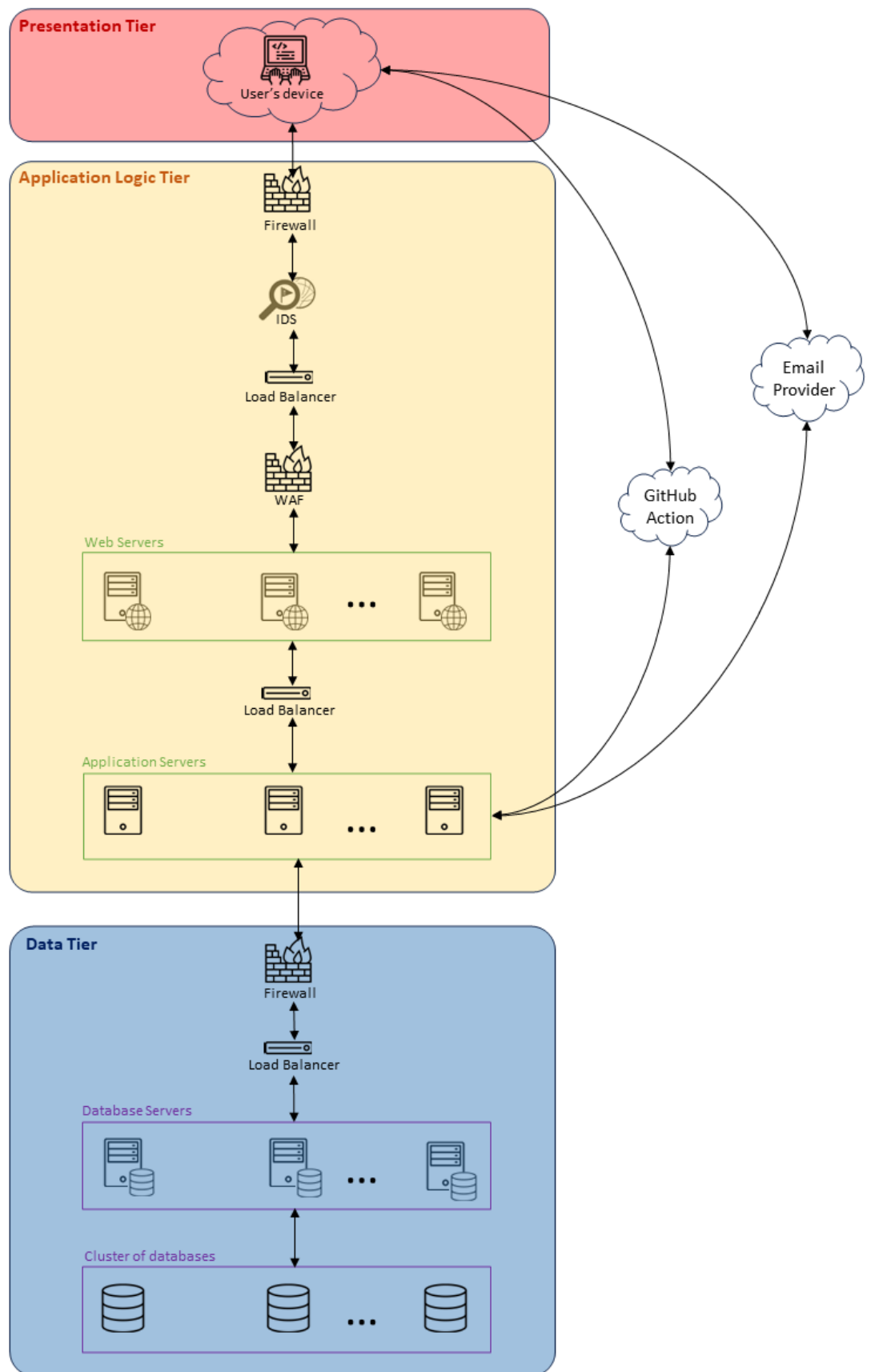


Figure 2: System Architecture

2.2 Component View

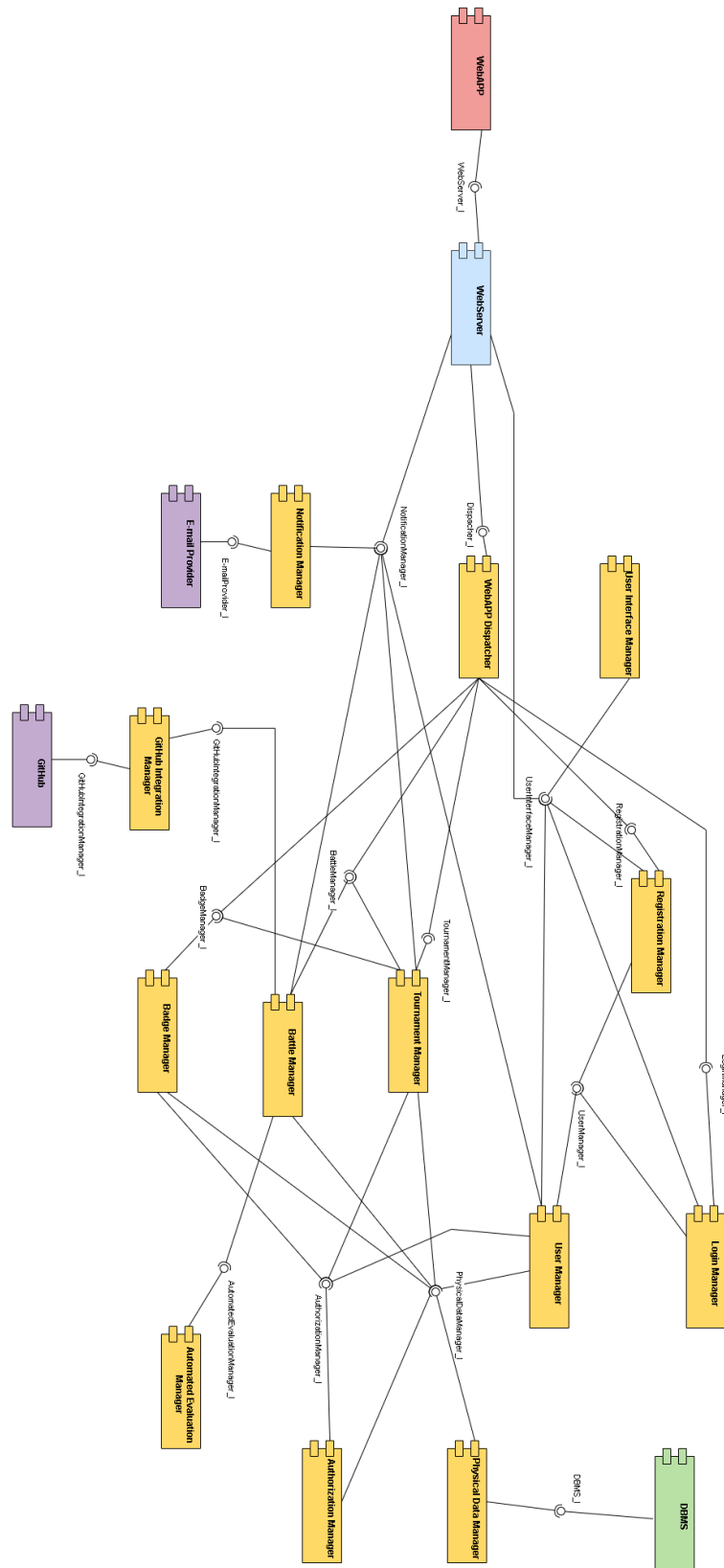


Figure 3: Component diagram

- **Web App Dispatcher:**

Handles the incoming requests, forwarding them to the correct component in order to subdivide the workload and correctly elaborate each request. Since all of the incoming requests need to be analyzed and dispatched to the correct manager, this component is replicated in the physical system in order to handle important loads of requests.

- **User Manager:**

Handles students' and educators' data such as login parameters, profile description, badges, and more. When a new user is created, its data are collected from this component. When a user wants to log into the system, the User Manager collects the relevant data for the authentication of the user. When the profile of a student or an educator is required, this component's task is to retrieve all the required information about the requested user.

- **Registration Manager:**

Handles user registration process. Checks the validity of the inserted data, imposing password complexity policies and preventing users from creating multiple accounts with the same information. If all the provided information is correct, it is stored in a secure manner.

- **Login Manager:**

Handles user login and authentication processes, keeping track of unusual behaviors, and correctly handles sensitive data. The implementation of secure channels for the exchange of sensitive data even within the system is important to avoid data breaches: the Login Manager keeps this and all the other important security aspects into account.

- **Authorization Manager:**

Handles user authorization processes. Manages user roles (Students, Educators) and their respective permissions. For example, when an educator wants to create a new battle within a tournament, this component checks whether permission has been granted from the creator of the tournament. When specific data about a user or about a battle are requested, the Authorization Manager ensures that the issuer is allowed to read the desired data.

- **Tournament Manager:**

Manages the creation, updating, and closure of tournaments. Includes components for handling registrations and scoring. Upon creation of a new tournament by an educator, the Tournament Manager handles all the necessary changes, providing continuously updated battles list and ranking. When a tournament is closed, the Tournament Manager maintains all the relevant information.

- **Battle Manager:**

Manages the creation, updating, and closure of battles. Includes components for handling registrations, submissions, and scoring. When a new battle is created, the Battle Manager keeps track of the registration issued by the student, and once the battle starts it works synergistically with the GitHub Integration Manager and Automated Evaluation Manager to provide a real-time experience to the user. Once the battle closes, the Battle Manager awaits the educator's evaluations if necessary, providing all the relevant information to the Tournament Manager once the final ranking has been established.

- **GitHub Integration Manager:**

Facilitates the integration with GitHub for repository creation, forking, and workflow setup. Manages the communication between the CKB platform and GitHub.

- **Automated Evaluation Manager:**

This component handles the ongoing evaluation of student submissions using build automation and testing tools. This component is designated to provide a real-time evaluation of student solutions to create a continuously updated battle rank. If the educator of the battle chooses not to manually evaluate final submissions, once the battle closes this component automatically evaluates the last solution issued by each team.

- **User Interface (UI) Manager:**

Responsible for presenting information to users and capturing their interactions. Includes components for displaying tournament and battle information, user profiles, and submission interfaces. This component is instantiated in order to be able to implement new features inside the user interface without the need of structural changes in the system.

- **Badge Manager:**

Manages the creation and assignment of badges within the context of tournaments. Handles the association of badges with specific rules and user achievements. When a tournament is closed, the tool checks the eligibility of each student registered to the tournament, collecting data from all the battles created within the tournament and assigning each badge to deserving students.

- **Notification Manager:**

Manages the generation and delivery of notifications to users about new tournaments, battles, and closures. Notifications can be either visualized on the web app or sent via email to the users.

- **Physical Data Manager:**

Stores and retrieves data related to users, tournaments, battles, submissions, and other relevant information. It is responsible for the communication with the DBMS, handling the topology of distributed and replicated databases, keeping track of the

accesses and modifications performed by different processes for maintainability and security reasons.

2.3 Deployment View

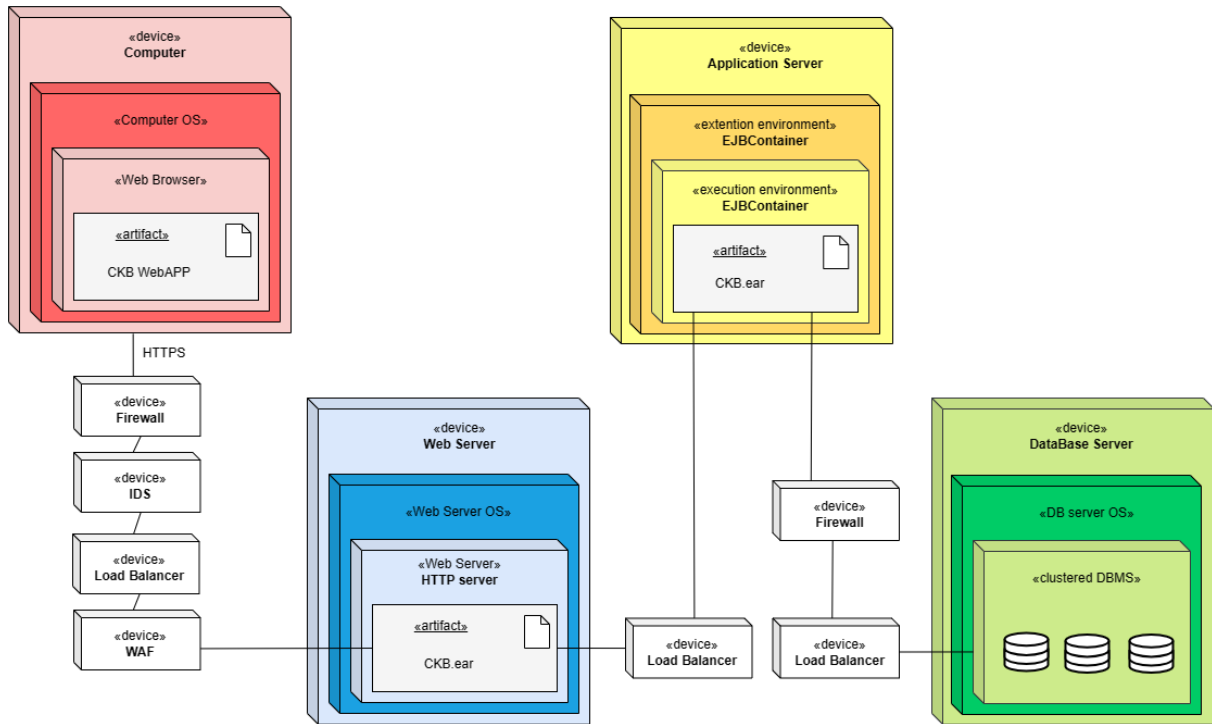


Figure 4: Deployment view of the system

- **Client devices (Computers with web browsers):**

The physical device required to access the CKB platform is a computer with the capability of connecting to the internet. No specific applications have to be installed on the device in order to access the platform apart from a generic web browser, required for navigating the web and reaching the CKB web page.

- **Firewall:**

It acts as a crucial line of defense for a network, employing a range of functionalities to safeguard against potential threats and unauthorized access, monitoring all incoming and outgoing network traffic within the system, distinguishing between harmless data and potentially harmful elements. It acts as a gatekeeper, checking all data packets based on a set of predetermined or personalized security rules: it evaluates the characteristics of each packet, such as source and destination addresses, port numbers, and protocol types. By doing so, it makes informed decisions about whether to permit or deny passage to specific packets and so works as a barrier between the trusted internal network and the untrusted external networks. It's positioned between every tier, this strategic positioning allows the firewall to exercise control over the flow of traffic, ensuring that only legitimate and authorized packets

pass through. In addition, it establishes a secure perimeter around the internal network, actively managing the access permissions. It acts as a shield against various cyber threats, including malware, phishing attacks, and intrusion attempts.

- **Intrusion Detection System (IDS):**

It's a fundamental component in cybersecurity, designed to provide continuous monitoring, analysis, and response to safeguard the integrity of a platform. Its primary function is to actively monitor and analyze system activities, controlling users' behavior, applications, and network traffic in real-time, with the goal to identify any deviations from established patterns that could signal potential security threats or unauthorized accesses. The monitoring is done through the observation of data sources such as network traffic logs, system logs, user activities, and application behaviors, aggregating and correlating data from these sources. It also gains a comprehensive knowledge of the typical patterns of operation of the system.

The analysis phase employs advanced algorithms to detect anomalies or patterns indicative of malicious activities: it compares the observed behavior against predefined profiles of normal or acceptable behavior, and any deviation triggers an alert, signaling potential security incidents.

It is also equipped with responsive capabilities, in fact after detecting suspicious behavior, it initiates appropriate responses to mitigate the threat. These responses can go from generating notifications for further investigation to taking immediate action to block or quarantine malicious entities.

- **Load Balancer:**

Its function is to distribute tasks and requests efficiently across multiple servers to optimize resource utilization, enhance performance maximizing the utilization of computing resources, and ensure high availability. This distribution is done in a way that prevents any single server from becoming overloaded, to prevent performance degradation or potential system downtime. It evenly distributes not just processing loads, but also memory usage, and network bandwidth among the servers. It also regularly monitors the health and status of individual servers and helps with system's availability by directing traffic away from servers that may have issues or are temporarily unavailable, directing traffic to healthy servers, ensuring uninterrupted service for users. It also facilitates horizontal scaling by efficiently adding or removing servers from the system.

