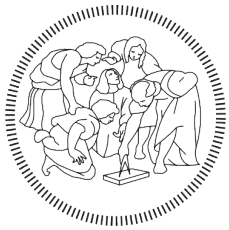


POLITECNICO
MILANO 1863

Design Document

CodeKataBattle

Authors

Name	Surname	ID
Alessandro	Saccone	11013852
Matteo	Sissa	10972783
Sara	Zappia	11016799

Version 1.0 - 07/01/2024

A.Y. 2023/2024

Contents

1 Introduction

1.1 Purpose

The main idea behind the CodeKataBattle platform comes from the term ”kata”, which is a Japanese word describing detailed patterns of movements repeated many times in order to memorize and master them. This kind of exercise is usually applied in learning karate. CodeKataBattle aims at exporting the same learning process also in the domain of coding and programming.

The software described in this document involves two main actors, which are students and educators. Educators can propose code kata battles (or simply battles for brevity), which are problems and teasers to be addressed by coding with a specific programming language, while students can engage in these battles in order to find solutions to them. Code kata battles are published in the context of tournaments in which students compete with each other.

The main goals of this system are tightly related to the users of the application. On the educator’s side, CodeKataBattle allows to easily organize tournaments and coding battles, also for a great number of students. On the student’s side, CodeKataBattle creates an enjoyable and playful environment to foster and improve the student’s coding skills.

1.2 Scope

The CodeKataBattle system is a dynamic web application fostering collaborative coding experiences for both students and educators. On this platform, students actively engage in tournaments and code kata battles to showcase their skills and monitor individual progress. Educators also play a crucial role in taking care of organizing tournaments and designing battles.

CodeKataBattle is integrated with GitHub, to leverage the GitHub abilities to store different versions of a coding project. For every battle generated on CodeKataBattle, a dedicated GitHub repository is created after the registration deadline of the battle expires. From that moment, students are able to push their code solutions for the battle in a personal repository that is forked by the one created as a reference by CodeKataBattle . Every student willing to participate in a battle, also has to autonomously develop a GitHub workflow (through GitHub Actions) in order to fire a notification from GitHub to CodeKataBattle every time a commit is performed on his/her repository. When CodeKataBattle receives a notification from GitHub, it pulls the source code from the remote GitHub repository and automatically computes the score to assign to that solution based on some aspects like number of test cases passed, time went by from the beginning of the battle and static analysis on the source code run by an external static analysis tool.

Both tournaments and battles have rankings, that are kept updated anytime by CodeKataBattle until the battle or the tournament ends. When a battle ends, all students involved in the battle are notified, and the enclosing tournament ranking updated accordingly. When a tournament is closed by the educator who created it all students that took part in the tournament are notified. The final ranking of the tournament is also published and the badges are assigned to students who deserve them.

This document offers a finer insight on the design decisions that have been taken in order to implement the CodeKataBattle platform. From a very high-level standpoint, CodeKataBattle is first of all a web application. This choice is relevant because it allows a straightforward and easy access to the functionalities of the application from any client device equipped with an internet connection.

The internal architecture follows the microservices architectural style, as all major functionalities of the system are scattered among different programs. The microservice architectural style promotes a more structured way of developing an application, by loosely coupling the different components and therefore boost maintainability and scalability. It also has a great impact on the development process when working with a team of developers, as each microservice can be designed independently from the others. From a deployment point of view, the CodeKataBattle platform can be described as a three-tier architecture, since a single server runs all microservices, clients can connect to that server and the microservices can contact a remote database server for persistent data storage and manipulation. This structure of the system is shown more in detail in the following of the document. As for the data layer, the CodeKataBattle is built on a shared database, that all microservices can access. This way of thinking the data layer simplified data management, especially in situations where the data content managed by different microservices is often times related and logically interconnected. Finally, some architectural patterns are employed in the design to allow an effective communication between microservices. CodeKataBattle microservices expose RESTful APIs to receive requests from the outside and respond with data (or to provide some sort of computation on data). An event-driven approach is also employed internally to improve performance and reduce coupling between microservices. A Model-View-Controller pattern is also adopted in order to separate concerns between the software components that have to manage incoming requests and showing graphical user interfaces.

All these architectural choices are just mentioned here to provide an overview of the system, they will be better explained and unpacked down the line of this document.

The following image shows the major components of the CodeKataBattle system.

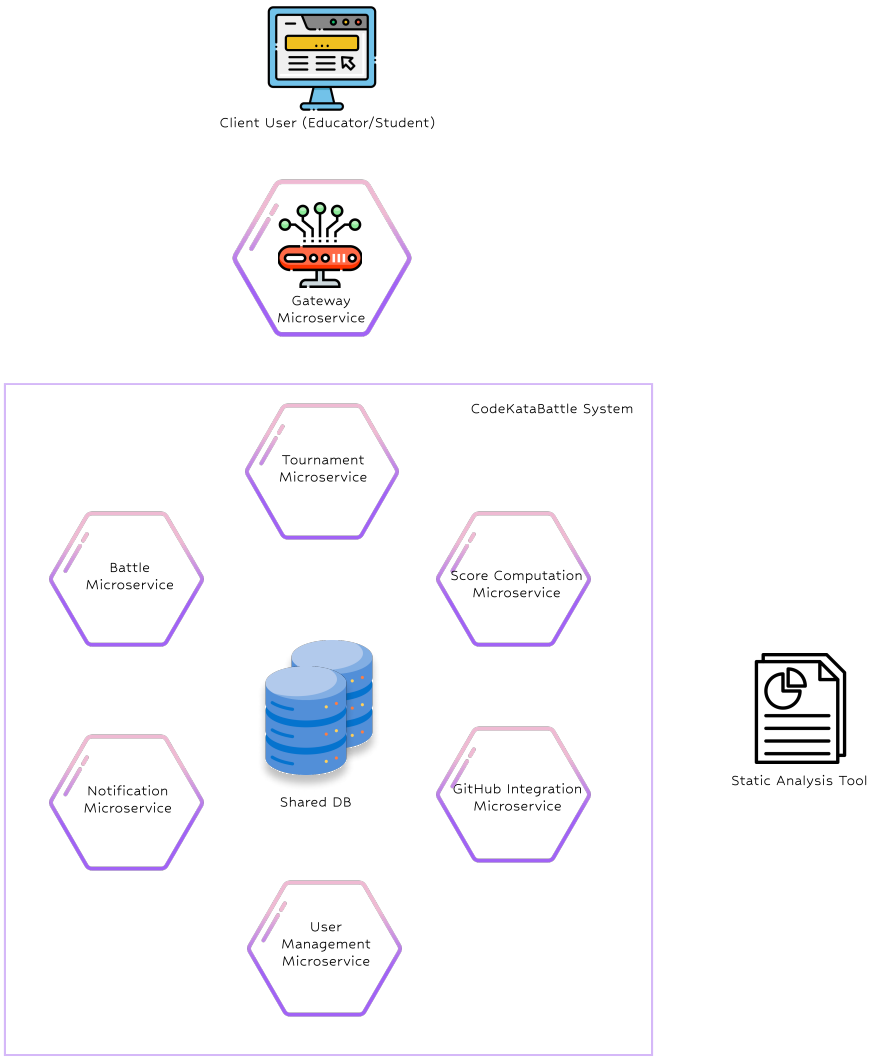


Figure 1: High-level view of the major components

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

A brief list of the most meaningful and relevant terms and synonyms used in this document is reported here, in order to make reading process smoother and clearer:

Term	Definition
User, Actor	A user of the CodeKataBattle system, which includes both educators and students.
Educator, Professor, Teacher	A type of user of the CodeKataBattle system.
Student	A type of user of the CodeKataBattle system.
System, Platform, Application, Machine, Software	All synonyms to indicate CodeKataBattle.
Battle, Coding challenge, Problem, Teaser	A code kata battle offered on CodeKataBattle.
Consolidation Stage	Phase occurring at the end of a battle (after submission deadline) in which the educator that created the battle is asked to personally assess the students' code solutions.
Registration Deadline	Valid for both tournaments and battles, it is the date by which students are asked to subscribe to a tournament or a battle on CodeKataBattle.
Submission Deadline	In the context of a battle, the submission deadline is the date after which no additional code solution for the battle can be submitted to the system.
Ranking, Classification	Ordered list of teams or students based on achieved scores.
Push	Upload code solutions on GitHub repository.
Pull	Download code solutions from GitHub repository.

Term	Definition
Fork	Creating a personal copy of a GitHub repository, which inherits the structure and content of the original repository at the moment the fork is performed.
GitHub workflow	File containing code (usually written in YAML) to instruct GitHub on actions to take when certain events happen in a GitHub repository. In this project, workflows are used to fire a notification from GitHub to CodeKataBattle when a commit is performed on the repository.
GitHub Actions	Pre-configured actions that can be employed inside GitHub workflows to make GitHub carry out specific operations based on the specific needs of the user.
Build automation scripts	Files containing code useful to automatically build a project or (in this specific case) the students' code solutions for a battle.
Test cases	Files containing code useful to automatically test the students' code solutions for a battle.
Evaluation Criteria	Indicate the set of parameters that are accounted for by an external static analysis tool in order to assign a partial score to the students' code solutions for a battle. Examples of Evaluation Criteria might be security, reliability, robustness and so on.
Badges, Rewards	In a tournament, Badges or Rewards are prizes that are assigned to students when the tournament ends based on personal achievements of the student in that tournament.
Architectural style	A collection of principles that shape or govern the design of a software application.
Structure, View	In the context of software architectures, a structure or a view is a perspective or point of view that is highlighted when describing the software architecture. A structure or a view is usually characterized by a specific set of elements/components and types of interactions between them.

Term	Definition
Event-Driven Architecture	Event-driven architecture is a software design pattern where the flow of the system is determined by events. Components or services within the system communicate asynchronously by producing, detecting, and responding to events, promoting decoupling and scalability in distributed systems.
Component	In the context of the Component & Connector view of a software architecture, a component is a replaceable piece of software that is responsible for a specific functionality of the system that it is part of
Participate, Join, Sign up, Take part, Subscribe	Synonyms employed to describe the action of students engaging with tournaments or battles on CodeKataBattle.
Create, Publish, Generate	Synonyms to express the action of educators creating new battles or tournaments on CodeKataBattle.
Evaluate, Assess, Assign score	Synonyms to express the action of associating a score to a battle code solution.
Submit, Hand in	Synonyms describing the action of a student giving to the system a code solution to a battle.
Show, Display	Synonyms to express the responses of the system to user requests. CodeKataBattle shows or displays something to the users.

1.3.2 Acronyms

A list of acronyms used throughout the document for simplicity and readability:

1. RASD - Requirements And Specification Document
2. CKB - CodeKataBattle
3. UI - User Interface
4. GH - GitHub
5. UML - Unified Modeling Language

6. DB - Data Base
7. DBMS - Data Base Management System

1.4 Reference Documents

Here's a list of reference documents that have been used in order to shape the Design Document of the CodeKataBattle system. In the following, all external sources of information that have contributed to the fabrication of this document are mentioned.