- **Web Application Firewall (WAF):**

It is an important component within the system's security infrastructure, its purpose is to provide defense against a wide range of common web application threats, vulnerabilities, and attacks, safeguarding the integrity, confidentiality, and availability of user data and system resources. It is designed to identify, analyze, and mitigate common web application attacks such as SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF), and other injection-based vulnerabilities, performing real-time analysis of HTTP requests and responses and identifying patterns

indicative of potential threats, it prevents malicious users from exploiting vulnerabilities in the platform. Network administrators adjust rules and policies dynamically, such as specifying what type of requests are allowed or denied and customizing the security strength of the system. To prevent unauthorized access and data leakage, it controls incoming and outgoing traffic for sensitive information such as personal information and other confidential data, and it also can mask, block, or log sensitive data to maintain confidentiality. Another important function is the safeguarding of user sessions by detecting and preventing session-related attacks that could be employed to compromise user sessions and access unauthorized areas of the platform. Finally, it generates logs and reports that can be valuable to demonstrate platform's adherence to security best practices.

- **Web Server:**

It's part of system's architecture, responsible for managing the communication between clients (browsers) and the application server, it has a fundamental role in receiving and handling HTTP requests initiated by users' browsers, ensuring the correct delivery of web pages, in fact it works as the entry point within the platform. To provide dynamic content and fulfill user requests that require data from the database, it communicates with the application server via API calls, ensuring that served web pages are up-to-date and reflect the most recent information stored in the system: it acts as an intermediary between the client-side and back-end components such as the database, abstracting the complexities of back-end operations and preventing direct connections, enhancing security and modularity within the system. It also implements caching strategies to optimize content delivery, in fact frequently accessed static content can be cached to reduce latency and increase the responsiveness of the platform.

- **Application Server:**

It works as the core computational engine of the system and is responsible for executing business logic, managing data flow, and facilitating communication between various components, ensuring the dynamic and responsive functioning of the system. One of its primary responsibilities is to communicate with the database server through API calls for retrieving, adding, deleting, or modifying data stored in the database, ensuring dynamic updates, and maintaining data integrity. It also communicates with third-party tools and services through API calls.

- **Database Server:**

It is an intermediary between the application server and the Database Management System (DBMS). This component manages and optimizes interactions with the underlying database, providing an efficient data storage and retrieval process. It handles the establishment, maintenance, and termination of connections, ensuring that the application server can communicate with the database securely and reliably. It acts as a gateway for queries: it processes SQL queries received from the application server and optimizes them, enhancing the overall performance. It

also implements concurrency control mechanisms, since multiple requests may occur simultaneously, ensuring that transactions are executed in a controlled manner, preventing conflicts, and maintaining the consistency of the database. In addition, it enforces data integrity constraints defined in the database schema: it validates incoming data to prevent inconsistencies and errors in the stored information. It has a caching mechanism to store frequently accessed data temporarily to reduce latency and improve response times. Finally, it facilitates the creation of database backups at regular intervals and assists restoring data in the event of data loss or corruption.

2.4 Runtime View

The following sequence diagrams represent the runtime view of the system applied to the most important use cases. The purpose of these diagrams is to illustrate the interaction between the components described in the previous sections: for the precise details about the interfaces offered by each component, refer to the next section.

• Student Registration:

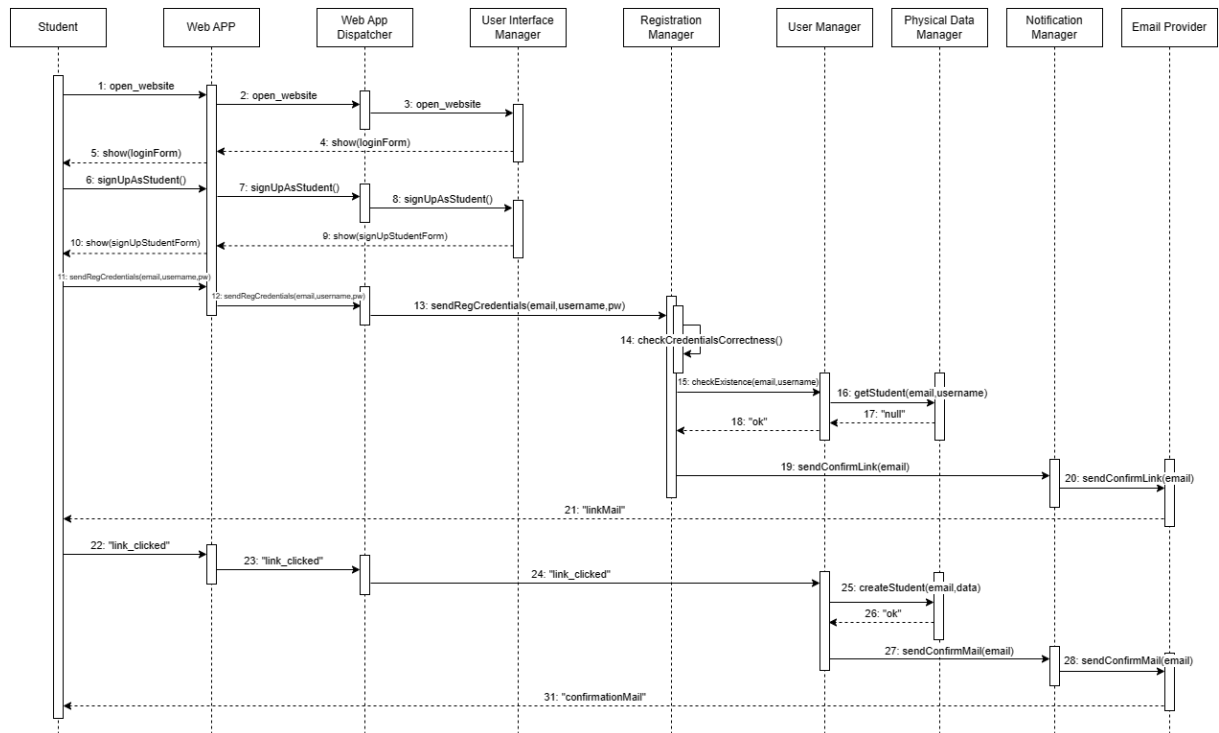


Figure 5: Student registration runtime view

This sequence diagram shows all the actions performed when a student registers to the system. When the "sign up as student" page has been created by the User Interface Manager and shown to the student via the Web Application, the student inserts their credentials and sends them to the Registration Manager. The Registration Manager checks the correctness of the incoming data, such as the validity of the

email or the length of the password, and then queries the User Manager to check if another student with the same credential exists. When the validity of the data has been assessed, an email containing the confirmation link is sent to the submitted email address. When the link is clicked, the system confirms the insertion of the new student, and a final email is sent to the user.

• Educator Registration:

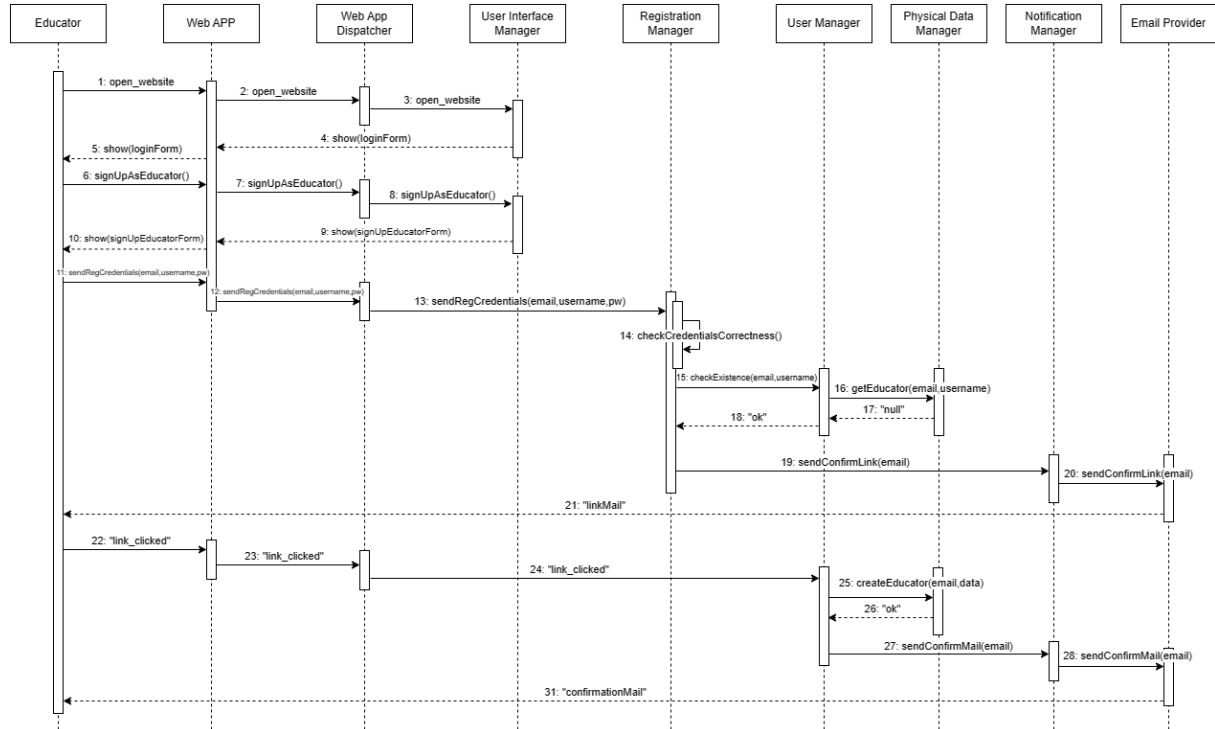


Figure 6: Educator registration runtime view

This sequence diagram shows all the actions performed when an educator registers to the system. When the "sign up as educator" page has been created by the User Interface Manager and shown to the educator via the Web Application, the educator inserts their credentials and sends them to the Registration Manager. The Registration Manager checks the correctness of the incoming data, such as the validity of the email or the length of the password, and then queries the User Manager to check if another educator with the same credential exists. When the validity of the data has been assessed, an email containing the confirmation link is sent to the submitted email address. When the link is clicked, the system confirms the insertion of the new educator, and a final email is sent to the user.

• Student Login:

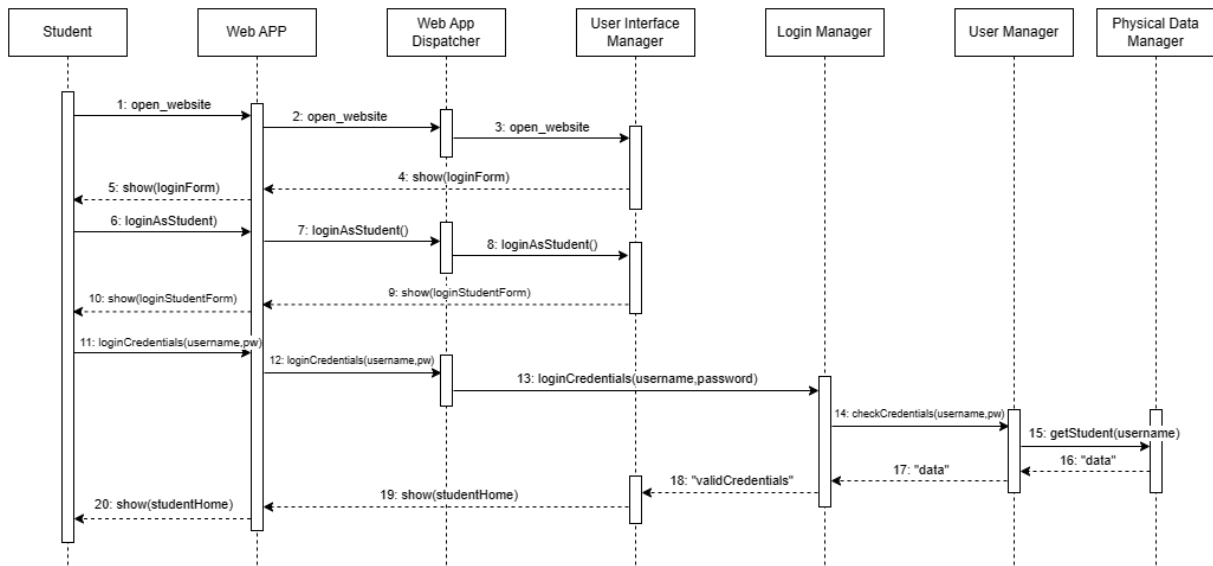


Figure 7: Student login runtime view

This sequence diagram shows all the actions performed when a student logs into the system. Once the student has selected "login as student", its credentials are sent to the Login Manager, which is responsible for querying the User Manager, collecting the data regarding the student corresponding to the inserted credentials and checking whether the authentication data are correct. If the check is positive, the User Interface prepares the home page, which is displayed by the Web Application to indicate the correct access to the profile.

• Educator Login:

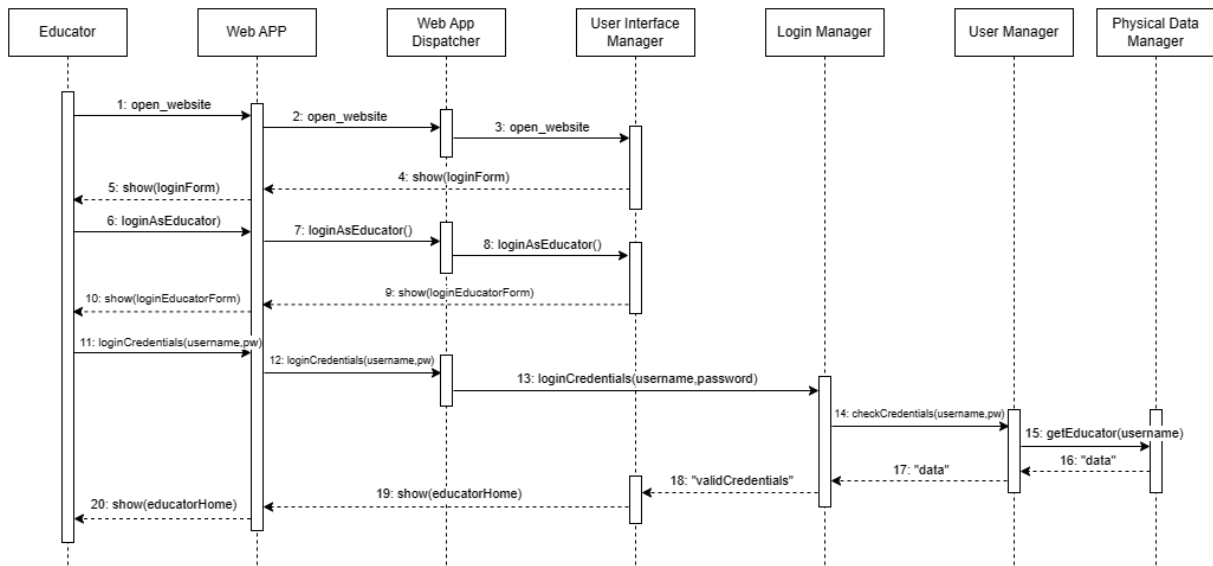


Figure 8: Educator login runtime view

This sequence diagram shows all the actions performed when an educator logs into the system. Once the educator has selected "login as educator", its credentials are sent to the Login Manager, which is responsible for querying the User Manager, collecting the data regarding the educator corresponding to the inserted credentials and checking whether the authentication data are correct. If the check is positive, the User Interface prepares the home page, which is displayed by the Web Application to indicate the correct access to the profile.

• Tournament Creation:

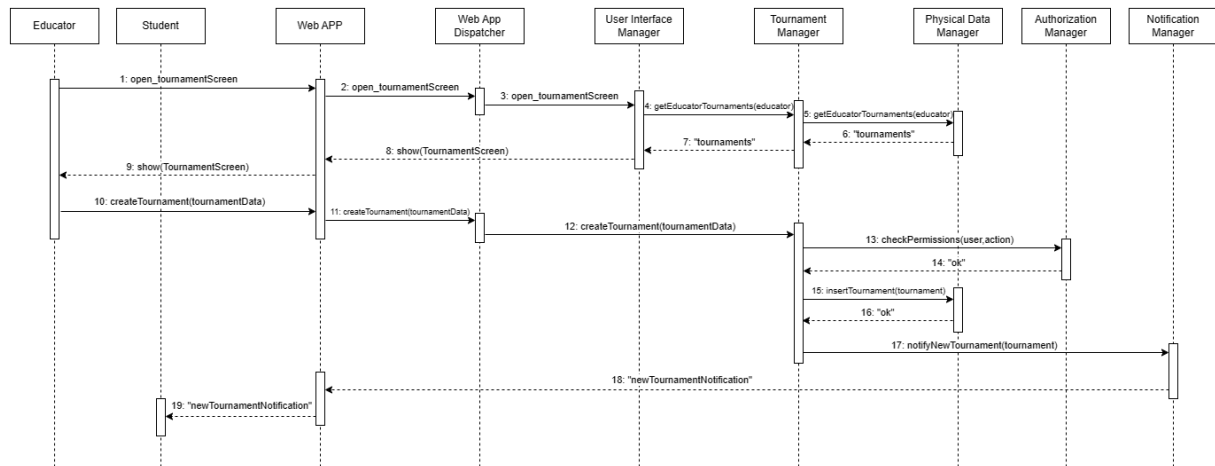


Figure 9: Tournament creation runtime view

This sequence diagram shows all the actions performed when an educator creates a new tournament. When the "tournaments" page has been displayed, the educator inserts all the required data for the creation of the tournament. When the educator confirms the action, the tournament's data is sent to the Tournament Manager. After a check by the Authorization Manager (to ensure that the user has permission to perform this action), the tournament is inserted into the database by the Physical Data Manager and a notification is sent to all the students.

- Add Battle Permission Granting:

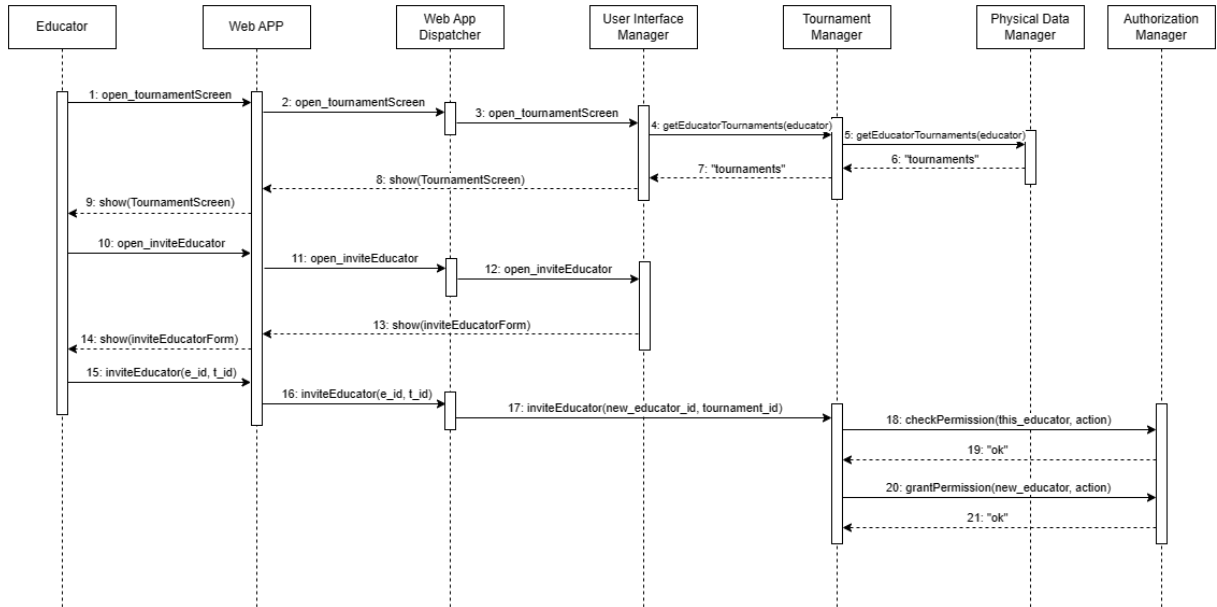


Figure 10: Add battle permission granting runtime view

This sequence diagram shows all the actions performed when an educator grants another educator the permission to create battles within a tournament. When the designated form has been displayed, the educator insert the other educator's id (either the username or the email address), which is then sent to the Tournament Manager. The Tournament Manager first checks if the issuer of the request has the granting permission and then signals the Authorization Manager that a new educator can create battles within the specific tournament.

• Battle Creation:

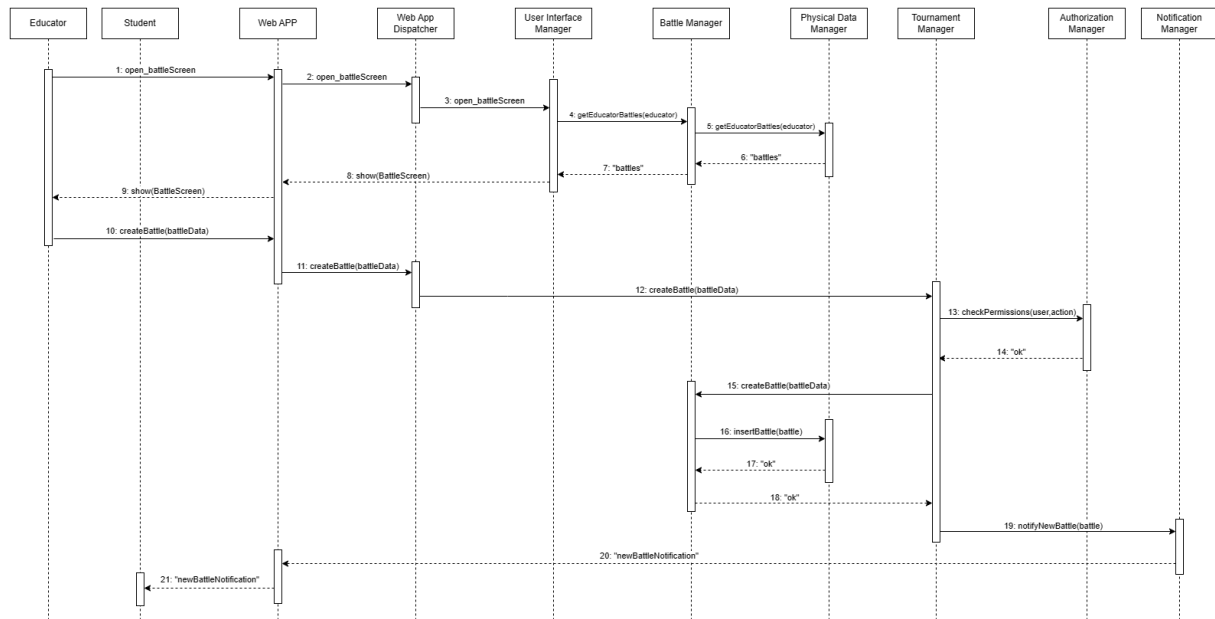


Figure 11: Battle creation runtime view

This sequence diagram shows all the actions performed when an educator creates a new battle. From the "battles" page, the educator submits all the battle's data, such as name, description, software components, automated evaluation settings, and so on. This information is then forwarded to the Tournament Manager. The Tournament Manager checks, via the Authorization Manager, if the educator has permission to create a new battle in the context of the specific tournament, and then triggers the creation of the battle via the Battle Manager. Finally, it sends a notification to all the students subscribed to the tournament.

• Tournament Subscription:

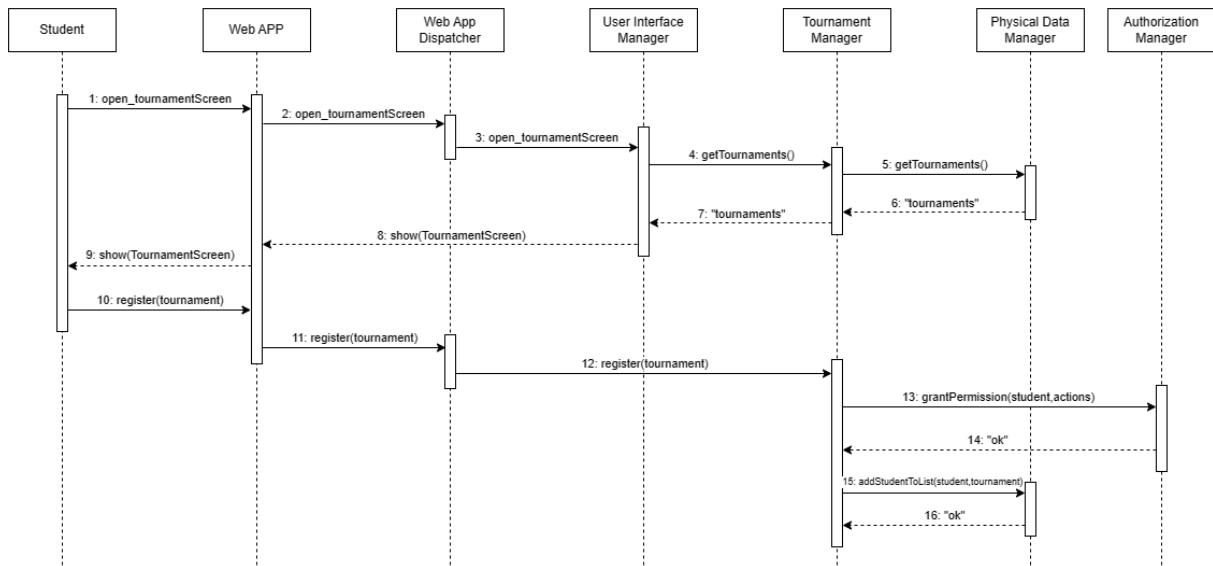


Figure 12: Tournament subscription runtime view

This sequence diagram shows all the actions performed when a student subscribes to a tournament. After the "tournaments" page has been displayed, the student select the tournament he wants to subscribe to. The request is forwarded from the Web App Dispatcher to the Tournament Manager, which is responsible for adding the permission to enlist for battle within the tournament to the student. The student is then added to the tournament student's list.

• Team Formation:

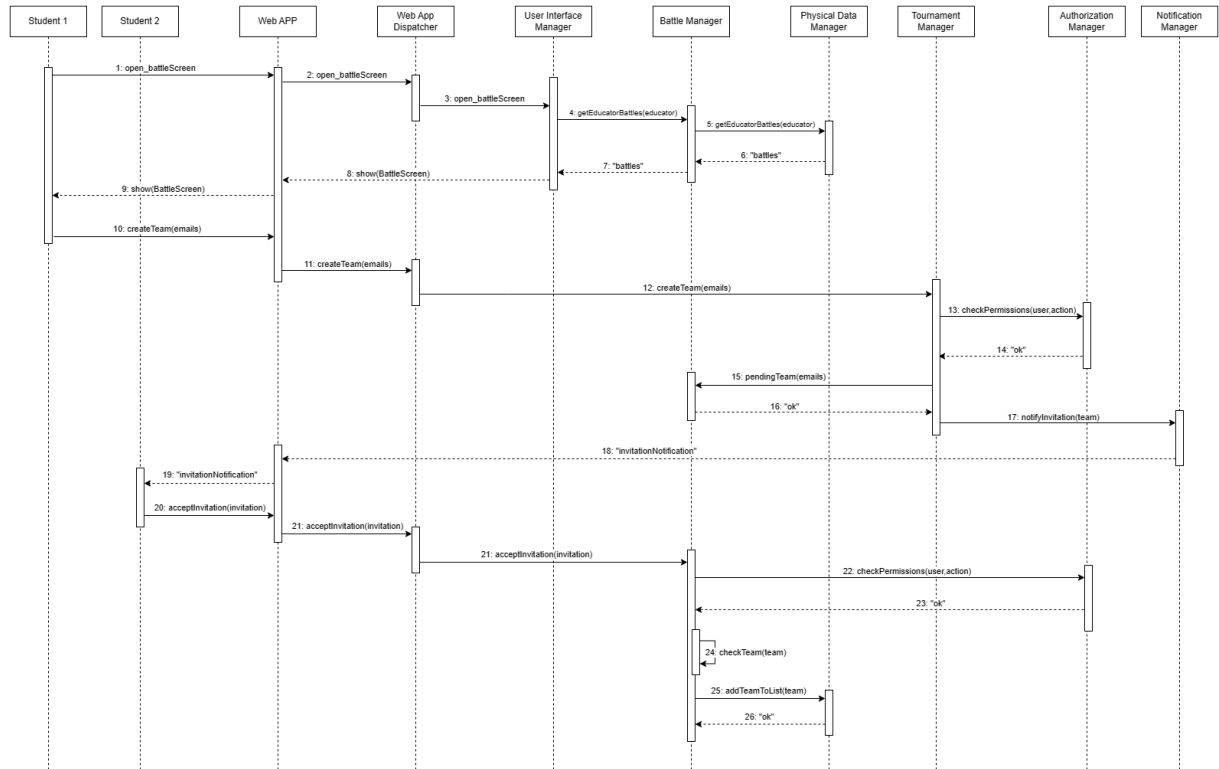


Figure 13: Team formation runtime view

This sequence diagram shows all the actions performed when a student wants to form a team to compete in a battle. The diagram includes both the team formation and the invitation handling. The student who wants to create the team selects the battle in the "battle" page, inserting the emails (or usernames) of their teammates. The request is sent to the Tournament Manager, which is responsible to check whether all the students in the team have permission to join the battle (for example, the student must be subscribed to the tournament). If the checks are positive, the team's information are forwarded to the Battle Manager and a notification is sent to the other teammates. When enough teammates accept the invitation, the Battle Manager inserts the team into the list of enlisted groups. If the group is formed by one student, and the battle allows single students to participate, no invitations are sent and the student is directly enlisted to the battle.

- **Continuous Integration:**

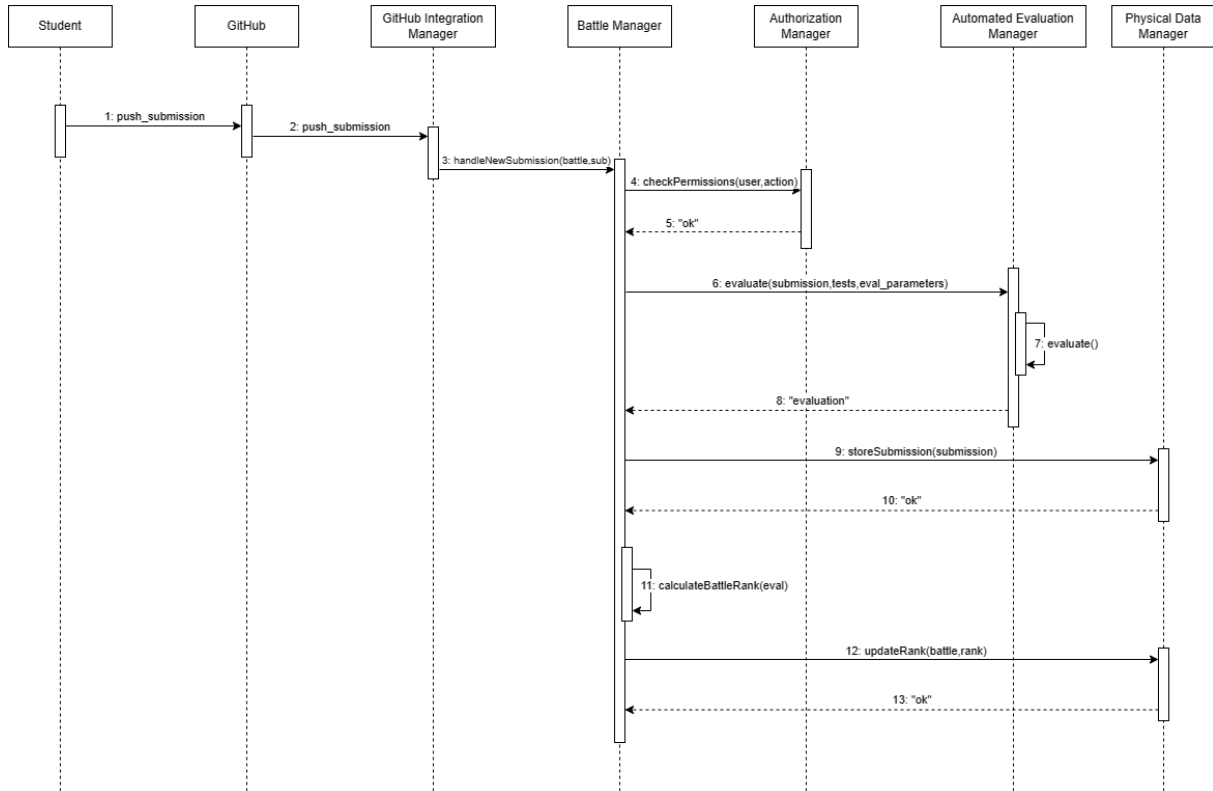


Figure 14: Continuous integration runtime view

This sequence diagram shows all the actions performed when a student submits a new solution for the Code Kata. Since no direct interaction happens between the student and the system, this runtime view does not correspond to any use case, but it is useful to display what happens during the battle. When the student performs a "push", via GitHub Action the submission is notified to the GitHub Integration Manager. The solution is then forwarded to the Battle Manager, which is responsible for checking if the issuer has permission to submit a new solution. Once the check has been performed, the student's code is sent to the Automated Evaluation Manager, which evaluates the solution based on the test cases and on the evaluation parameters provided by the educator at battle creation time. Finally, the solution is stored in the database and the battle rank is updated.