1. Stakeholders' specification provided by the R&DD assignment for the Software Engineering II course at Politecnico Di Milano for the year 2023/2024.
2. IEEE Std 1016™-2009, IEEE Standard for Information Technology—Systems Design—Software Design Descriptions.
3. ISO/IEC/IEEE-42010, Second edition, 2022-11, Software, systems and enterprise-Architecture description.
4. UML specifications, version 2.5.1.
   Link: https://www.omg.org/spec/UML/2.5.1/About-UML

1.5 Document Structure

The following of the Design Document for the CodeKataBattle project is divided into four major sections, which aim at describing the overall design of the system in order to support the development and implementation of the final product.

Section 2, Architectural Design, is the most design-relevant one. Its main objective is to provide the reader with a series of views or structure of the software architecture that has been selected for CodeKataBattle . Indeed, section 2 is further divided into:

- Overview: provides a high-level description of the most important components of the system and their interactions (in an informal notation).
- Component view: this section provides the first structure of the software architecture, which is the Component & Connector structure, useful for illustrating the relevant components of the system from a dynamic perspective and the way they collaborate to achieve the final goals. In this section, UML components diagrams are largely employed to convey these ideas and notions.
- Deployment view: this section is about another structure of the software architecture, related to the deployment of CodeKataBattle . The most important objective of this chapter is to show the mapping between the software components of CodeKataBattle and the hardware devices that will physically execute the app. UML deployment diagrams are a great tool to unpack this topic.
- Runtime view: this section employs sequence diagrams in order to explain the flow of events and interactions within the system's components, in a consistent way with the previous chapters.
- Component interfaces: in here it is possible to provide a complete specification of the most important methods and functions that each interface exposed by the various components of the system must show.
- Selected architectural styles and patterns: a revision of the main architectural styles and patterns with a more in detail explanation of the reasons why they have been chosen for this project.

Section 3, User Interface Design, is about user interfaces (UI). More specifically, this section is concerned with giving some guidelines to UI designers on how of the final application should look like (color themes, placement of most relevant UI items...), as well as on the logical role that these interfaces have in the development (what functionalities they provide to the user).

Moving on, **section 4 (Requirement Traceability)** is dedicated to a matrix that shows how the requirements that have been drawn and derived for CodeKataBattle map onto the components that have been highlighted in the previous sections of the document.

Finally, **section 5 (Implementation, Integration and Test Plan)** is concerned with illustrating the implementation strategy adopted (order of implementation of components), the integration strategy (how to integrate new sub-components into the application under-development), and the test strategy for the integration of different components in the system.

2 Architectural Design

2.1 Overview

2.1.1 High Level View

This section is dedicated to an overview of the architectural elements composing the CodeKata system and their most important interactions.

The CodeKataBattle system is built with a microservices architectural style. A microservices architectural style is a type of software architecture in which the application is developed as a collection of services, which are units of functionalities designed to cooperate with one another in order to achieve the goals for which the system is developed. In order to make this description as clear as possible, a high-level view of the system (with its microservices) is illustrated in the following image:

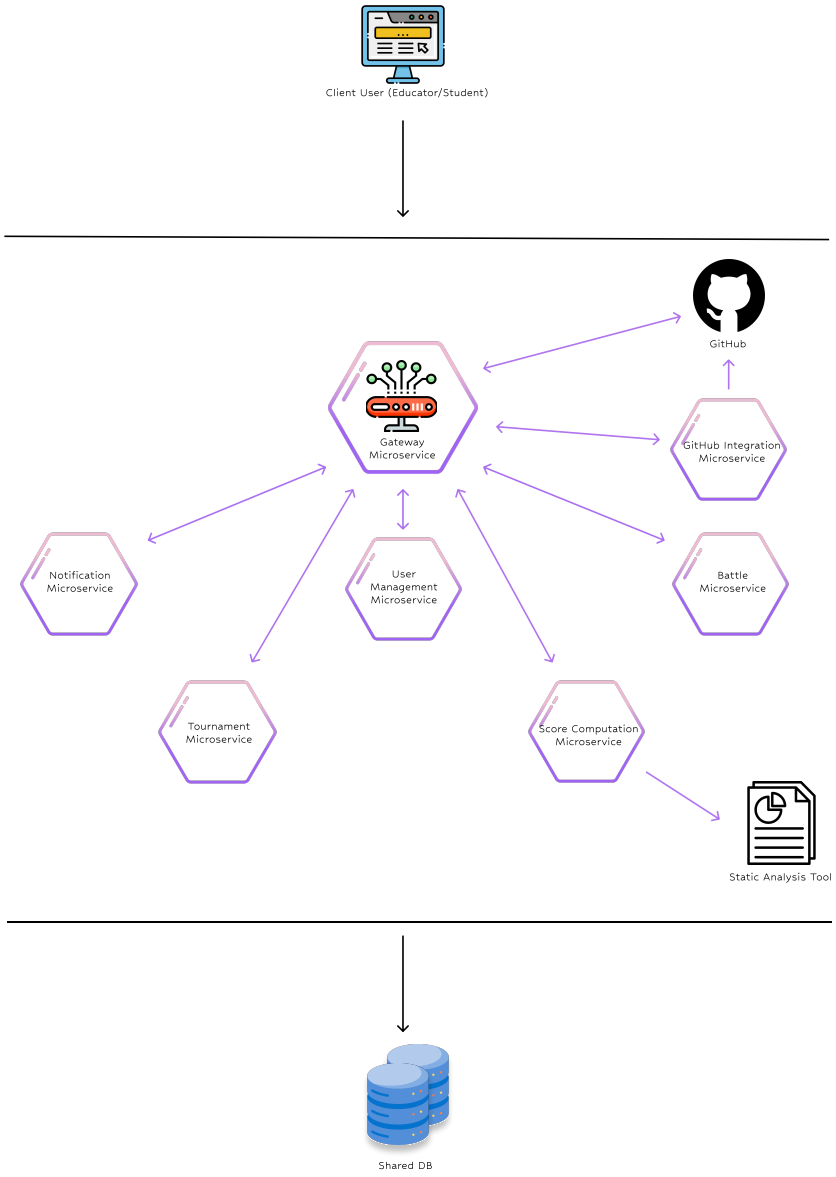


Figure 2: Overview of microservices and main interactions

From a very high-level perspective, all the previously mentioned elements are also represented in this figure. The client is the first icon on the top, the CodeKataBattle system is represented as a set of hexagonal microservices, which also leverage external platforms, like GitHub and the Static Analysis Tool. Finally, at the very bottom, the data layer is shown with a database icon.

There are seven microservices in the CodeKataBattle system, each one offering some functionality, described as follows:

- **Gateway Microservice**: this microservice is responsible for a couple of things. First off, the Gateway Microservice handles the authentication process of users through GitHub. From the figure, it's easy to spot the relationship between this microservice and GitHub external actor. Moreover, the Gateway handles all the incoming users' requests and dispatches them to the specific microservice that has to handle them. All users requests pass by the Gateway, as well as all responses of the system traverse the gateway to reach the users. In order to grant availability and performance, this microservice can be replicated multiple times on the server side of the application.

- **User Management Microservice**: this microservice is responsible for the management of users' data and profiles. Thus, the User Management Microservice controls the usernames of students and educators, as well as providing the user profile page every time a user accesses CodeKataBattle .

- **Tournament Microservice**: this microservice grants control over the tournaments and manages tournaments data in the shared database. It is responsible for the creation of a tournament from an educator, as it provides the user interface to do that. Besides, it maintains data on open tournaments, closed tournaments, tournaments rankings and so on. It is also responsible for providing the tournaments' home pages when requested.

- **Battle Microservice**: this is the microservice responsible for managing battles. It provides the interface to create a new battle to an educator requesting it, maintains all data related to battles and delivers the battle home page when requested. It is also responsible for handling teams of students, as well as keeping the rankings of battles updated.

- **GitHub Integration Microservice**: this microservice is in charge of handling all the API requests involving GitHub, apart from the login phase (delegated to the Gateway Microservice). More specifically, this microservice is very helpful when CodeKataBattle has to create a new GitHub repository for a battle, or when students code solutions must be downloaded from GitHub.

- **Score Computation Microservice**: the purpose of this microservice is to calculate the score to be assigned to a student code solution for a battle. In order to do so, this microservice uses the build automation scripts and test cases provided by the educator, calculates the time passed from the beginning of the battle and leverages the Static Analysis Tool as well.

- **Notification Microservice**: this microservice is concerned with all the notifications needed for tournaments and battles. The Notificaiton Microservice listens to events in the system (event-driven development) and produces notifications for students or educator.

As for the interactions between these microservices, they will be addressed in much greater detail in the following section of this document. For now, it is relevant to know that all communications going from the client to the CodeKataBattle platform and vice versa traverse the gateway. Also the

several user interfaces which CodeKataBattle is composed of are provided by different microservices (concerned with the proper data related to the interface), but the transmission of these interface is always mediated by the gateway.

All microservices also expose a **RESTful API**, which is the access point to the data and functionalities they offer. Thus, microservices will interact with one another through this interfaces when necessary, and these details will be shown in the following views.

Furthermore, every microservice has access to the same **shared database**, thanks to the interfaces exposed by the DBMS system, but each component covers different types of manipulations and computations on this shared data space, which are dependent on the functionalities offered by the microservice in question.

The interactions that occur inside this system also rely on an **event-driven pattern**. This pattern allows components (microservices) to asynchronously produce events on an Event Bus (or Event Queue) and other components to consume (read) those events and take actions accordingly. Further details will be supplied in the following of the Design Document.