- Real-time battle rank visualization:

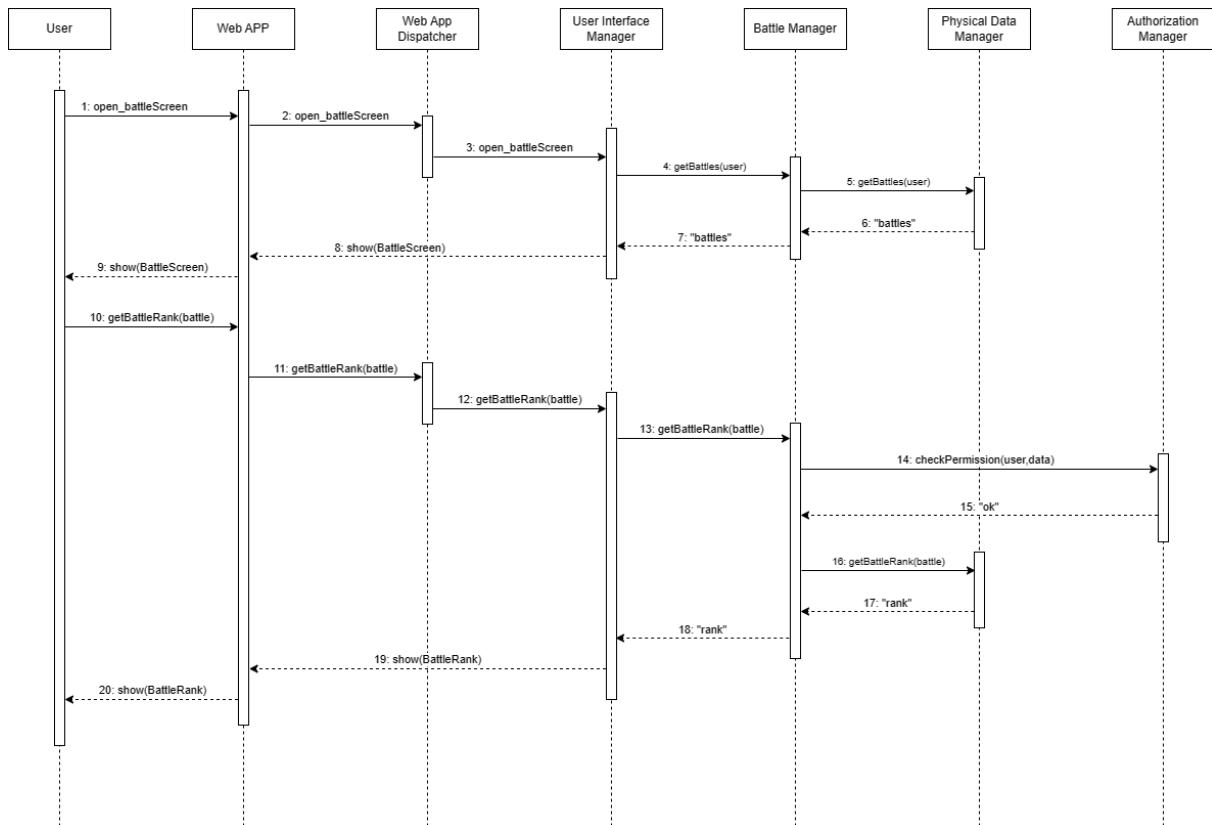


Figure 15: Real-time Battle Rank Visualization runtime view

This sequence diagram shows all the actions performed when a user (either a student or an educator), requests the current rank of a battle. Once the "battles" page has been displayed, the user selects the battle whose rank they are interested in. The request is forwarded to the User Interface Manager, that is responsible for collecting the data to be displayed. When the Battle Manager is consulted, after checking that the user has permission to access the data (in order to see the battle's rank, the user must be involved in a battle, either as the creator or as a participant), it queries the Physical Data Manager, returning the requested ranking to the User Interface Manager, that finally loads the rank page to show to the user.

• Code Manual Evaluation and Rank Calculation:

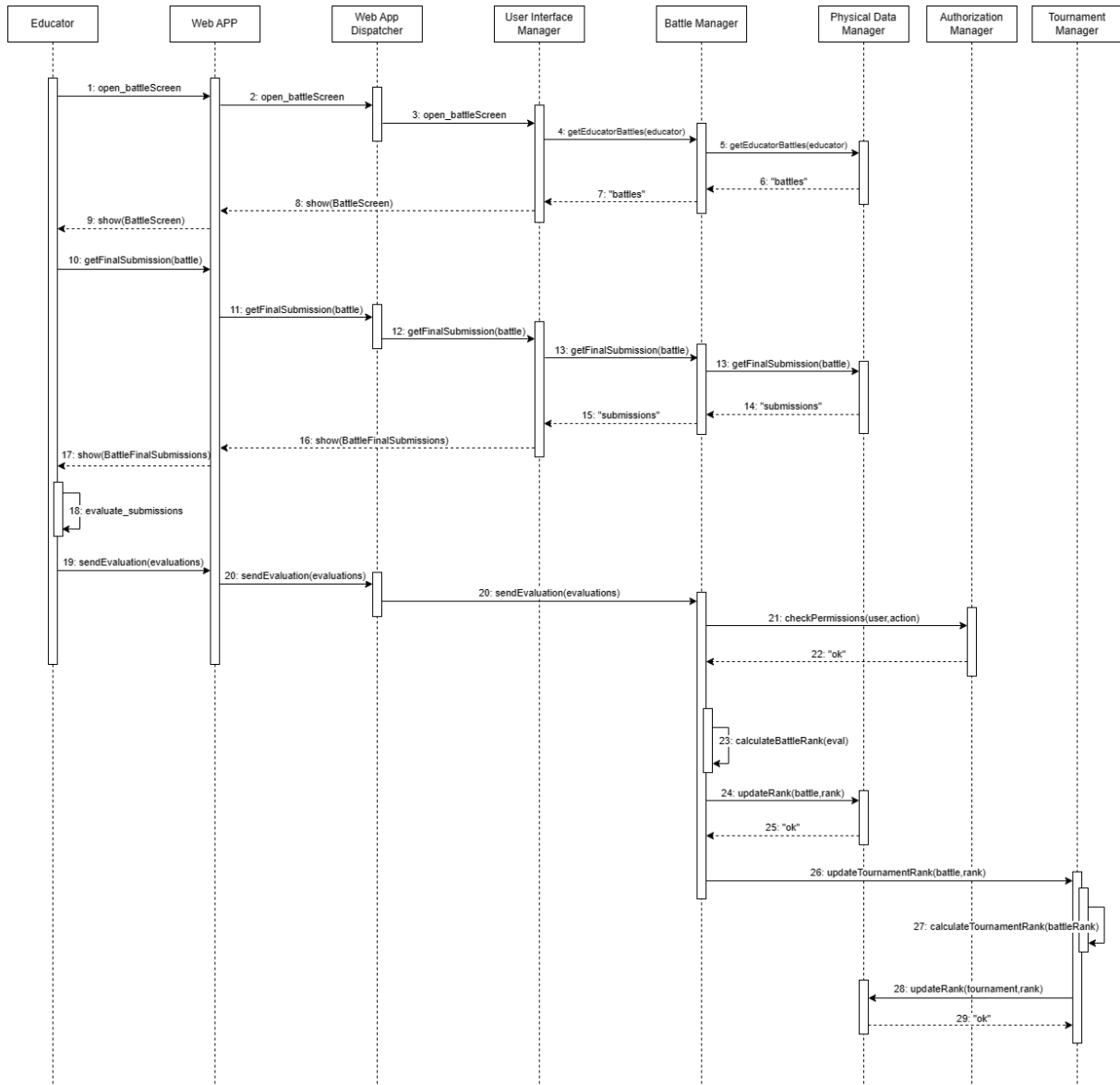


Figure 16: Code manual evaluation and Rank calculation runtime view

This sequence diagram shows all the actions performed when an educator evaluates the students' solutions after a battle has been closed. The educator collects all the teams' final submissions via the "battles" page. Once the submissions have been graded, the evaluations are sent to the Battle Manager. After an authorization check, the final battle ranking is calculated and stored in the database. The rank is then sent to the Tournament Manager, which is responsible for updating the rank of the tournament corresponding to the close battle.

The runtime view in the case where the manual evaluation is not required is a simplified version of this one: when the battle is closed, the Battle Manager collects the evaluations from the Automated Evaluation Manager, proceeding then from step 23 with the calculation of the battle rank.

• Tournament Rank Visualization:

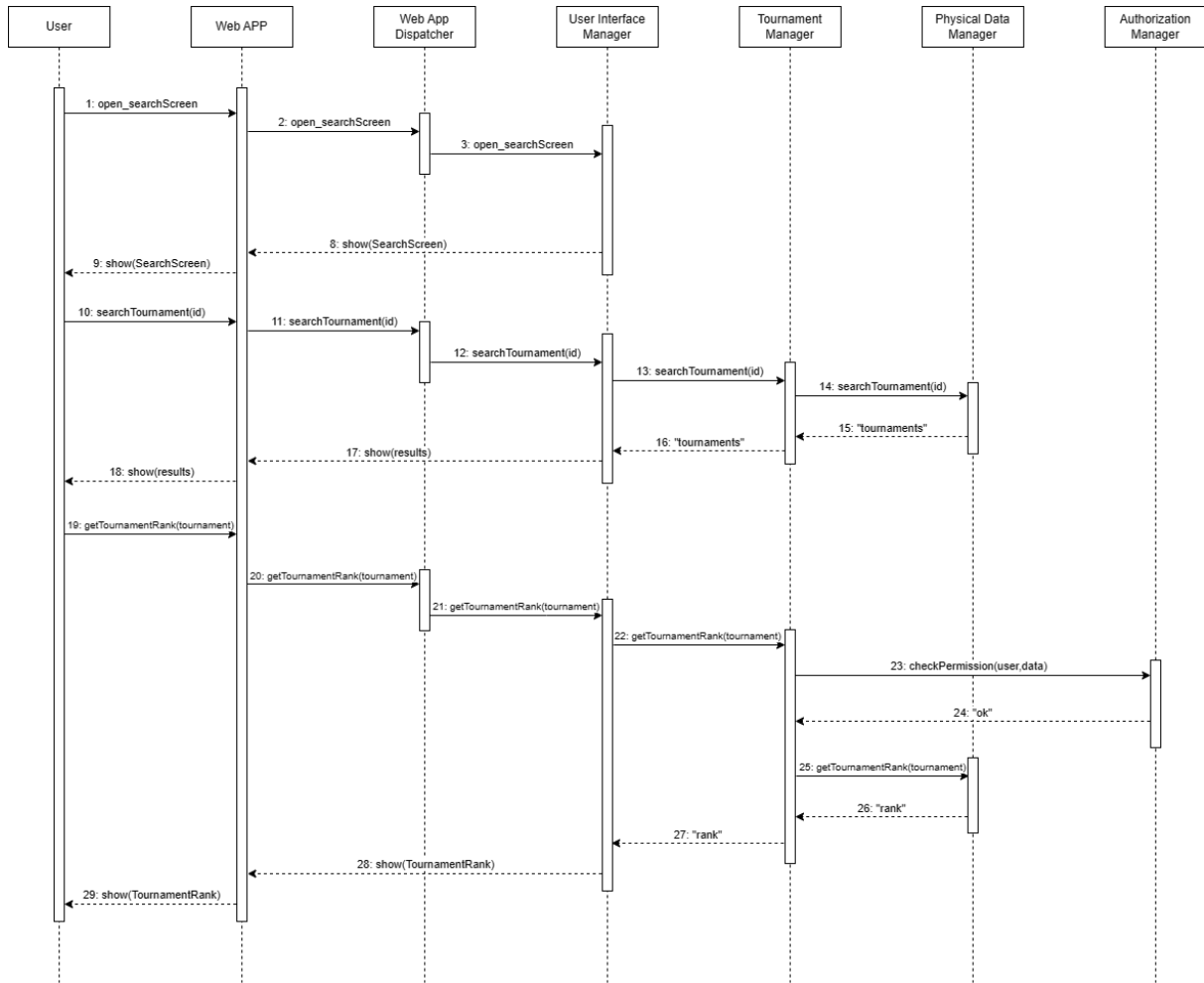


Figure 17: Tournament Rank Visualization runtime view

This sequence diagram shows all the actions performed when a user (either a student or an educator), requests the current rank of a tournament. In order to be able to see the rank of all ongoing or closed tournaments, the user first enters the "search" page. From there, the user inserts an identification of the requested tournament (i.e. the name or part of it). The request is sent to the Tournament Manager, which in turn provides the set of tournaments corresponding to the provided id. Once the list of tournaments has been displayed to the user, they select the tournament whose rank they are interested in. The request is again forwarded to the Tournament Manager, that after an authorization check returns to the User Interface Manager the rank. Finally, the User Interface Manager prepares the content to be provided to the user.

• Tournament Closure and Badge Obtaining:

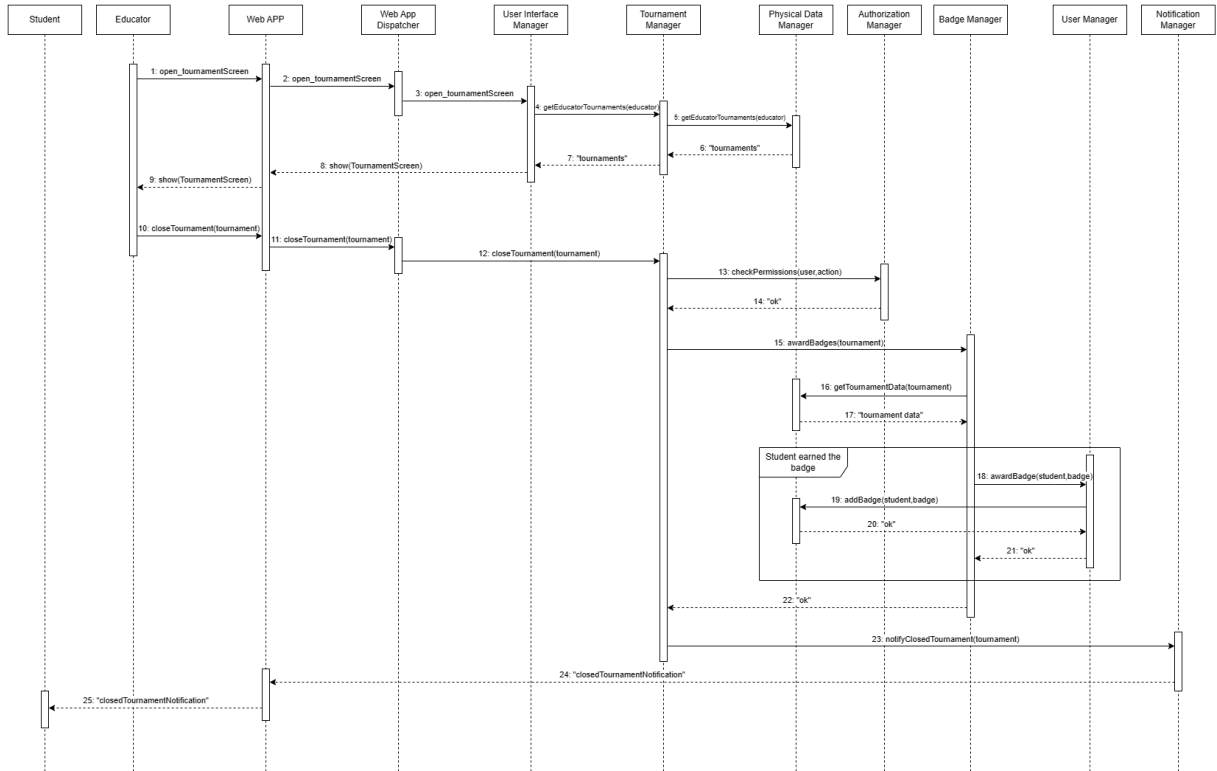


Figure 18: Tournament closure and Badge obtaining runtime view

This sequence diagram shows all the actions performed when an educator closes a tournament. After the educator selects the tournament to close from the "tournaments" page, the request is sent to the Tournament Manager that performs the required authorization checks. The Badge Manager is then responsible for collecting all the necessary data and checking, for each student subscribed to the tournament, if the constraints for winning each of the badges (defined during the tournament creation) are respected. If so, the badge is sent to the User Manager, which is responsible for adding the badge to the student's profile. At the end of the badge assignation process, the tournament is officially closed and a notification is sent to all the registered students.