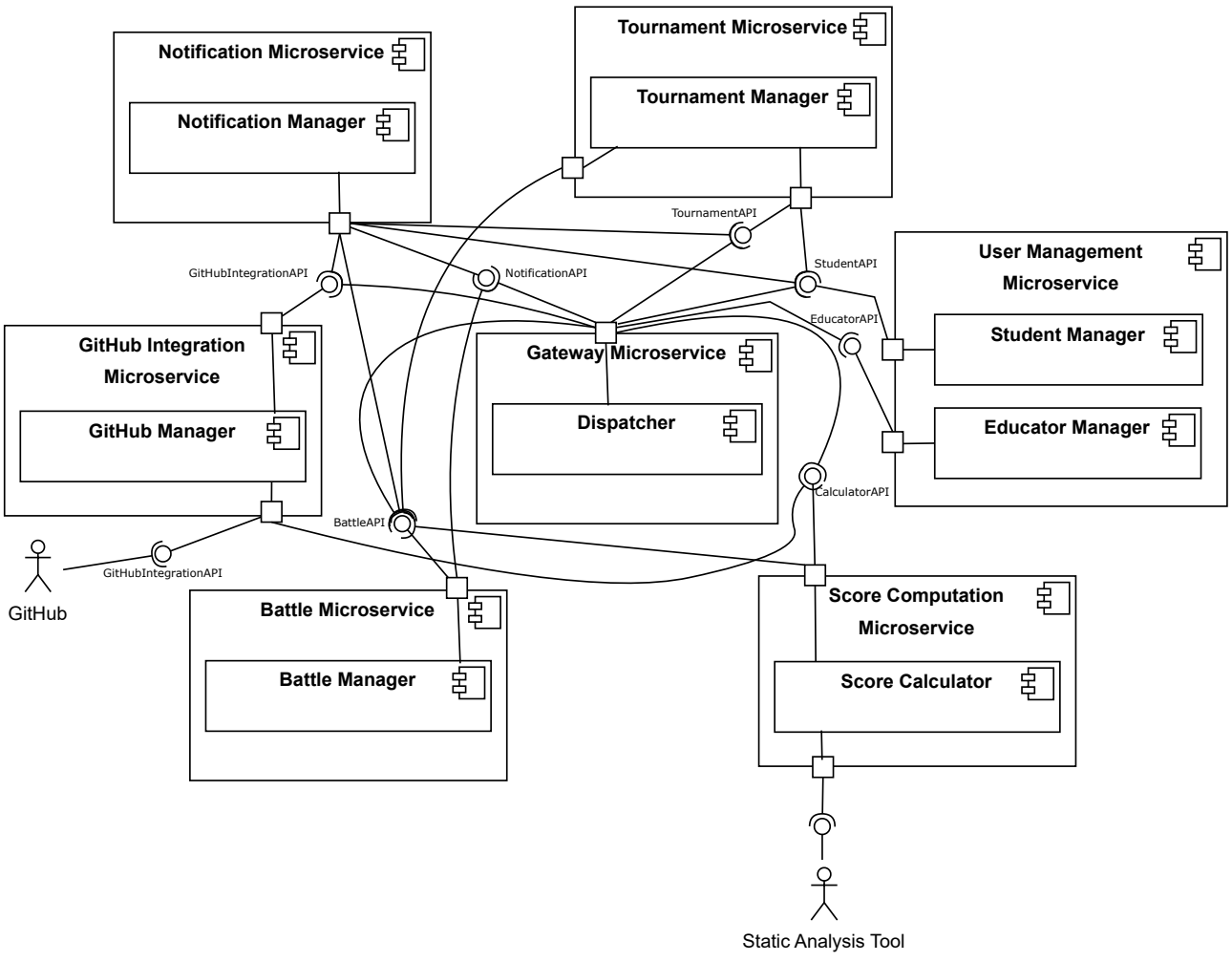
2.2 Component View

This section aims at illustrating the various components that make up the CodeKataBattle system. In the following, UML Component diagrams are employed to illustrate the logical software elements that collaborate in order to achieve the goals set for the system to be developed.

Since the structure of components of the CodeKataBattle system wouldn't fit into a single diagram, multiple representations are supplied to make the document as clear as possible and avoid cluttering. The decisions regarding how to divide the diagrams of the Component View have been taken based on the idea of grouping together components that frequently interact with each other. It is important to mention that although the explanation makes use of separated graphics, the system is a single one and the elements represented here in different diagrams will be part of the same system and will take part in the overall functioning of the platform together.

2.2.1 RESTful APIs component diagram

One of the first characteristics that have been mentioned in the introductory section about CodeKata-Battle is that microservices expose RESTful APIs in order to receive and respond to commands, expose data and collaborate with one another. This view aims at describing from a high perspective the ways microservices exploit these APIs to work.



Let's break down the main ideas that the diagram tries to convey. This UML component diagram is employed in order to show the RESTful APIs that are offered by each microservice in the system. These APIs provide specific computations on data related to the service offered by the microservice in question. For instance, the Tournament Microservice will expose an API that works on the tournaments data inside the shared database and exposes relevant information on tournaments.

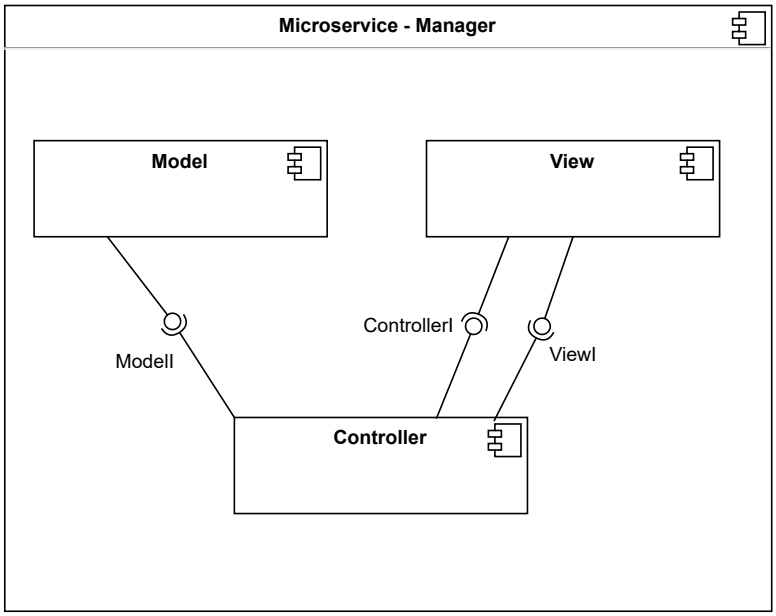
The gateway is the entry point for all users' requests. The Dispatcher component inside the Gateway is responsible for delivering the users' requests to the correct microservices that will have to handle them. This is why the Dispatcher "uses" all the APIs exposed by the other microservices.

For some kinds of computations, the various microservices that compose CodeKataBattle may also exploit each other through their RESTful APIs. In this diagram, these interactions are shown. For instance:

- The Notification Manager needs to query the Tournament manager and the Battle manager every time it has to retrieve the list of students subscribed to a tournament or to a battle (in order to deliver the notification to a specific set of students).
- The Notification Manager has to query the Student Manager when the list of all students of CodeKataBattle is needed (for example to notify all students of a new tournament available on the platform).
- The Notification Manager requires access to the data handled by the GitHub Manager in order to retrieve the link to the remote repository of a battle and compose the notification for all students subscribed to that battle (when the registration deadline of the battle passes).
- The GitHub manager component uses the CalculatorAPI in order to pass the downloaded source code of a new student solution to the Score Calculator component that will in turn automatically compute the score to assign to it.
- The Score Calculator must exploit the BattleAPI in order to request an update on the battle ranking every time a new score is available.
- The Tournament Manager requires access to the Student Manager data in order to assign badges when the tournament is closed.
- The Battle Manager leverages the NotificationAPI in order to produce individual notifications when a student invites another student to join a battle together as a team.

This diagram has to be read and interpreted also accounting for the event-driven paradigm that is adopted in the project. Some of the interactions and flows of events inside the system happen in an asynchronous manner with an event-driven approach that is further described in the following views. That's why some interactions, that might seem missing here are actually handled in an asynchronous way to increase the performance of the overall system.

The last important point to make about this diagram is related to the "Manager" components inside the various microservices. These components internally implement a Model-View-Controller pattern to provide their services. The following representation shows a component diagram that only focuses on the internals of the "Manager" component for a generic microservice:



The Model component is responsible for organizing and managing data. Each microservice in the CodeKataBattle system is specific for a portion of the data domain. For instance, the Tournament Microservice only focuses on managing and providing information on tournaments. Therefore, each Model component inside the Manager components take care of maintaining consistent data with the DBMS, accessing the DBMS when needed and so on.

The View is the component offering the user interface. Since different microservices focus on different types of information in the CodeKataBattle ecosystem, the several user interfaces are spread across the microservices and managed through the View components.

The Controller is the ”bridge” between the Model and the View. This component is responsible for:

- Receiving the requests from the View component when the user interacts with the user interface.
- Performing the correct computation based on those requests.
- Manipulating data through the Model, propagating the users' requests to the data layer.
- Updating the View if the computation resulted in some changes of the user interface.

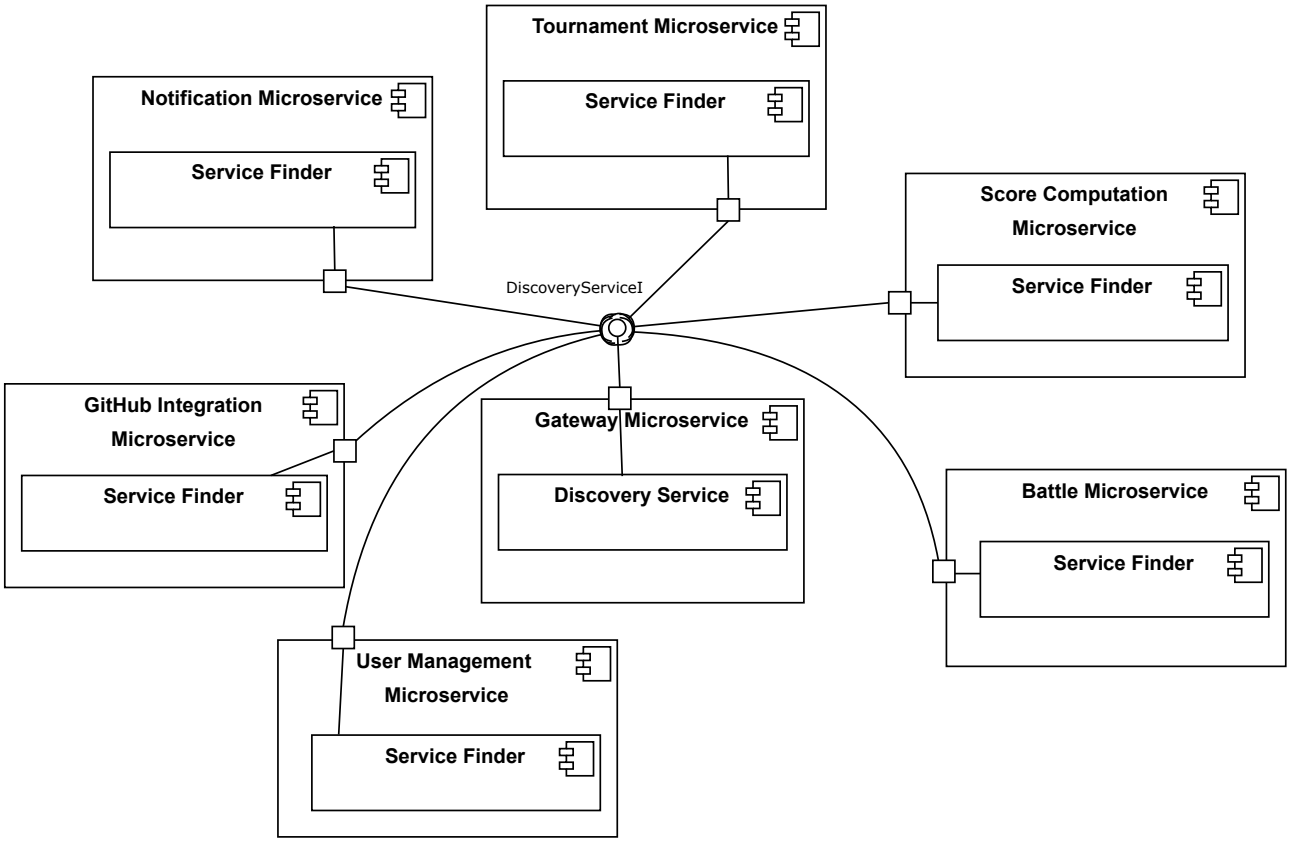
In the following of the document, the Manager component will be used in a more general sense to capture the behavior and functionality of all this internal architecture.