• Student Profile Visualization:

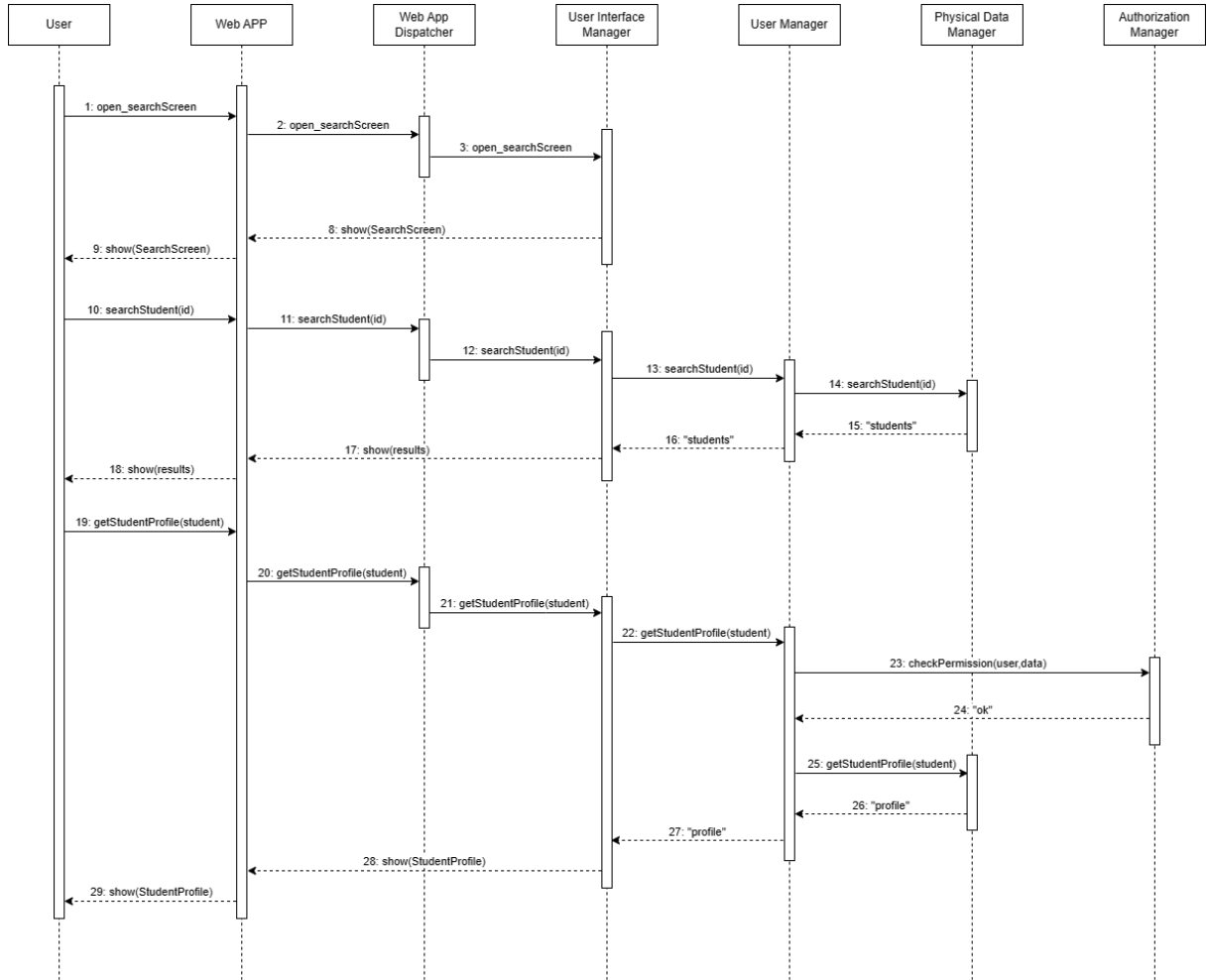


Figure 19: Student Profile Visualization runtime view

This sequence diagram shows all the actions performed when and user (either a student or an educator), requests the profile of a student, containing information such as the badges earned by the student. The user first enters the "search" page. From there, the user insert an identification of the requested student (i.e. the username or part of it). The request is sent to the User Manager, which in turn provides the set of students corresponding to the provided id. Once the list of students has been displayed to the user, they select the students whose profile they are interested in. The request is again forwarded to the User Manager, that after an authorization check returns to the User Interface Manager the student's data. Finally, the User Interface Manager prepares the content to be provided to the user.

2.5 Component Interfaces

This section contains the functionalities provided by each interface to the other components. The lists include both generic methods offered in order to perform a wide set of operations, and case-specific methods employed for a clear and efficient usage of the provided functionalities.

- **Web App Dispatcher:**

- `handleRequest(request)`

- **User Manager:**

- `checkStudentExistence(email, username)`
- `checkEducatorExistence(email, username)`
- `createStudent(email, username, password, student_data)`
- `createEducator(email, username, password, educator_data)`
- `getStudentData(email)`
- `getEducatorData(email)`
- `updateStudentData(email, student_data)`
- `updateEducatorData(email, educator_data)`
- `awardBadge(student_email, badge)`
- `deleteStudentData(email)`
- `deleteEducatorData(email)`

- **Registration Manager:**

- `handleStudentRegistrationCredentials(email, username, password)`
- `handleEducatorRegistrationCredentials(email, username, password)`
- `checkRegistrationCredentialsCorrectness(email, username, password)`

- **Login Manager:**

- `handleStudentLoginCredentials(username, password)`
- `handleEducatorLoginCredentials(username, password)`
- `checkLoginCredentialsCorrectness(username, password)`

- **Authorization Manager:**

- `checkPermission(user, action)`
- `grantPermission(user, action)`
- `revokePermission(user, action)`

- **Tournament Manager:**

- getTournaments(search_filters)
- getStudentTournaments(student)
- getEducatorTournaments(educator)
- createTournament(name, creator, badges, tournament_data)
- inviteEducator(educator, tournament)
- createBattle(battle_data)
- registerStudent(student, tournament)
- createTeam(students, battle)
- updateTournamentRank(battle, rank)
- closeTournament(tournament)
- getTournamentRank(tournament)
- getTournamentData(tournament)
- updateTournamentData(tournament)
- deleteTournamentData(tournament)

- **Battle Manager:**

- getBattles(search_filters)
- getStudentBattles(student)
- getEducatorBattles(educator)
- createBattle(name, creator, description, parameters, software_components, battle_data)
- addPendingTeam(students, battle)
- acceptInvitation(student, team, battle)
- checkTeam(team)
- handleNewSubmission(submission, battle)
- calculateBattleRank(evaluations, battle)
- getBattleLastSubmissions(battle)
- handleEducatorEvaluations(evaluations, battle)
- getBattleRank(battle)
- getBattleData(battle)
- updateBattleData(battle)
- deleteBattleData(battle)

- **GitHub Integration Manager:**

- createRepository(battle)
- setupWorkflow(repository, students)
- closeRepository(repository)

- **Automated Evaluation Manager:**

- evaluateSubmission(code, software_components, eval_parameters)

- **User Interface Manager:**

- preparePageContent(page, content)
- loginPage()
- registerAsStudentPage()
- registerAsEducatorPage()
- loginAsStudentPage()
- loginAsEducatorPage()
- studentHomePage(student_data)
- educatorHomePage(educator_data)
- studentTournamentsPage(student, tournaments)
- educatorTournamentsPage(educator, tournaments)
- studentBattlesPage(student, battles)
- educatorBattlesPage(educator, battles)
- inviteEducatorPage(educator, tournament)
- tournamentBadgesPage(educator, tournament)
- battleFinalSubmissionsPage(educator, battle_final_submissions)
- battleRankPage(battle_info)
- searchPage()
- tournamentRankPage(tournament_info)
- studentProfilePage(student_info)

- **Badge Manager:**

- awardBadges(tournament)
- checkBadge(student, badge, tournament)

- **Notification Manager:**

- sendNotification(email, content)
- sendConfirmationLinkEmail(email)
- sendConfirmationEmail(email)
- notifyNewTournament(tournament)
- notifyInvitation(team, emails)
- notifyNewBattle(battle)
- notifyBattleClosure(battle)
- notifyTournamentClosure(battle)

- **Physical Data Manager:**

- executeQuery(query)
- insert(data)
- select(query)
- update(data)
- delete(data)
- createStudent(email, username, password, student_data)
- getStudentData(student)
- updateStudentData(student, student_data)
- addBadge(student, badge)
- deleteStudentData(student)
- createEducator(email, username, password, educator_data)
- getEducatorData(educator)
- updateEducatorData(educator, educator_data)
- deleteEducatorData(educator)
- createTournament(name, creator, badges, tournament_data)
- getTournaments(search_filters)
- getStudentTournaments(student)
- getEducatorTournaments(educator)
- getTournamentRank(tournament)
- updateTournamentRank(battle, rank)
- getTournamentData(tournament)
- updateTournamentData(tournament)
- deleteTournamentData(tournament)

- createBattle(name, creator, description, parameters, software_components, battle_data)
- getBattles(search_filters)
- getStudentBattles(student)
- getEducatorBattles(educator)
- getFinalSubmissions(battle)
- storeSubmission(submission, battle)
- getBattleRank(battle)
- updateBattleRank(battle, rank)
- getBattleData(battle)
- updateBattleData(battle)
- deleteBattleData(battle)

2.6 Selected Architectural Styles and Patterns

This chapter talks about the architectural design choices that define the CodeKataBattle platform, highlighting patterns that contribute to its robustness and flexibility.

- **3-Tier Architecture:**

The CKB platform is based on a 3-tier architecture, that organize its system's components logically and physically into three distinct tiers: presentation, application, and data tier. Thanks to the modularization, this architecture allows for a clear division of responsibilities, promotes modularity and scalability, facilitating efficient development and maintenance, and guarantees a higher flexibility by allowing to develop and update a specific part of the system at time.

- **Chain of Responsibility Pattern:**

It's employed to establish a chain of handler objects. Each handler has the capability to process a request or pass it along the chain. This pattern enhances flexibility by allowing multiple objects to handle a request without the sender needing to specify the exact handler. It's particularly useful for processing various user requests and system events, enabling a modular and scalable approach to request handling. In CKB system the first chain link is the dispatcher, which is in charge of passing the request to one or more components. Each component then handles the request, forwarding it to the other components if necessary.

- **Microservices Architecture:**

The CKB platform's architecture is based on a microservices architectural style. Microservices break down the system into independently deployable and scalable services, enhancing agility and adaptability. Each component of the system embodies a small set of functionalities related to single entities (tournaments, battles, ...),

and the common operations performed by each component, such as data retrieval or the preparation of the user interface to display, are allocated to specialized components.

- **Strategy Pattern for Badges:**

For badge management, the CKB platform incorporates the strategy design pattern. It defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. In the context of badges, it allows educators to define scoring algorithms dynamically, that can be changed at run-time, influencing how badges are awarded. The Badge Manager component is then responsible for employing the defined checks in order to award badges to student, based on the large set of rules offered to the educators.

- **Observer Pattern for Subscriptions:**

For subscription management, the CKB platform uses the observer pattern. This pattern establishes a one-to-many dependency between objects, ensuring that when one object changes state, all its dependents are notified and updated automatically. Applied to subscriptions, this enables real-time notifications to students about upcoming battles and changes in tournament status.

- **Command-Query Separation (CQS):**

This pattern separates the methods that modify data (commands) from the methods that return data (queries) to improve clarity and maintainability. Commands and queries are distinct, preventing unintended side effects when retrieving information. It is employed in the Physical Data Manager: the component offers, in addition to interfaces for generic queries, specific types of requests and commands for each case.

- **State Pattern:**

It is applied to enable an object to change its behavior when its internal state changes. This pattern is particularly useful when an object has different behaviors in response to various states. In CKB, it's employed to manage the lifecycle of battles and tournaments, allowing them to transition between states and modify the response to certain requests based on them.

- **Façade Pattern:**

The façade pattern is applied to simplify the interactions between complex subsystems offering a unified interface. Within the CKB system, it's used in the implementation of the dispatcher component. It encapsulates the underlying complexities of the system to hide the complexity of the application server to the client application, that has a user-friendly interface, and reducing dependencies. In fact, this also provides a simple interface for users.

- **Mediator Pattern:**

The mediator pattern manages the communication between all the different components without direct dependencies, to facilitate communication and coordination

between them. In the CKB platform, it reduces coupling, promotes flexibility and easiness of modification, allowing changes in one component to have minimal impact on others. It is interposed between two or more objects and encapsulates their communication, so that they cannot know each other's implementation details. In addition, it not only enhances flexibility but also simplifies the addition of new features and functionalities.

2.7 Other Design Decisions

- **Load Balancer and Server Replicas:**

To strengthen system availability, the CKB platform integrates three load balancers that distribute incoming traffic across multiple server replicas. This setup not only optimizes resource utilization but also mitigates the risk of downtime due to server failures. The load balancer intelligently redirects requests, ensuring a balanced and efficient distribution of workloads among server instances. In case of a server failure, the load balancer redirects traffic to healthy replicas, minimizing interruptions and providing users with continuous access to the platform.

- **Distributed Database on a Cluster:**

At the database level, the CKB platform adopts a unique logical database distributed on a cluster of physical databases. This architecture involves the distribution of database instances across multiple nodes, forming a resilient cluster. In the event of a database node failure, the distributed nature of the cluster ensures that other nodes take over, preventing service interruptions. This approach not only improves fault tolerance but also supports scalability and performance optimization.

The combination of load balancing for system components and a distributed database cluster strengthens the CKB platform's availability. By distributing workloads and data across multiple nodes, the platform remains robust to potential malfunctioning, offering users uninterrupted access and reliable performance.

3 User Interface Design

The design of an intuitive yet expressive User Interface is important during the creation of the described application. The main activity of CKB is the development of source code both by students and educators, activity that is mainly carried out on computers rather than smartphones. Because of this, the target architecture for the final application is the computer. As indicated in the Requirement Analysis and Specification Document, there are two types of users (students and educators), requiring two different sets of functionalities. To keep these two sets separated, two User Interfaces are illustrated: a Student UI and an Educator UI. As reported in the RASD, these representations are not meant to show the final product in details, but to provide a possible user interface along with the promise of the represented functionalities. In particular:

- The Student UI must allow students to log in the system, visualize the badges won in all the tournaments they participated to, join new tournaments and see the ranking of all the opened and terminated tournaments. The UI must also allow students to join battles and invite other students to form a team. A Search option has been added to search for Tournaments and Students, allowing the visualizations of other students' badges.
- The Educator UI must allow educators to see all the tournaments created by them, displaying the actions performable for each one. It must display also the set of battles created in the context of each tournament. Educators must be able to create battles and tournaments and add personalized badges in new tournaments. The interface for defining badges will be analyzed later.

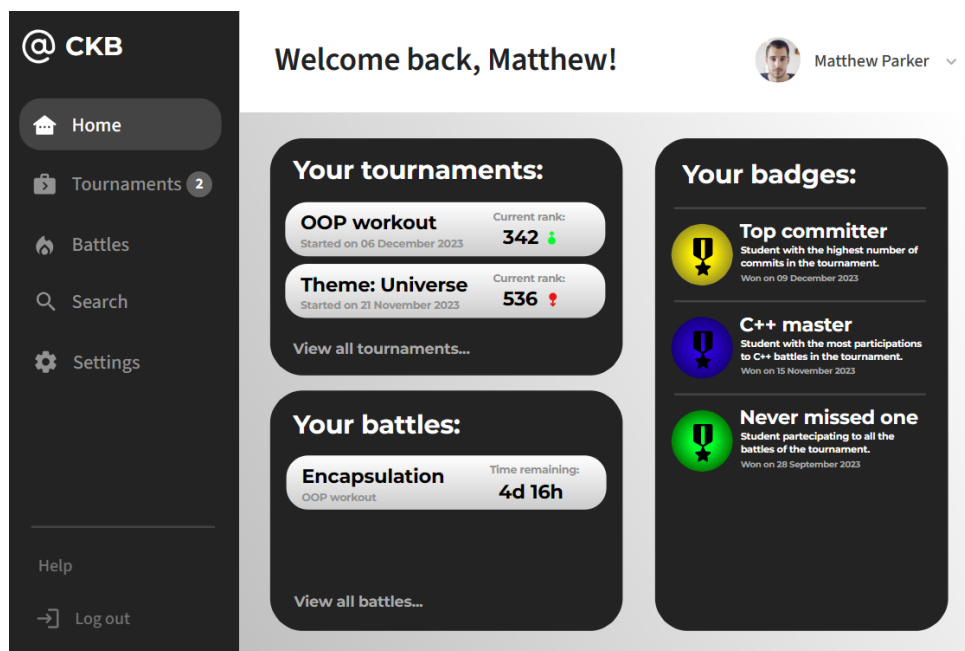


Figure 20: Student Home screen

@ CKB

Home

Tournaments

Battles

Search

Settings

Help

Log out

Tournaments

Matthew Parker

Your tournaments:

Order by: Date

Name:	Started on:	Your rank:
OOP workout	06 December 2023	342
Theme: Universe	21 November 2023	536

New tournaments:

Order by: Newest

Search

Name:	Time for registration:	Actions:
Time to optimize! NEW	4d 7h	Register
Data structures NEW	4d 2h	Register
Fast typers	2d 16h	Register
Web applications	16h 43m	Register
Cybersecurity	2h 5m	Register

Figure 21: Student Tournaments screen

@ CKB

Home

Tournaments

Battles

Search

Settings

Help

Log out

Battles

Matthew Parker

Your battles:

Name:	Tournament:	Time remaining:	Live rank:
Encapsulation	OOP workout	4d 16h Submit	16 View rank

New battles

Name:	Tournament:	Starting in:	Members:	Actions:
To the moon!	Theme: Universe	4d 7h	1	Create team
Interfaces	OOP workout	4d 2h	2 - 4	Create team

Closed battles:

Name:	Tournament:	Status:
Galactic complexity	Theme: Universe	Consolidating...