2.2.2 Service Discovery component diagram

This very simple component diagram shows an important service that is offered by the Gateway microservice to allow all microservices to locate each other and therefore collaborate. This service is called ”Service Discovery” and it consists in maintaining a register (inside the Gateway microservice) of active microservices.
In this way:

- A new microservice brought up can register itself in order to be localized by the other microservices.
- All microservices can contact other active microservices in the system to advance some requests.

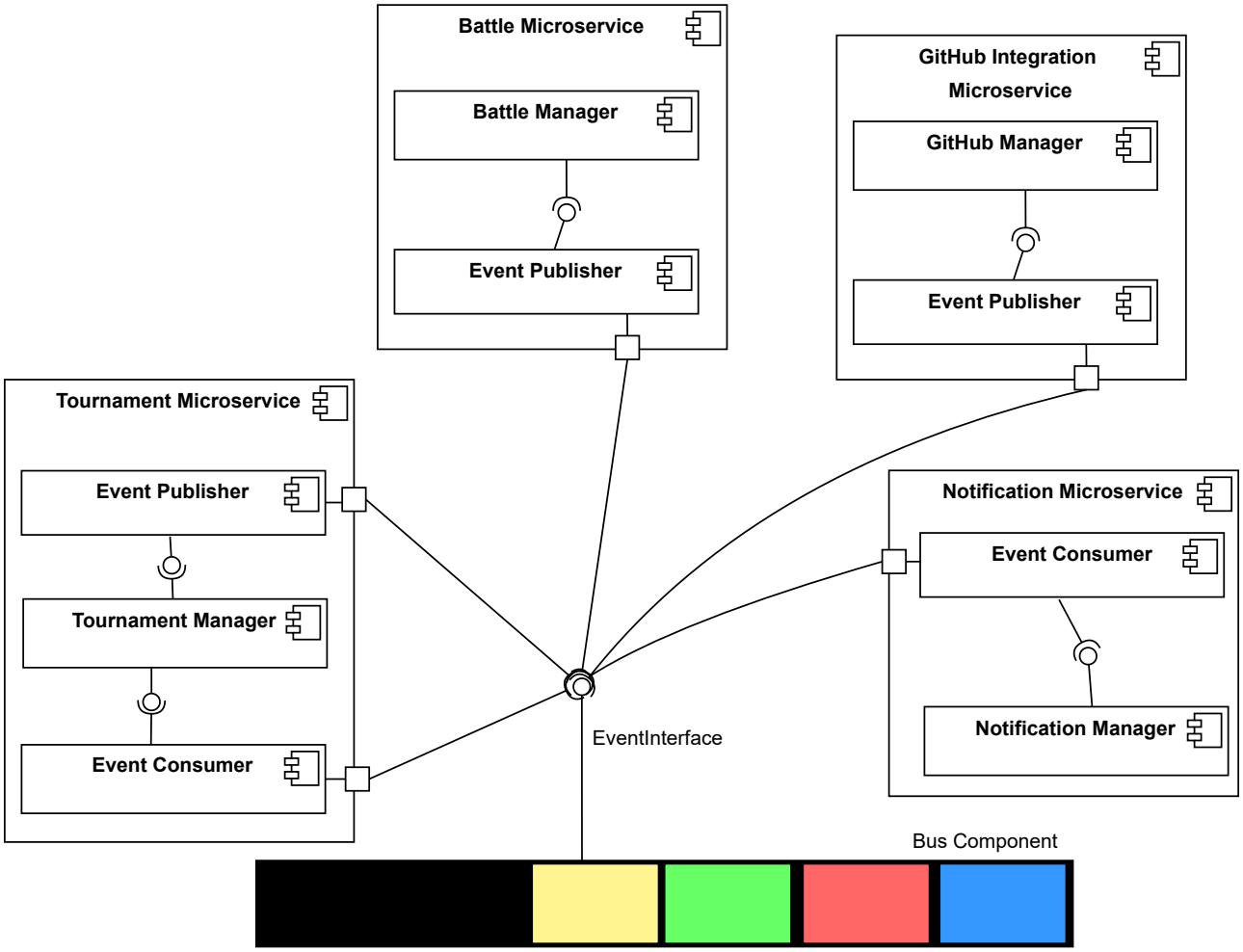
Here’s the diagram:



The interpretation is quite simple. The Gateway Microservice contains a component called Discovery Service, which is the software component that implements the logic to offer the discovery service. The Discovery Service component exposes an API (DiscoveryServiceI), that can be used by all other microservices in order to locate each other.

2.2.3 Event-driven pattern components

This section illustrates all the components that are needed in the system in order to implement an asynchronous, event-driven communication between microservices. This design choice has been taken in order to lighten up the system in specific interactions between microservices that would otherwise require a computationally expensive synchronous communication.
The following diagram shows the major components that help achieve an event-driven architecture:



The most important element is the Bus component, which can be thought of as a queue of messages or events. Microservices in the CodeKataBattle system can either produce events (i.e. push events on the queue) or consume events (i.e. read events on the queue) and take actions accordingly.

It is important to mention that this representation is purposely general and not tied up with any specific software product, framework or implementation. The concrete way in which this design is achieved is left to the implementation of the product.

From the diagram, it is possible to see that only some microservices are illustrated, which are the

only one exploiting this mechanism to communicate and interact. Inside these microservices, there is an Event Publisher component if the microservice has to be able to publish events on the Bus, and there might be an Event Consumer component in case the microservice has to be able to read from the Bus the events that are queued.

The Tournament Microservice, for instance, has both the Event Publisher component and the Event Consumer component. Indeed, the Tournament Microservice has to publish an event every time a new tournament is created or closed. At the same time, the tournament has to be able to read from the bus when a battle is finished, to take actions accordingly.

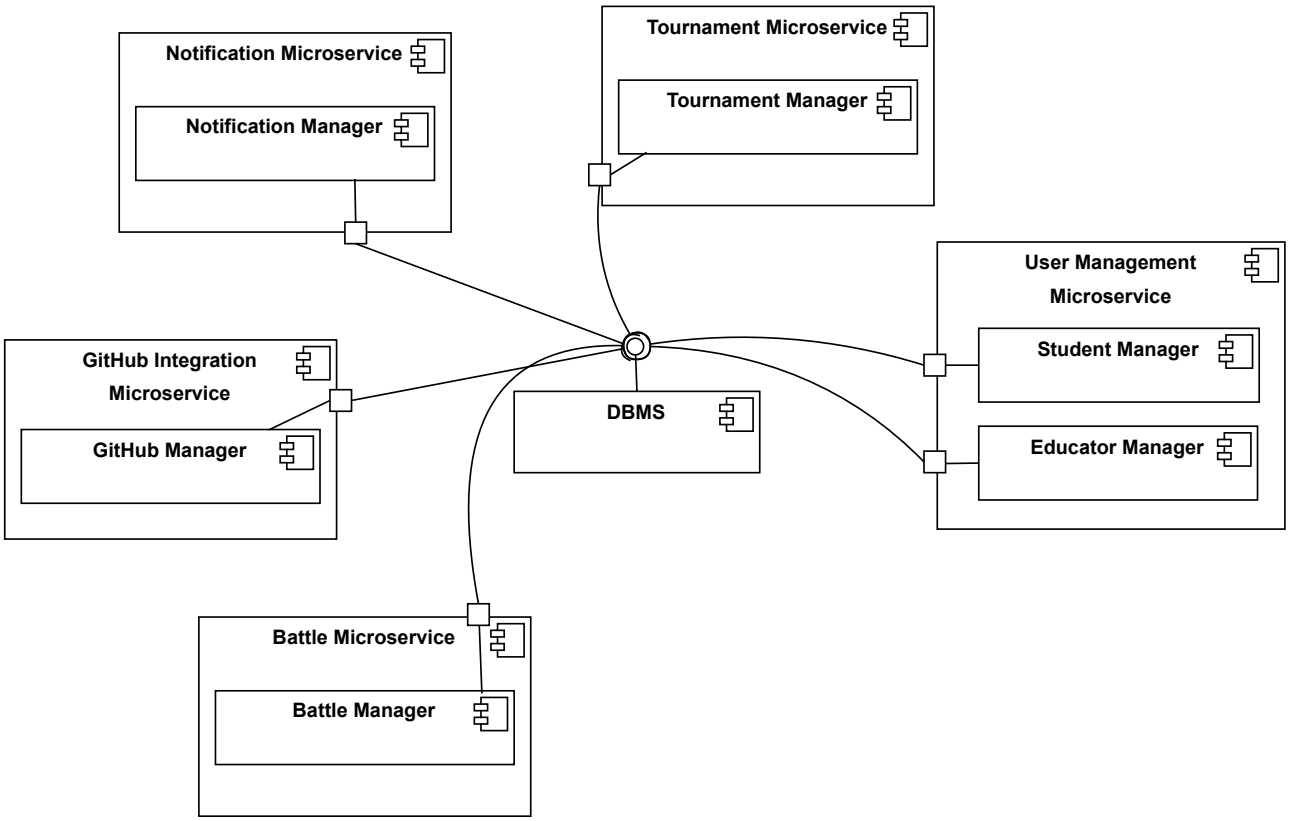
The Battle Microservice only publishes events when the battle is created and terminates.

The GitHub Integration Microservice publishes a new event every time the GitHub repository for a battle is created, since a notification to all the students subscribed to that battle has to be fabricated.

The Notification Microservice is the one always listening to upcoming events in order to fabricate the correct notifications for the users of the CodeKataBattle platform.

2.2.4 Data layer access component diagram

This diagram proposes a convenient view to see how the access is performed by the various microservices on the shared data layer (shared DBMS).



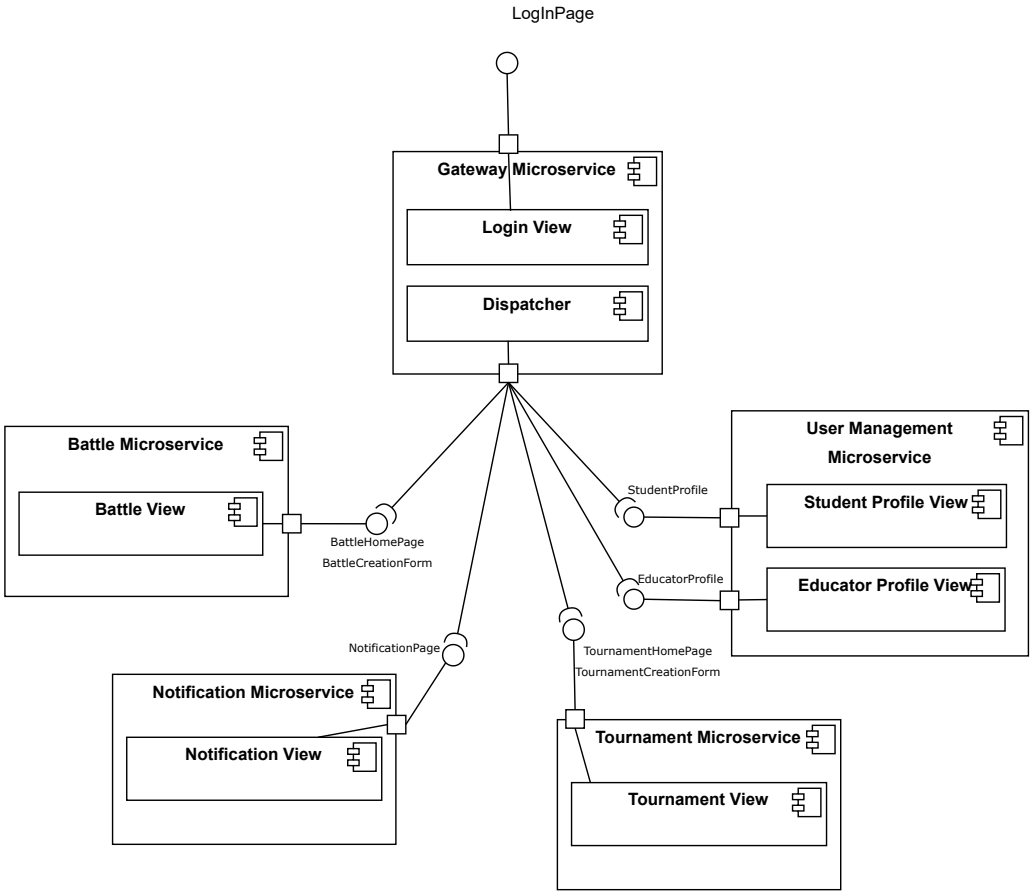
The DBMS exposes an interface that is employed by all microservices to work with the persistent data stored in the DBMS.
Looking at every microservice individually, it is possible to notice that they all are equipped with a ”Manager” component, which is responsible for manipulating and accessing data.
Every microservice of the CodeKataBattle application is responsible for a different section of the data domain. For instance, the Tournament microservice only works on the data stored in the shared DBMS that is related to tournaments and offers to the other microservices all the relevant information about tournaments that might be needed.

2.2.5 User interfaces component diagram

This diagram proposes a convenient view of the system for what concerns user interfaces. The design choices that have been taken in the CodeKataBattle system as concerns the user interfaces can be summed up in this way:

- The different interfaces that are offered to the user are not gathered in a single component, but they are scattered across the various microservices. This decision is due to the fact that different UIs offer different views on the data of the application. Therefore, some microservices (working on a specific section of the data domain) might be more suitable for providing a specific user interface than others.
- All the times a user interface is offered to the user, the Gateway Microservice handles the transmission of data, standing in between the user and the access to the internal microservice. Therefore, all the UIs content passes by the Gateway before reaching the client user.

This diagram tries to illustrate the two points of the above list, by representing a View component inside each microservice that is responsible for building and offering specific user interfaces when required. The Gateway microservice intercepts the user's requests and when a new UI is needed, it contacts the correct microservice that will supply it. It has to be mentioned the fact that this UML diagram is not a standard one, in the sense that the interfaces exposed by the components in this view are not sets of methods but UIs. Nevertheless it is intended as a beneficial illustration to show part of the MVC pattern (with the View components inside the microservices) and the distribution of UIs in the system.



2.3 Deployment View

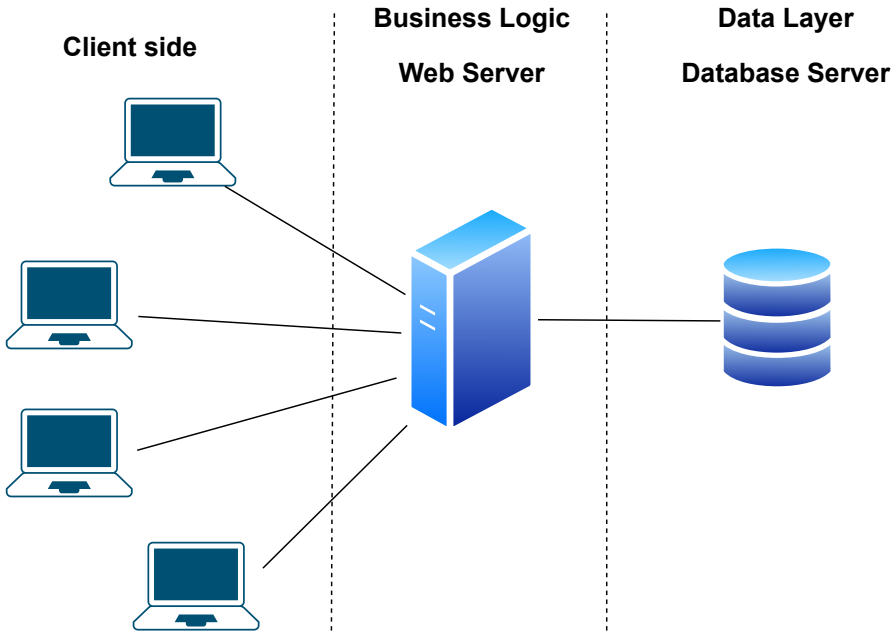
The deployment view addresses the how and where of the CodeKataBattle system's execution, examining the distribution of components, the orchestration of services, and the overall infrastructure that supports the software. In other terms, this section is dedicated to a mapping between software artifacts that implement the logic that makes CodeKataBattle work and the hardware devices that are needed to concretely execute them.

This document aims to provide a comprehensive understanding of the architectural choices that facilitate efficient resource utilization, scalability to meet varying workloads, and the ability to adapt to evolving operational requirements.

As in the previous sections, this view of the system is provided through multiple diagrams that go deeper and deeper into the details of the deployment environment. These diagrams are not exclusive to one another, rather they should be seen as a hierarchical illustration of the same concept, from a more high-level approach, to a more detailed insight.

2.3.1 High-level Deployment View

This diagram shows the bare minimum set of components that are needed to illustrate the deployment view for the CodeKataBattle system.



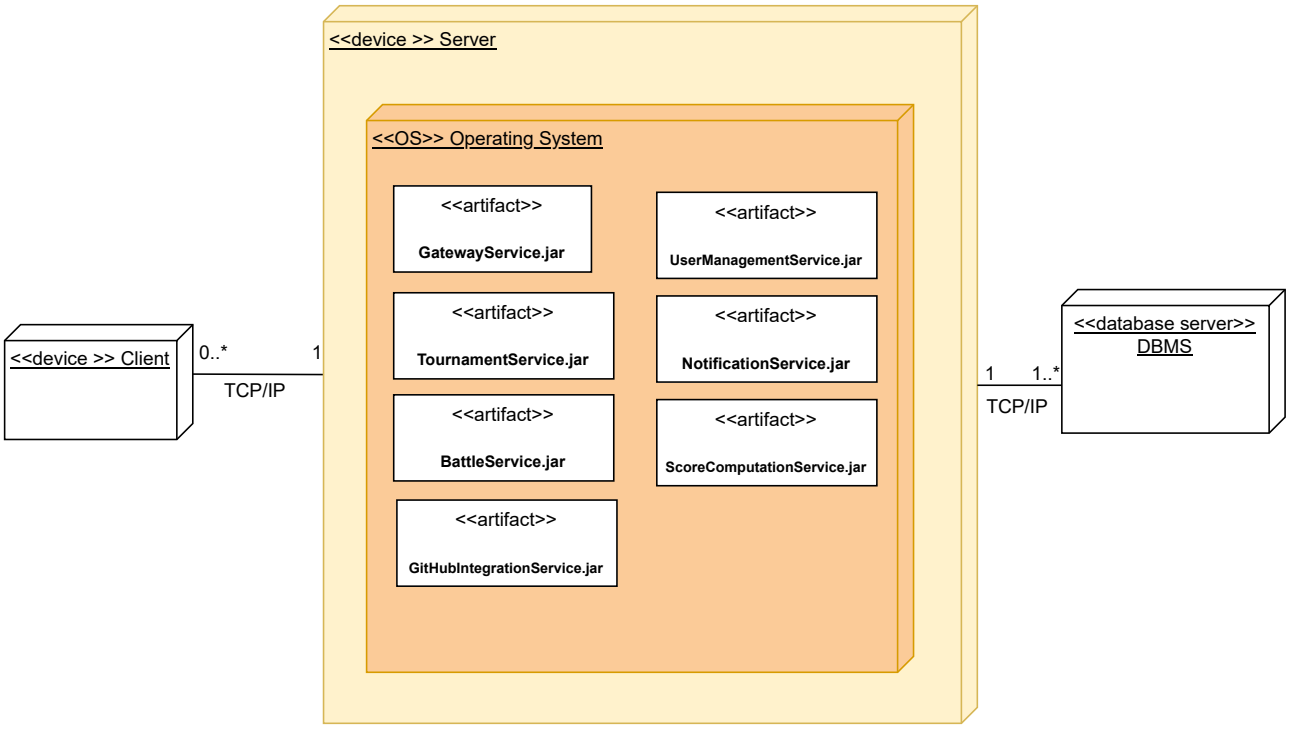
As it is possible to see from the figure, from a deployment perspective, the CodeKataBattle platform can be classified as a three-tier architecture. The three distinct areas in the illustration represent the three tiers (layers) of the architecture, more specifically the client users (on the left), the server (in the center) and the data layer (on the right).

All users are equipped with a device with internet connection that is able to send and receive requests to the server hosting CodeKataBattle . The entire CodeKataBattle is hosted on a single remote server, where all microservices are launched as separate programs, interacting with each other. The data layer can be implemented with a remote Database server that provides persistent data support to the whole system.

This deployment architecture is very convenient for a series of reasons. First off, it allows for a complete separation of concerns between the various layers, in particular between the business logic of the application (hosted on the server) and the data layer (in the DBMS). Moreover, it is an extensible architecture that can be further developed and enlarged in order to accommodate for a varying workload. For instance, the data layer can be replicated on multiple servers, in order to increase performance for accessing and manipulating data. This same strategy can be adopted for the business layer in case of necessity.