Figure 22: Student Battles screen

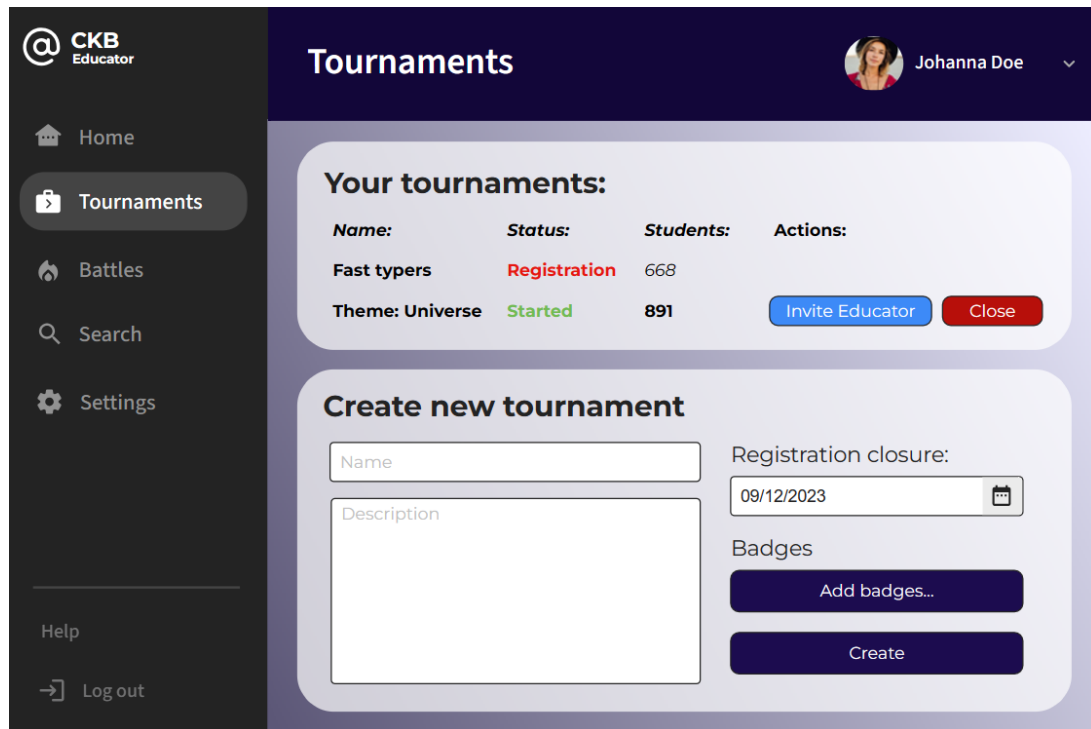


Figure 23: Educator Tournaments screen

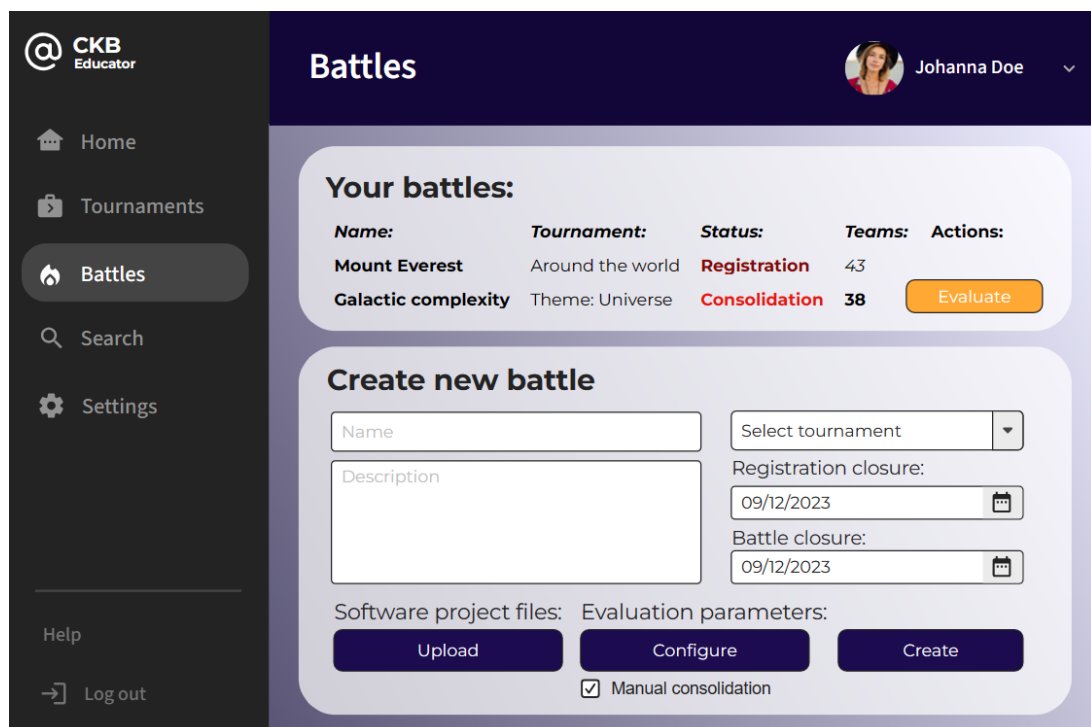


Figure 24: Educator Battles screen

To conclude the description of the User Interface, we present the interface offered to educators to define badges when creating a tournament, along with the specification of the rules that must be met by students in order to earn each badge. The criteria on which the deserving students can be selected depend on the subjective evaluation of the educator, and defining a closed set of variables and rules shared by all the educators limits the possible usage of a powerful mechanic such as gamification. On the other hand, defining a generic, powerful tool to define a wide range of filters, runs the risk of result in an interface too complicate for its common use, rendering the definition of simple, repetitive rules a tedious task. CodeKataBattle mitigates this issue by offering two ways of defining rules: one simple, intuitive way and one detailed and expressive way.

The first modality of rule definition consists in a simple comparison between two variables: one referred to the individual performance of each student (student specific) and one referred either to the overall performance of all the student (global) or equal to a fixed constant. The terms of comparison between the two quantities are the ones applicable to two numeric parameters: $<$, \leq , $=$, \neq , \geq , $>$. The badge is then assigned to all the student whose specific variables satisfy the constraint (or the set of constraints). For example, assigning a badge to the student with the highest number of commits can be achieved in the following way:

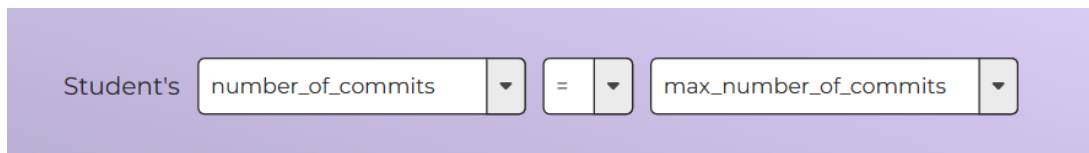


Figure 25: Student with maximum number of commits

As another example, assigning a badge to a student participating in all the battles in the tournament and reaching the first ten positions in the final tournament ranking can be performed in the following way:

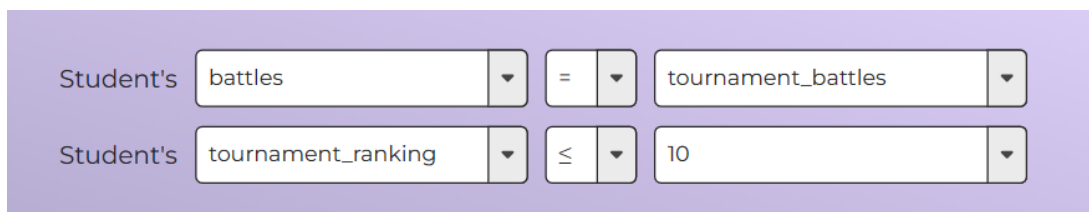


Figure 26: Student participating in all the battles and reaching the first ten positions in the tournament

The set of student variables, which will have a dedicated section explaining their meaning in the final application, is the following:

```
battles, //the number of battles in which the student participated
battles_won, //the number of battles in which the student
```

```
        //reached the first position in the final rank
max_battle_ranking, //the maximum rank reached at the end of a battle
max_commits, //maximum number of commits issued in a battle
max_tournament_ranking, //the maximum rank reached during the tournament
number_of_commits, //total number of commits issued by the student
number_of_teammates, //total number of distinct teammates that
        //participated with the student to the battles
tournament_ranking, //the final rank of the student in the tournament
```

The set of global variables, which will have a dedicated section explaining their meaning in the final application, is the following:

```
tournament_battles, //the number of battles in the tournament
max_battles_won, //the maximum number of battles won
        //by any student
max_number_of_commits, //maximum number of commits issued by any student
max_number_of_max_commits, //maximum number of commits issued
        //by any student in a battle
```

This set of predefined variables is not meant to be strictly closed and can be enriched during the development and maintenance of the system, along with a clear explanation of their meaning in a dedicated section.

This form of rules is very intuitive and easy to use, but allows only for a restricted set of possible criteria for deciding whether to assign a badge or not. As a first step towards a more expressive interface, the system offers a way of defining new variables using JavaScript, specifying a function that returns a number starting from a wide dataset, which allows more complex information to be taken into account. The information offered to the educator is the following:

tournament

```
.startTime //timestamp of the beginning of the tournament
//timestamps are expressed as second elapsed since 00:00:00 of 1/1/1970 UTC
.finishTime //timestamp of the beginning of the tournament
.students[] //array of students
    .email //email of the student
    .finalRank //final rank of the student
    .ranks[] //rank of the student at the end of each battle
        //regardless if the student participated or not
.battles[] //array of battles
    .startTime //timestamp of the beginning of the battle
    .finishTime //timestamp of the ending of the battle
    .languages[] //programming languages of the battle
    .registeredStudents[] //array of student emails
    .teams[] //array of teams
```

```

.members[] //emails of members of the team
.rank //final position in the rank of the battle
.commits[] //array of commits:
           //the last entry is the final submission
.timestamp //timestamp of the submission
.language //programming language used
.author //student email
.text[] //lines of code

```

All the educator has to do is define a function that uses the previous data to return a quantity: if the function describes a student specific variable, then the "student" object will be passed as a parameter. As an example, here is the process to describe the number of participations of each student in battles solvable with the programming language C++:

```

function cpp_partecipations(student){
  let count = 0;
  for(let b of tournament.battles){
    if(b.registeredStudents.includes(student.email) &&
      b.languages.includes("C++")){
      count++;
    }
  }
  return count;
}

```

After this, we can define a new global variable as the maximum number of participations in a C++ battle by any student:

```

function max_cpp_partecipations(){
  let max = 0;
  for(let s of tournament.students){
    let curr = cpp_partecipations(s);
    if(max < curr)
      max = curr;
  }
  return max;
}

```

Once the two variables have been defined, the rule indicating the student with the most participations in C++ battles can be expressed as above:



Student's =

Figure 27: Student with the most participations to C++ battles

We note that the scripts proposed by the educators will be executed in the same controlled environment as the student submissions, and badges with incorrect descriptions will be discarded.

To add yet another layer of expressiveness, the system allows educators to define a rule function that directly returns if a student is eligible for a certain badge or not. To make a final example, here is the rule for selecting the students who issued two commits in less than a minute:

```
function two_commits_one_minute(student){
  for(let b of tournament.battles){
    for(let t of b.teams){
      if(t.members.includes(student.email)){
        for(let i=0; i<t.commits.length-1; i++){
          for(let j=i+1; j<t.commits.length &&
            t.commits[j].timestamp - t.commits[i].timestamp < 60; j++){
            if(t.commits[i].author == student.email &&
              t.commits[j].author == student.email){
              return true;
            }
          }
        }
      }
    }
  }
  return false;
}
```

These are all the options that an educator has to describe the rules for assigning badges to students. The system offers both a simple, intuitive way to assign badges to students based on simple parameters and a more complex, expressive way of defining detailed rules. The scripts used for defining rules can be saved to be reused and shared among educators and many of them are offered as templates to make the draft of complex rules more intuitive, along with removing the strict need for educators to eventually learn JavaScript in depth just for this purpose.

4 Requirements Traceability

In this section, the design elements employed for the fulfillment of the requirements presented in the Requirement Analysis and Specification Document are presented.

Requirements Components	R1: The system must allow users to register on the platform
	<ul style="list-style-type: none"> • WebAPP • Web Server: <ul style="list-style-type: none"> – User Interface Manager – WebAPP Dispatcher • Application Server: <ul style="list-style-type: none"> – Registration Manager – User Manager – Notification Manager • Database Server: <ul style="list-style-type: none"> – Physical Data Manager • DBMS • E-mail Provider

Table 1: Requirement traceability of R1

Requirements	R2: The system must allow users to authenticate themselves and login securely
Components	<ul style="list-style-type: none"> • WebAPP • Web Server: <ul style="list-style-type: none"> – User Interface Manager – WebAPP Dispatcher • Application Server: <ul style="list-style-type: none"> – Login Manager – User Manager • Database Server: <ul style="list-style-type: none"> – Physical Data Manager • DBMS

Table 2: Requirement traceability of R2

Requirements	<p>R3: The system must allow educators to create new tournaments</p> <p>R4: Tournament creation must include specifying a description of the tournaments</p> <p>R16: The system automatically notifies students and educators about new tournaments, battles, and tournament closures</p> <p>R17: Notifications must include deadlines and updates</p> <p>R18: The system must allow the educator owning the tournament to authorize other educators to create CKB</p>
Components	<ul style="list-style-type: none"> • WebAPP • Web Server: <ul style="list-style-type: none"> – User Interface Manager – WebAPP Dispatcher • Application Server: <ul style="list-style-type: none"> – Tournament Manager – Authorization Manager – Notification Manager • Database Server: <ul style="list-style-type: none"> – Physical Data Manager • DBMS

Table 3: Requirement traceability of R3, R4, R16, R17 and R18

Requirements	<p>R5: The system must allow authorized educators to create code kata battles within a tournament</p> <p>R6: Creation of CKB must include kata description, software project, build scripts, and scoring configurations</p> <p>R7: The system must allow students to register for individual battles or form teams based on specified team size limits</p> <p>R11: The system must allow educators to manually evaluate submissions</p> <p>R12: The system must allow authorized educators to add battles to tournaments</p>
Components	<ul style="list-style-type: none"> • WebAPP • Web Server: <ul style="list-style-type: none"> – User Interface Manager – WebAPP Dispatcher • Application Server: <ul style="list-style-type: none"> – Tournament Manager – Authorization Manager – Battle Manager – Notification Manager • Database Server: <ul style="list-style-type: none"> – Physical Data Manager • DBMS

Table 4: Requirement traceability of R5, R6, R7, R11, and R12

Requirements	<p>R8: The system automatically creates GitHub repositories for battles</p> <p>R13: The battle closure triggers updates to personal tournament scores for each student</p>
Components	<ul style="list-style-type: none"> • WebAPP • Web Server: <ul style="list-style-type: none"> – User Interface Manager – WebAPP Dispatcher • Application Server: <ul style="list-style-type: none"> – Tournament Manager – Battle Manager – Automated Evaluation Manager – GitHub Integration Manager – Notification Manager • Database Server: <ul style="list-style-type: none"> – Physical Data Manager • DBMS • GitHub

Table 5: Requirement traceability of R8 and R13

Requirements	<p>R9: Students must be able to fork the repository and set up automated workflows using GitHub Actions</p> <p>R10: During the battle, the platform updates scores with every push made by students in real time</p>
Components	<ul style="list-style-type: none"> • WebAPP • Web Server: <ul style="list-style-type: none"> – User Interface Manager – WebAPP Dispatcher • Application Server: <ul style="list-style-type: none"> – Tournament Manager – Authorization Manager – Battle Manager – GitHub Integration Manager – Notification Manager – Automated Evaluation Manager • Database Server: <ul style="list-style-type: none"> – Physical Data Manager • DBMS • GitHub

Table 6: Requirement traceability of R9 and R10

Requirements	<p>R14: The system must allow educators to create badges associated with specific tournaments</p> <p>R15: Badges are automatically assigned based on predefined rules and student performance</p>
Components	<ul style="list-style-type: none"> • WebAPP • Web Server: <ul style="list-style-type: none"> – User Interface Manager – WebAPP Dispatcher • Application Server: <ul style="list-style-type: none"> – Badge Manager – Tournament Manager – Authorization Manager – Notification Manager – User Manager • Database Server: <ul style="list-style-type: none"> – Physical Data Manager • DBMS

Table 7: Requirement traceability of R14 and R15

5 Implementation, Integration and Test Plan

The concluding chapter explores the system implementation details, clarifying the integration of its components and delineating the methodologies employed for validation and verification.