2.3.2 Detailed Deployment View

The following representation offers some more insights on how the software artifacts that implement the various components of the CodeKataBattle application map onto hardware elements.



The major elements in the representation are still the same as in the previous description. On the left, the client side, equipped with a device that has internet access in order to communicate with the server over a TCP/IP protocol. The multiplicity specifications on the line that connects the client with the server shows that many clients can be interacting with the server simultaneously.

The server is the big box at the center of this diagram. The UML deployment diagram offers the possibility to show the execution environment of the CodeKataBattle system by nesting multiple boxes.

All the microservices artifacts are mapped onto the single server instance. Indeed they are executed as separate processes inside the server machine.

Finally, on the right hand side, there is the data layer, which consists of a database server that is connected to the central server over the internet as well (TCP/IP protocol).

This design has been chosen because it allows for smooth and easy extension of the system whenever it is needed. For instance, the DBMS can be replicated multiple times, in order to parallelise the requests coming from the server. If a more advanced distributed architecture was needed, the artifacts implementing the various microservices could simply be split across different nodes of the network and the overall system would still work correctly (with some settings and configurations). The microservices themselves could be replicated inside the server machine in order to boost performance even more.

2.4 Run Time View

This section provides a detailed perspective on how the system internally works and its component interact with each other at runtime. It focuses on the dynamic aspects of the software, emphasizing the flow of control, data, and communication between different modules or components.
Just as a note, in order to make the diagrams more readable, some abbreviations have been used in the methods exposed by the various interfaces of CodeKataBattle components (as shown in section 2.5). In particular, tournamentName is tName, battleName is bName.

Educator login

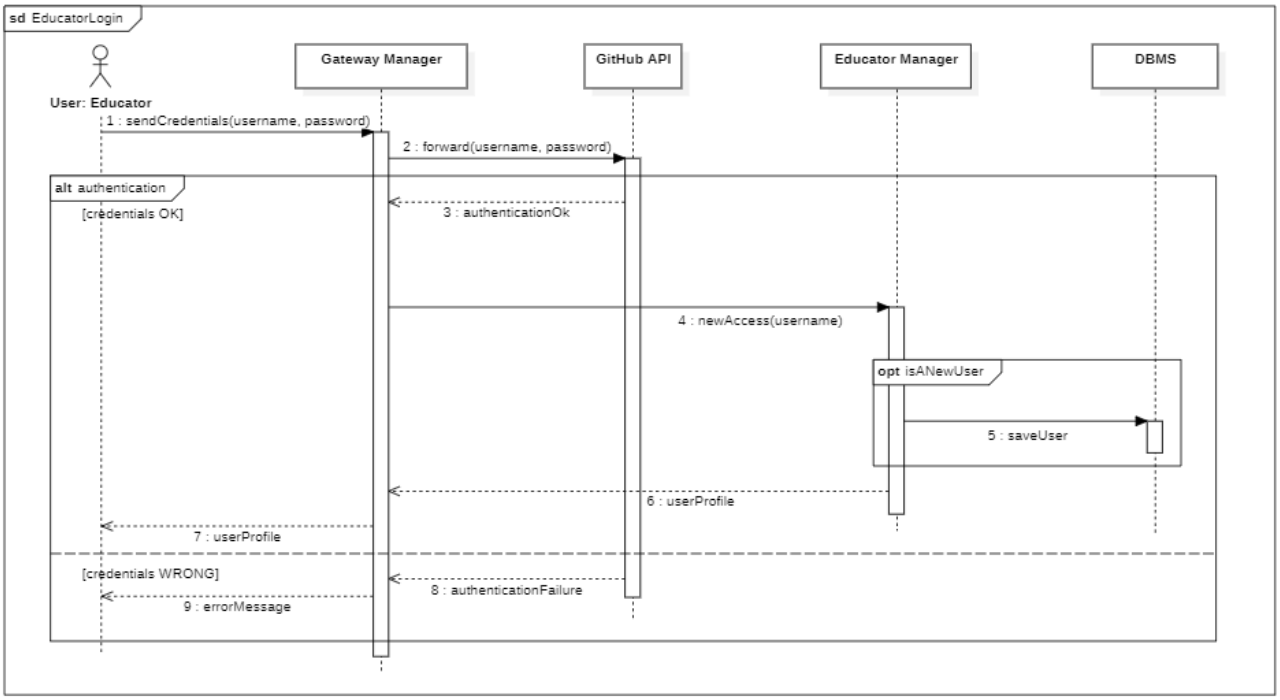


Figure 3: Runtime view of the login of an educator

The educator login process unfolds in the following manner: the Educator initiates the login by contacting the Gateway Manager and transmitting their credentials. Subsequently, the Gateway Manager interfaces with the GitHub API to verify and authorize access. Two possible scenarios may unfold: a successful authentication or an unsuccessful one. In the latter case, an error message is generated and displayed to the Educator. In the event of a successful authentication, the Gateway Manager communicates with the Educator Manager, forwarding all relevant data to be stored in case it is the Educator's first sign-in. In both scenarios, the Educator Manager is responsible for returning the user profile to the Educator.

This process ensures a seamless and secure login experience, with appropriate feedback provided to the Educator based on the outcome of the authentication process.

Student login

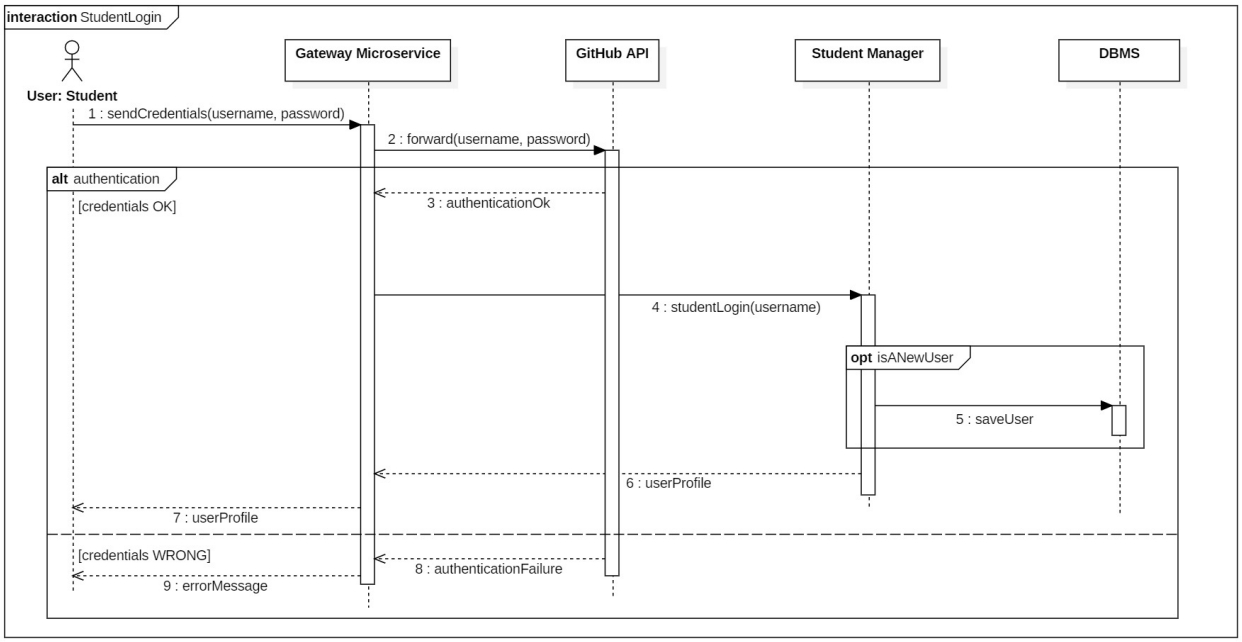


Figure 4: Runtime view of the login of a student

The student login process mirrors that of the educator, with the primary distinction lying in the actor, now the student, and the involvement of different microservices. Here's how it unfolds: the Student initiates the login process by reaching out to the Gateway Manager and submitting their credentials. Subsequently, the Gateway Manager interfaces with the GitHub API to validate and authorize access.

Similar to the educator's login process, two potential outcomes are possible: a successful authentication or an unsuccessful one. In the case of an unsuccessful authentication, an error message is generated and presented to the student. However, in the event of a successful authentication, the Gateway Manager now communicates directly with the Student Manager. The Student Manager is then responsible for handling the data, including storage in the case of the student's initial sign-in, and returning the user profile to the student.

Despite the differences in actors and microservices, the student login process ensures a comparable and secure user experience, tailored to the specific needs of the student user role.

Tournament creation

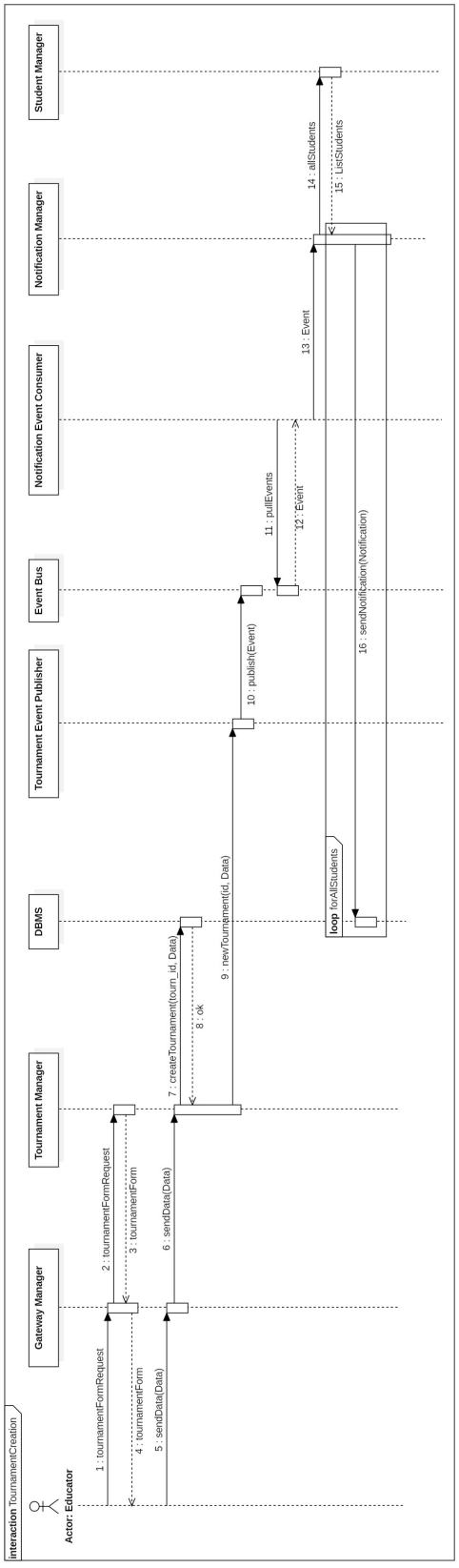


Figure 5: Runtime view of the creation of a tournament

In Figure 5, the diagram illustrates the interactions between components when an educator initiates the creation of a new tournament. Initially, the educator contacts the Gateway Manager, which then sends a request to the Tournament Manager in order to obtain the form necessary for the creation process. After receiving the form, the educator fills it out and sends it back to the Tournament Manager. The Tournament Manager stores the information in the DBMS and triggers a notification to signify the successful creation of the tournament.

While the specific details of this notification process will be discussed momentarily, it is worth noting that this sequence remains consistent across various runtime views and will be omitted in subsequent descriptions.

Subsequently, the Tournament Publisher receives the data pertaining to the newly created tournament and communicates with the Event Bus to publish this information. Then, asynchronously, the Notification Consumer polls the Event Bus, retrieving the new event. Once obtained, the Notification Consumer generates a notification and forwards it to the Student Manager. The Student Manager then iterates through all the students, ensuring each receives the notification.

Battle creation

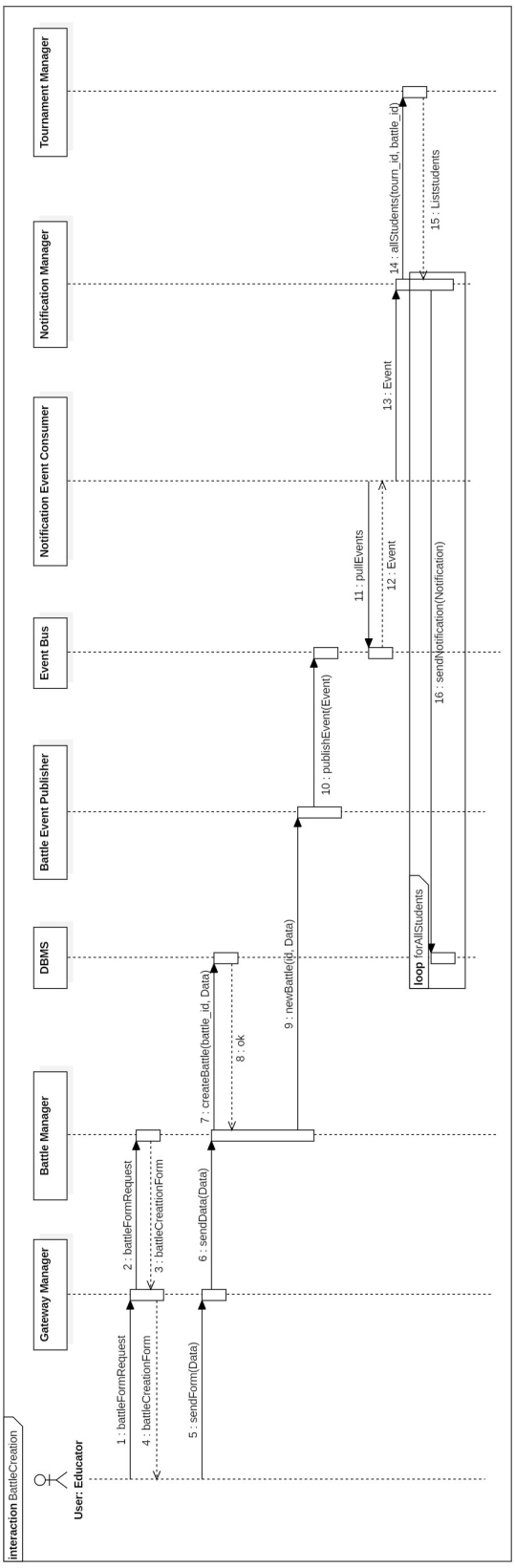


Figure 6: Runtime view of the creation of a battle

The process for battle creation mirrors that of tournaments. The Educator initiates the process by sending a request to the Gateway Manager, which forwards the request to the Battle Manager. The Battle Manager responds by providing the necessary form for the Educator to complete. Subsequently, the Educator fills out the form and returns it to the Battle Manager. The Battle Manager then stores the information for the newly created battle in the DBMS. Similar to the tournament creation, a new event is generated.

It's important to note that, in this context, all the students notified will only be those subscribed to the tournament in which the battle has been created.

End of a tournament

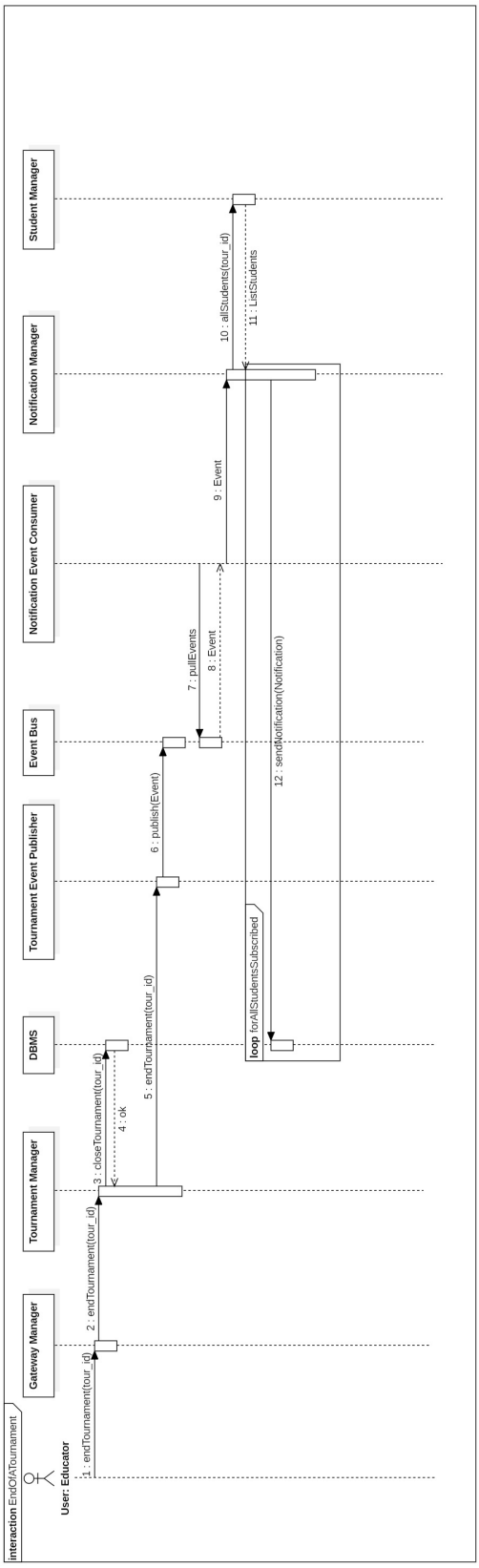


Figure 7: Runtime view when the end of a tournament is reached

When a tournament concludes, the determination is made by the Educator, who initiates the process by sending a request to the Gateway Manager. The Gateway Manager then forwards this request to the Tournament Manager, which in turn updates the relevant data regarding the tournament in the DBMS. Following the update, a new event is generated and disseminated to all the students who have subscribed to that particular tournament, adhering to the previously outlined notification mechanism.

End of a battle

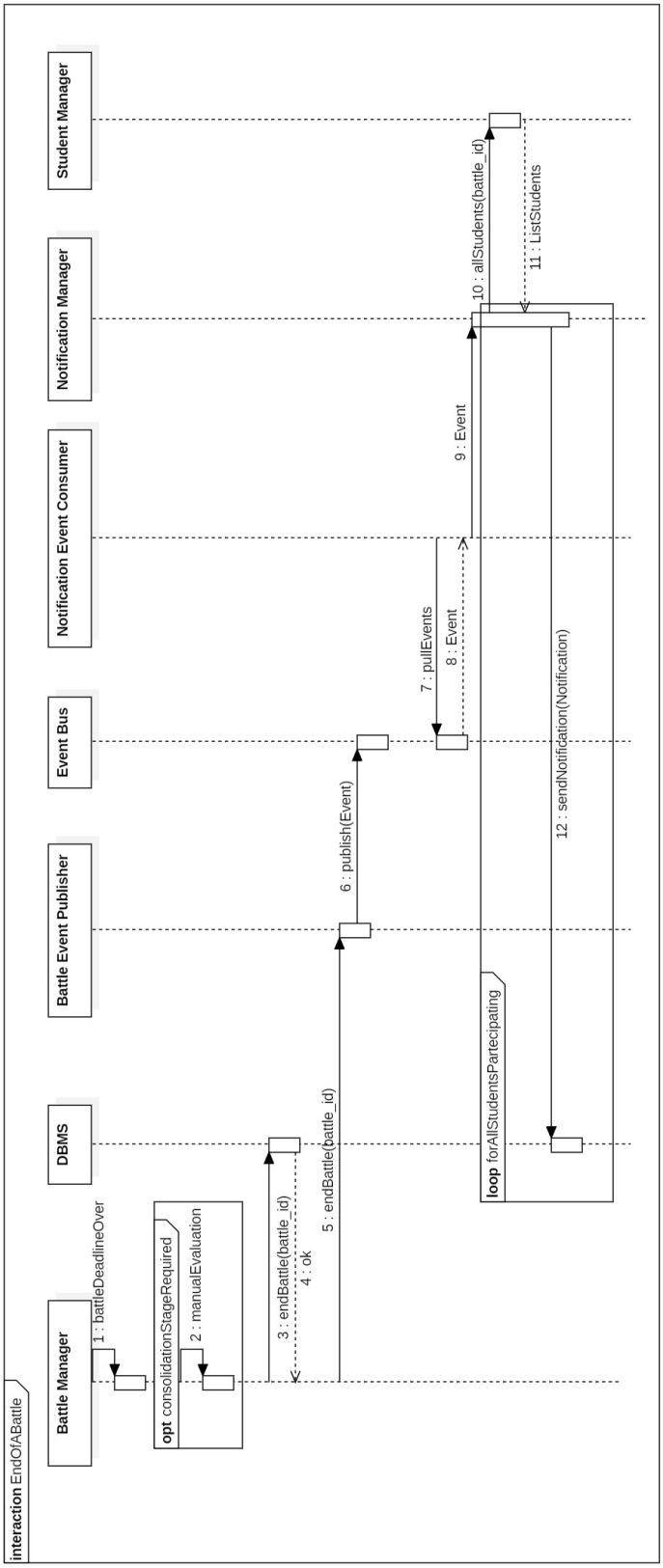


Figure 8: Runtime view when a battle ends

The conclusion of a battle occurs upon reaching its specified deadline, and the mechanism for monitoring this may involve a polling system, depending on the chosen development framework. Once the deadline is reached, the system checks whether a manual evaluation is required, as determined during the battle's creation. Subsequently, the parameters of the battle are updated by interacting with the DBMS. Following this update, an event is generated, mirroring the previously described process, and disseminated accordingly.