In light of this, the implemented testing protocols' primary objective is to fix most application bugs before each release, recognizing the intrinsic challenge of eliminating all defects. Therefore, while the initial paragraph focuses on the complexities of the implementation strategy, expected consideration is given to the integration test plan during its formulation.

Emphasizing the significance of comprehensive documentation, the code is expected to be well-commented and meticulously documented, utilizing tools such as Javadoc to extend the documentation process, ensuring clarity and facilitating a deeper understanding of the codebase. This approach to implementation, integration, and documentation aims to enhance the robustness and maintainability of the CodeKataBattle platform.

5.1 Implementation plan

The implementation approach for the CodeKataBattle platform blends the advantages of bottom-up and thread strategies, ensuring a comprehensive and effective development process. Adopting a threading strategy permits the generation of intermediate deliverables, providing stakeholders with measurable milestones crucial for validating the evolving system. Concurrently, employing a bottom-up strategy promotes incremental integration, boosting efficient bug tracking through the iterative testing of intermediate versions and the subsystems resulting from the integration of components.

The thread strategy entails a meticulous identification of system features and the corresponding portions of components (referred to as sub-components for practicality) responsible for delivering these features. A single function often involves collaboration among different sub-components, and a strategic order for their implementation is essential. The bottom-up approach is incorporated to address this challenge.

This implementation strategy facilitates the parallel assignment of feature implementation tasks to independent development teams, enabling concurrent progress. However, before this allocation, it is imperative to identify any common components to prevent redundant efforts in producing the same component or subcomponent. This hybrid approach, amalgamating bottom-up and thread strategies, is balanced to simplify the development process, improve collaboration among development teams, and optimize the overall implementation of the CodeKataBattle platform.

These processes provide incremental development, stakeholder engagement, bug tracking and testing, parallel development, adaptability to complexity, and avoidance of redundancy fitting perfectly the CodeKataBattle platform's needs.

Before proceeding with the component integration and testing paragraph, it can be useful to identify the features of the system starting from the requirements themselves.

- **[F1] User registration and authentication:**

User registration and authentication are entangled functionalities, each playing a role in ensuring secure access to the CodeKataBattle platform. Considering these features together is crucial, as the authentication process necessitates prior registration. During this phase, it is vital to establish a clear distinction between student and educator profiles. The system must store these distinct account types allowing for future distinguishability. This delineation ensures that, depending on whether a user is logged in as an educator or a student, the system provides dedicated commands and privileges tailored to the respective roles. This careful segregation improves the user experience, aligning the platform with the specific needs and permissions associated with each account type.

- **[F2] Tournament creation:**

Tournament creation is a feature exclusive to educators. Only educators can initiate tournaments, utilizing purpose-built interfaces to articulate essential details such as the description and rules. This separation of authority ensures that the responsibility of shaping the tournament landscape aligns with the instructional role of educators, facilitating a streamlined and organized creation process.

- **[F3] Battle creation:**

Educators can craft code kata battles within tournaments: the process consists of uploading code katas, defining group size constraints, establishing registration and submission deadlines, and configuring scoring parameters. It's important that within a given tournament, multiple educators may contribute by creating battles. However, an essential check is embedded in the system to ensure that educators seeking to create battles within the same tournament have obtained prior authorization from the tournament's creator. This measure maintains a structured and collaborative environment, aligning with the platform's governance model.

- **[F4] Allow other educators to create battles:**

Involving additional educators in the creation of battles within a tournament implicates a systematic invitation process facilitated through a dedicated interface. This step is exclusively within the privileges of the tournament creator.

- **[F5] Student registration to tournament and battle, Team formation:**

Upon logging into the platform, students can register for tournaments through a dedicated function. Once enrolled in one or more tournaments, students can further register and actively participate in the battles associated with those tournaments. A key aspect of the software development process lies in the concurrent consideration of team formation and battle registration. Team formation develops immediately after the registration process, inviting additional students through a specialized form. The team registration must be completed before the registration deadline expires. Only when the team reaches a valid number of members, it transitions to active participation in the battle. This set of sequential events and requisite conditions

highlights the intrinsic connection between these features, requiring a comprehensive and cohesive testing approach that evaluates their combined functionality.

- **[F6] GitHub Integration:**

The integration with GitHub is necessary for the correct functioning of the platform. This integration plays a dual role: the system relies on it for the accurate creation of a battle, while students leverage it to configure GitHub Actions, automating the code submission process to the platform. This collaborative interaction ensures the synchronization of platform operations and student workflows, reinforcing the platform's robust and efficient code submission mechanism.

- **[F7] Manual evaluation:**

This feature authorizes educators with the ability to manually evaluate student code, in addition to automated evaluation, before final rankings are determined.

- **[F8] Tournament closure:**

Educators can decide to manually close the tournament. The tournament closure requires several integrity checks that must be performed by many components.

- **[F9] Badge creation and assignment:**

After educators have created a tournament, they can augment gamification by adding badges through dedicated interfaces. Within this process, educators utilize a specific language to articulate the constraints that define the conditions under which a student can earn these badges. When the tournament is closed, the badges are automatically assigned to deserving students.

- **[F10] Notification system:**

Whenever a tournament or a battle undergoes a state change, users directly involved receive notifications containing all pertinent information to guide them through the game. The notification system is also integrated with the login process, educator invitation process, and team formation process.

5.2 Component integration and testing

The integration of components takes place following the implementation of features. Once all the components contributing to a specific feature are individually implemented, they are tested in conjunction with the functionalities they collectively represent. This systematic integration approach ensures that features are not only implemented on an individual basis but are also rigorously assessed for collaborative functionality.

- **[F1] User registration and authentication:**

The highest priority lies in the implementation and testing of registration and authentication functionalities since nearly every other system function necessitates user authorization and logging (as described in 28).

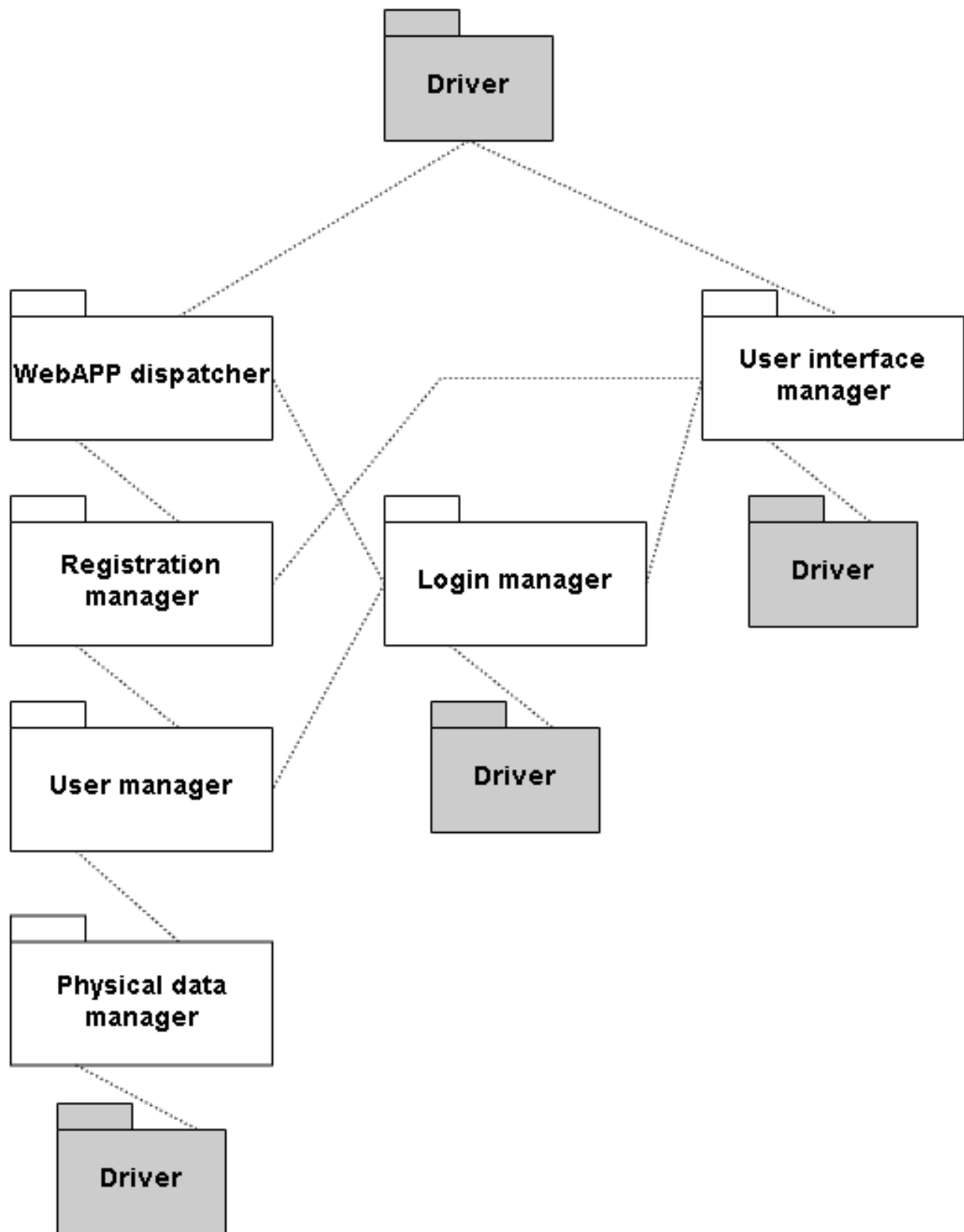


Figure 28: F1 Developing

- **[F2] Tournament creation:**

Next in line are the tournament creation functionalities, crucial to developing battles and supporting all the intricate functionalities within the game (as described in 29).

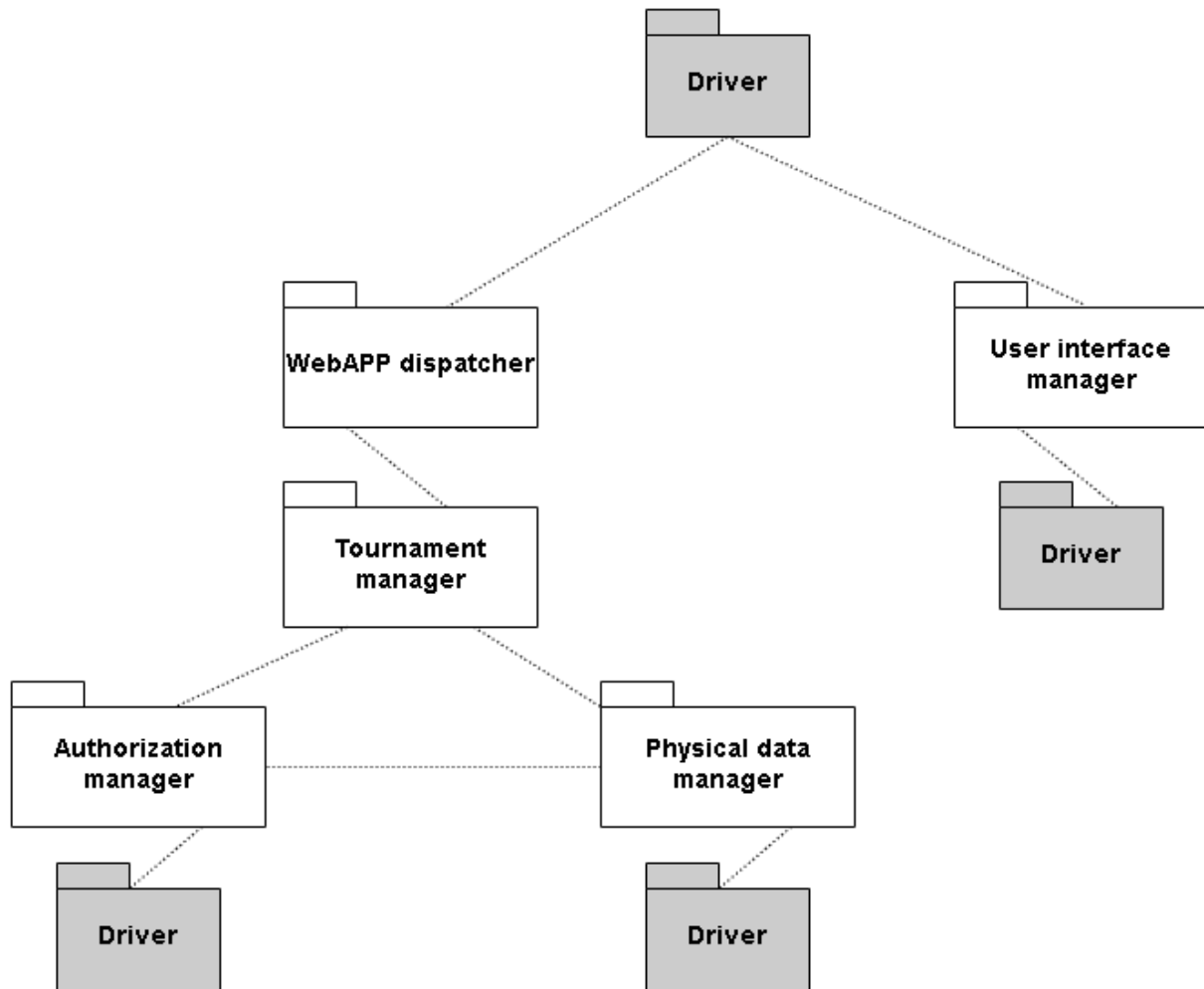


Figure 29: F2 Developing

- **[F3] Battle creation:**

To complete the basic functionalities of the platform we need to implement and test F3 (as described in 30).

Once the features F1, F2, and F3 are created and validated, the platform's fundamental building blocks are in place. The following functionalities can be implemented concurrently without a specific sequence, promoting a parallel development and testing process. This approach accelerates the system's production by allowing teams to work on different aspects.

- **[F4] Allow other educators to crate battles:**

The developing structure is described in the 30.

- **[F5] Student registration to tournament and battle, Team formation:**

The developing structure is described in the 30 and 29.

- **[F6] GitHub Integration:**

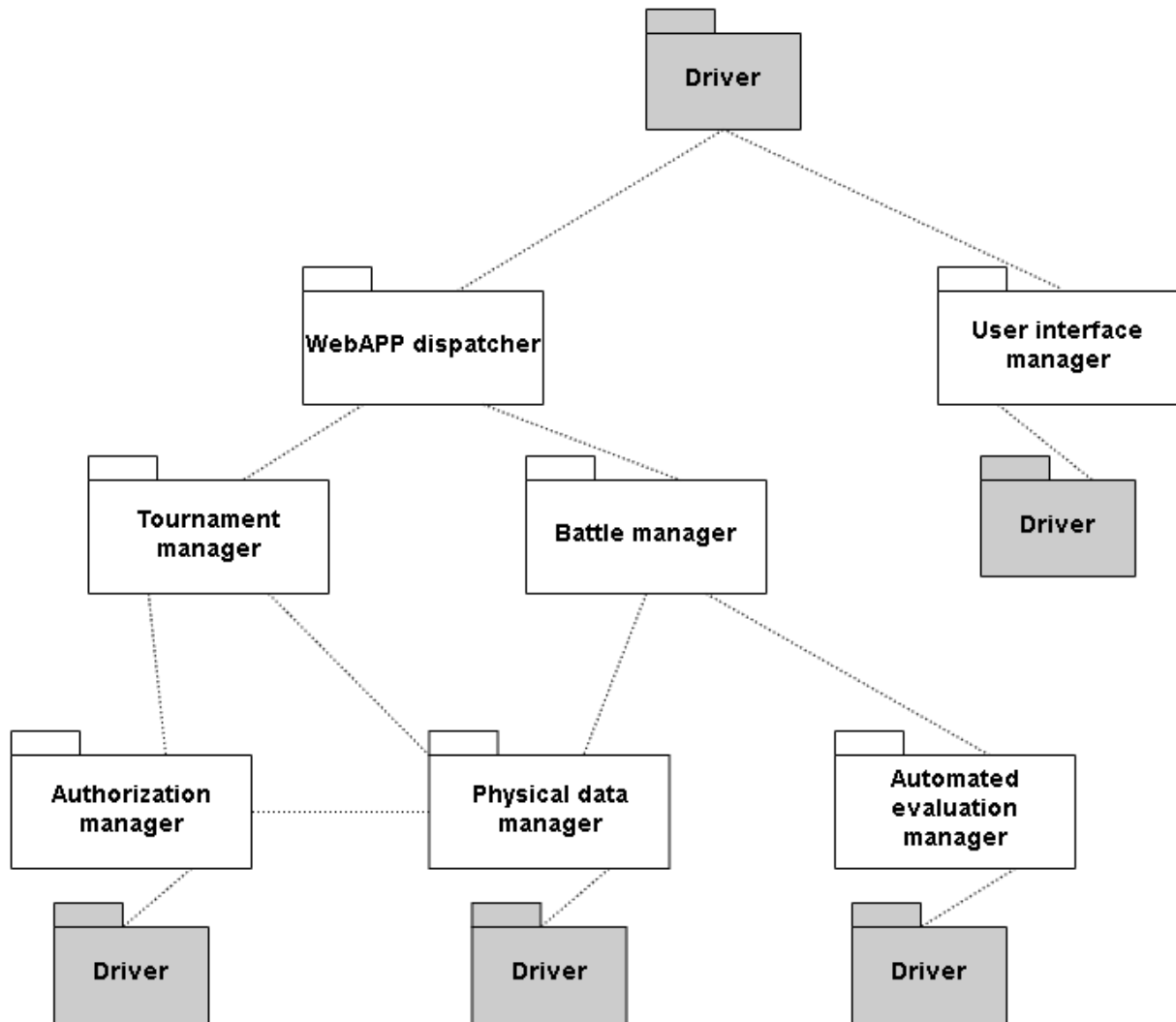


Figure 30: F3 Developing

In 31 is visible that we only need to add the GitHub Integration Manager component to implement this feature.

- **[F7] Manual evaluation:**

The developing structure is described in the 30 and 29.

- **[F8] Tournament closure:**

The developing structure is described in the 30 and 29.

- **[F9] Badge creation and assignment:**

In order to implement and test F9 we need to properly integrate the Badge Manager as described in 32.

- **[F10] Notification system:**

To conclude the implementation phase, the final component to be developed is the notification system. This element needs to be integrated with all components that directly communicate with users through push notifications or emails (see 33).

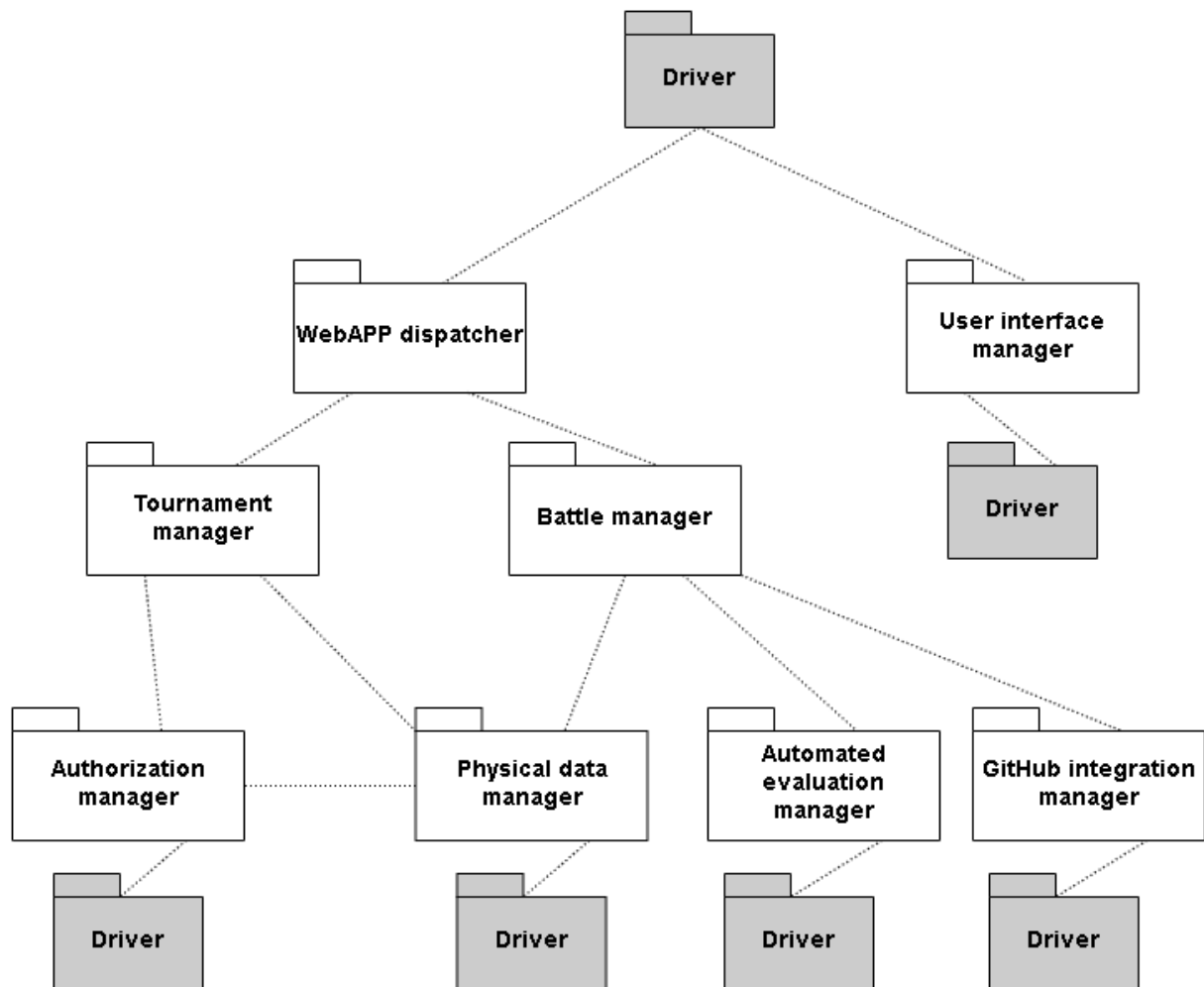


Figure 31: F6 Developing

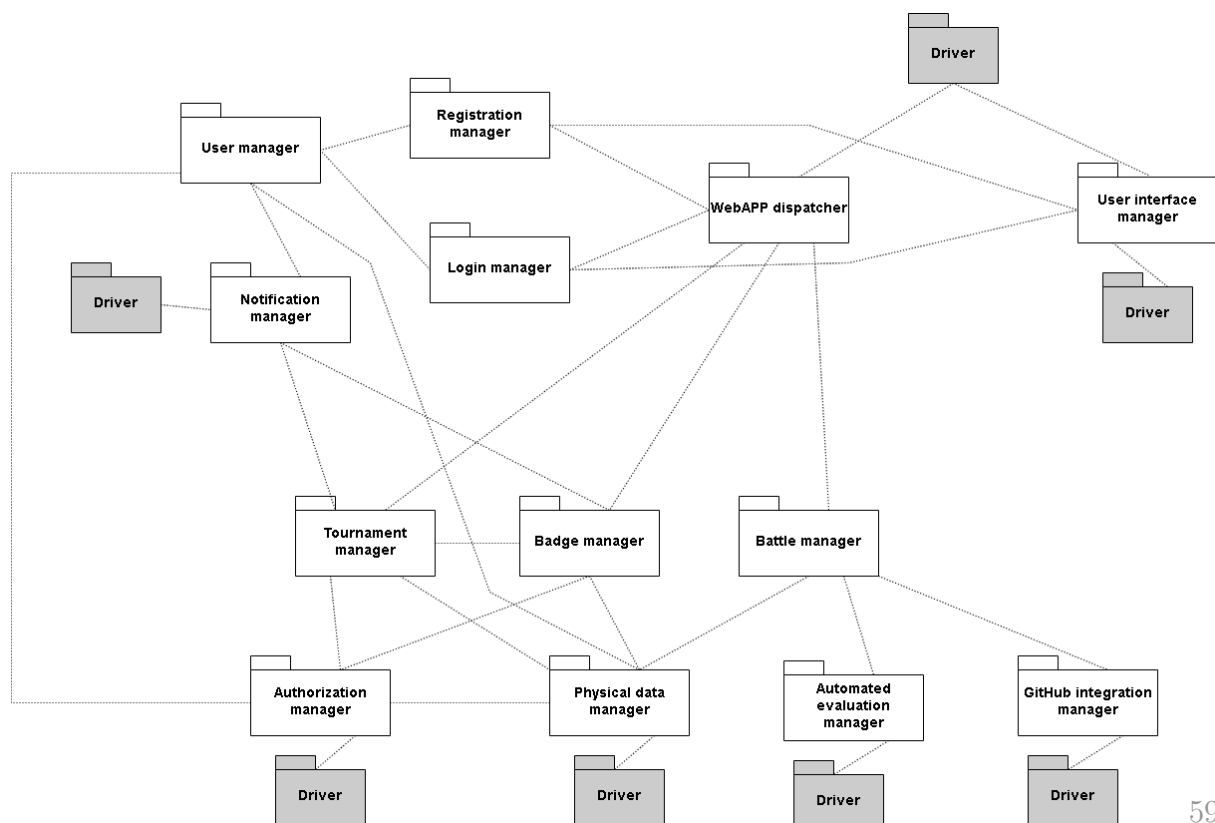


Figure 33: F10 Developing

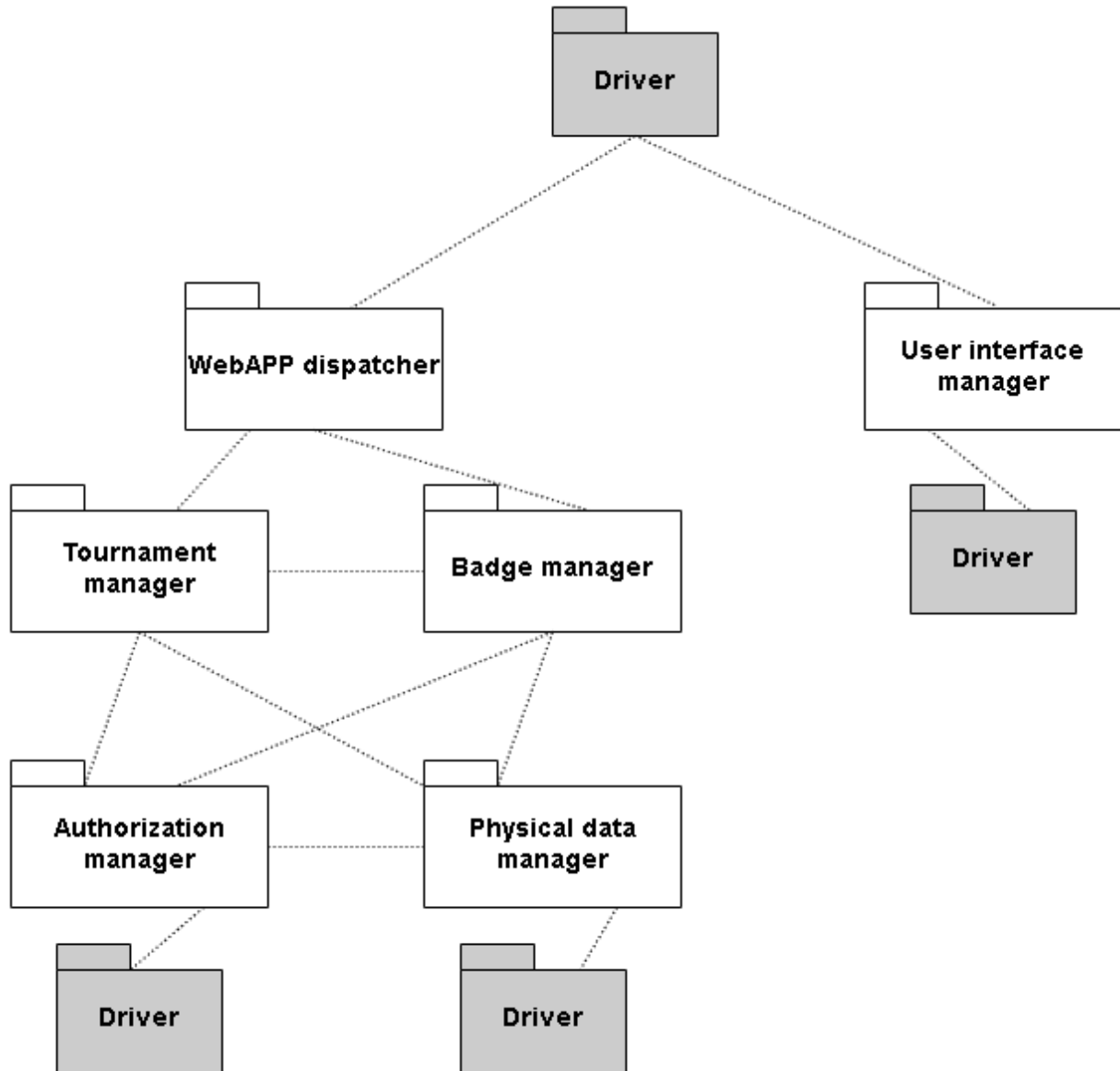


Figure 32: F9 Developing

5.3 System testing

The CodeKataBattle platform undergoes systematic testing, considering various levels of granularity and scopes to ensure the reliability and functionality of its components. In the development phase, individual modules and components go through unit testing in isolation to verify their expected performance. To facilitate this, the creation of Driver and Stub components simulates the behavior of surrounding modules, considering both expected and unexpected scenarios to evaluate robustness.

Subsequently, the integration and testing phase employs bottom-up and thread strategies, aligning with the platform's architectural design. As components are integrated, the focus shifts to validating their collaborative functioning. Once individual components and their integrations have been tested and fixed, the system testing begins to validate adherence to functional and nonfunctional requirements outlined in the RASD.

The system testing phase includes:

- *Functional Testing*: Ensuring the system meets all specified requirements from the RASD. This phase may also expose opportunities to improve user experience with new features.
- *Performance Testing*: Identifying potential bottlenecks, inefficient algorithms, and other performance-related issues that could affect the system's responsiveness, utilization, and throughput. This phase requires an expected workload and predefined performance targets.
- *Usability Testing*: Evaluating how well users can interact with the web application to accomplish tasks. Given the platform's emphasis on user-friendly design, usability testing is essential.
- *Load Testing*: Uncovering bugs such as memory leaks or mismanagement and determining the upper limits of system components. It involves gradually increasing the load until reaching the system's operational threshold.
- *Stress Testing*: Ensuring agile system recovery after failure by intentionally subjecting it to overwhelming resource demands or deprivation.

This testing approach ensures the CodeKataBattle platform's robustness, performance, and user-centric design across various scenarios and usage conditions. It also provides a workflow for continuous improvement and advancement based on testing outcomes.

6 Effort Spent

The following tables contain an approximation of the number of hours spent on the creation of this document by each member of the group. In particular, each row represents the time invested in the designing and redaction of the respective chapter. These tables do not include the time spent on the collective review of the work done by each member.

Chapter	Hours spent
1	2
2	23
3	6
4	3
5	3

Table 8: Alessandro Annechini

Chapter	Hours spent
1	7
2	20
3	2
4	2
5	3

Table 9: Nicole Filippi

Chapter	Hours spent
1	3
2	8
3	2
4	7
5	13

Table 10: Riccardo Fiorentini

7 References

1. Inspirational sites similar to our web application:
 - <https://codecombat.com>
 - <https://www.codingame.com>
 - <https://codebattle.hexlet.io>
 - <https://www.codebattle.in>
2. DrawIO, tool used for sequence diagrams and development diagrams. URL: <https://www.drawio.com>
3. Moqups, tool used for web application's mockups. URL: <https://moqups.com>
4. Specification document: "Assignment RDD AY 2023-2024"
5. DD sample from A.Y. 2022-2023
6. \LaTeX , used to redact the document. Documentation: <https://www.latex-project.org/help/documentation/>
7. Overleaf, \LaTeX online editor. URL: <https://www.overleaf.com/>