Subscribe To Tournament

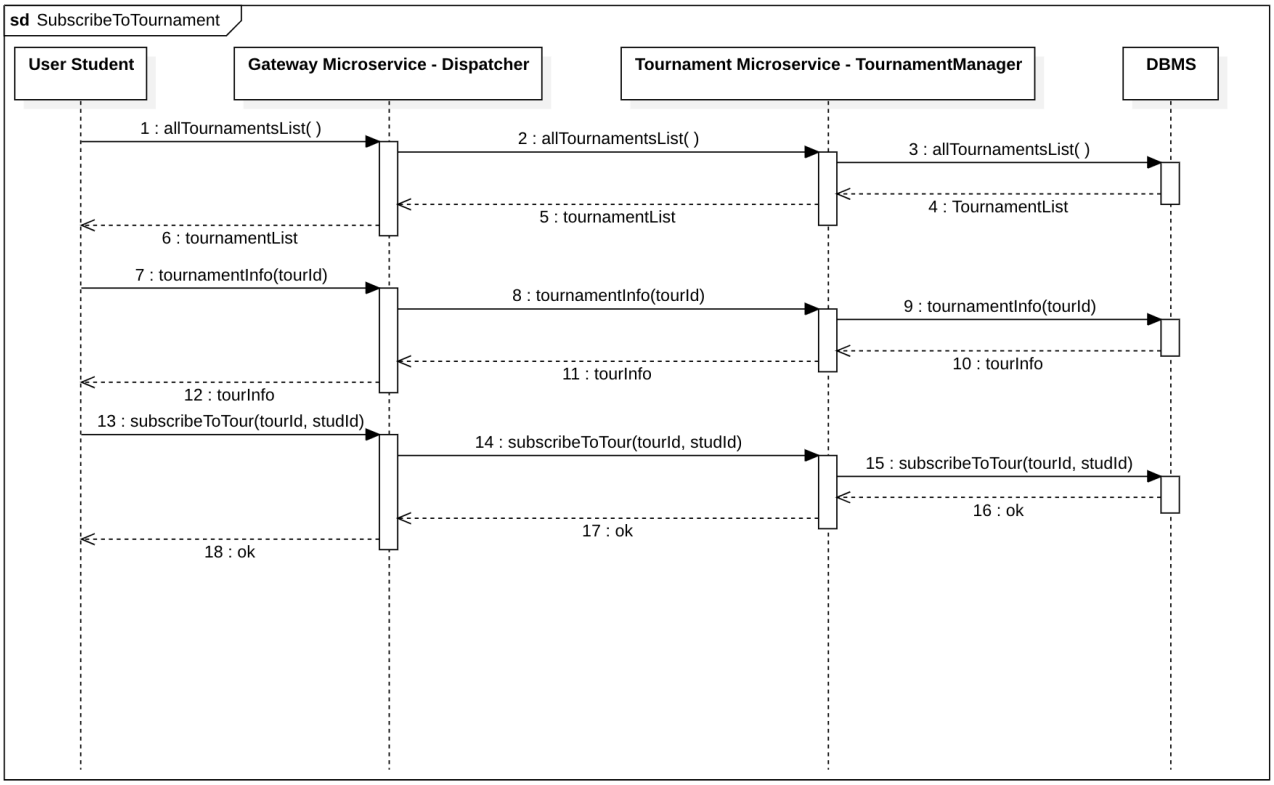


Figure 9: Runtime view when a student subscribes to a tournament

The client initiates a request for a list of tournaments. This request is transmitted to the gateway component, which acts as an intermediary. The gateway, upon receiving the request, forwards it to the Tournament Manager microservice. The Tournament Manager retrieves the relevant information from the database and subsequently transmits the data back to the gateway. The gateway, in turn, relays the tournament list to the client.
Subsequently, the client initiates a request for specific information about a particular tournament. This request traverses the gateway before reaching the Tournament Manager microservice. The Tournament Manager retrieves the requisite details from the Database Management System (DBMS) and transmits the information back.
Lastly, the client issues a subscription request. This request is routed through the gateway to the Tournament Manager. The Tournament Manager processes the request by writing the new subscription details to the DBMS, thereby completing the subscription process. After the processs is completed a confirmation message is trasmitted back.

Subscribe To Battle

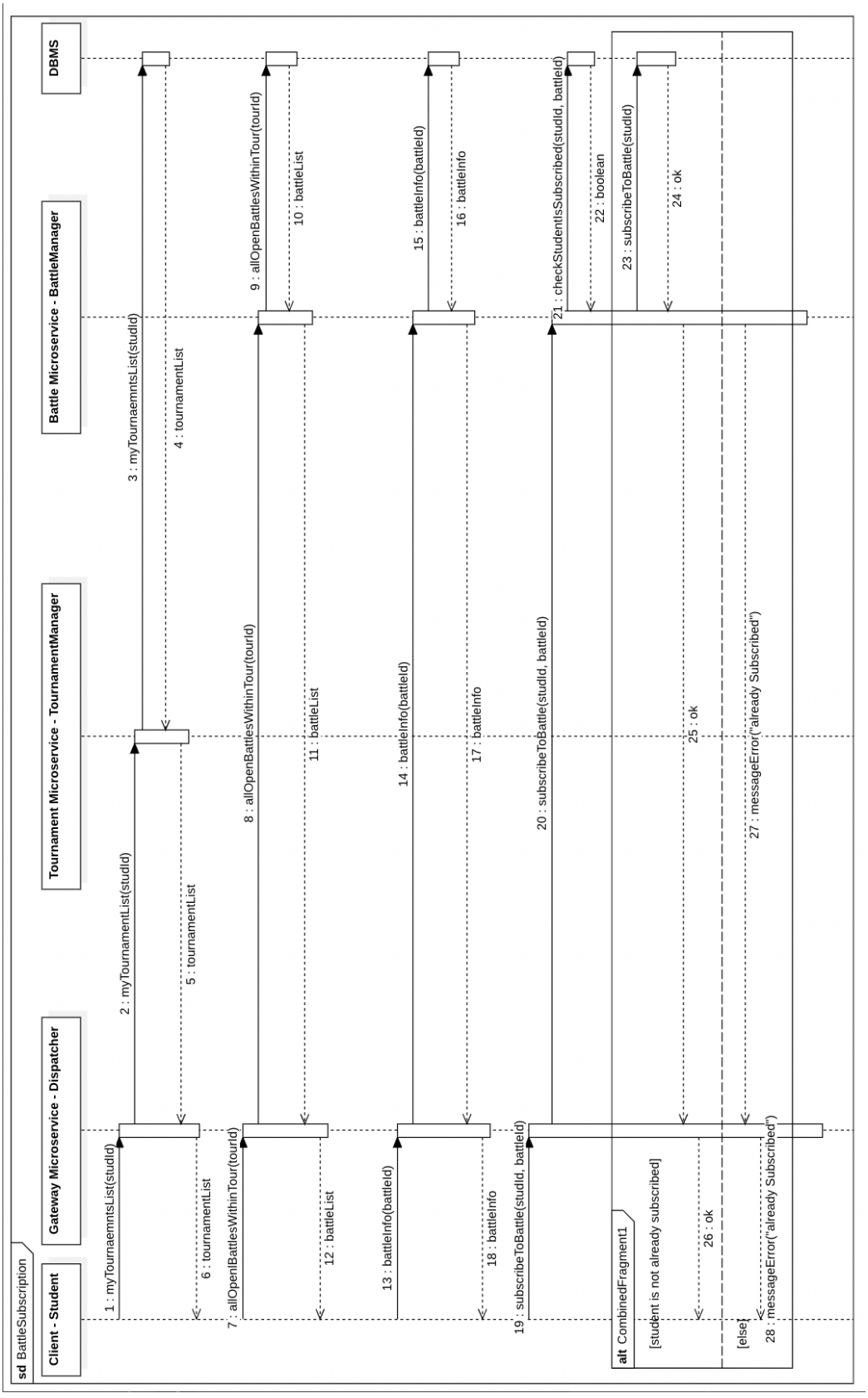


Figure 10: Runtime view when a student subscribes to a battle

The sequence begins with the client initiating a request for the list of tournaments. This request is directed to the gateway component. The gateway, acting as an intermediary, forwards the tournament list request to the Tournament Manager microservice. The Tournament Manager retrieves the pertinent information from the Database Management System (DBMS) and transmits it back to the gateway. Subsequently, the gateway relays the tournament list to the client.
Following this, the client makes a request for the list of battles within a specific tournament, specifically those currently open for subscription. This request is initially routed through the gateway before being further directed to the Battle Manager microservice. The Battle Manager, upon receiving the request, retrieves the relevant information from the database and sends the list of open battles back through the gateway to the client.
Subsequently, the client sends a subscription request for a specific battle. The subscription request first traverses the gateway and then reaches the Battle Microservice. Within the Battle Microservice, the system checks the database to verify if the student is already subscribed to the specified battle. If the student is already subscribed, an error message is generated and transmitted back to the client. If the student is not already subscribed, the Battle Manager writes the new subscription details to the database and generates a confirmation of the subscription. The confirmation of the subscription is then sent back through the gateway to the client.

2.5 Component Interfaces

This section illustrates a comprehensive list of interfaces offered by the various components of the CodeKataBattle system, along with the specifications for their internal methods. A description is provided for each method in order to understand what it is useful for.

The following tables are divided by microservice.

1 - Gateway Microservice

DiscoveryServiceI

Return Type	Signature	Description
URL	findService(String serviceName)	This method is offered to allow all microservices to locate each other and therefore collaborate.
void	registerService(String serviceName, URL location)	This method allows a newly-created service to register itself on the list of available services, in order to be located by the other microservices if needed.
List<URL>	allServices()	This method returns a list of all the available services at the moment in the system.

2 - User Management Microservice

The following interfaces are RESTful APIs offered by the Manager components that handle Students and Educators' data.

StudentAPI

Return Type	Signature	Description
List<Student>	getAllStudents()	This method is offered to retrieve a list of all the students with an account on CodeKataBattle.
void	newAccess(username)	Checks whether the user accessing the system is a new student and if that's the case, a new student profile is added to the DBMS.
void	removeStudent(username)	Removes the student profile with the specified username from the system.
List<String>	getStudentInfo(username)	Retrieves all the information related to a specific student, like list of badges, statistics...
List<String>	getBadges(username)	Returns the list of badges won by the student with the specified username.
List<File>	getStudentProfileUI(username)	This method builds and returns the files that are needed to offer the user interface for a student's profile.

EducatorAPI

Return Type	**Signature**	**Description**
List<Educator>	getAllEducators()	This method is offered to retrieve a list of all the educators with an account on CodeKataBattle.
void	newAccess(username)	Checks whether the user accessing the system is a new educator and if that's the case, a new educator profile is added to the DBMS.
void	removeEducator(username)	Removes the educator profile with the specified username from the system.
List<File>	getEducatorProfileUI(username)	This method builds and returns the files that are needed to offer the user interface for an educator's profile.

3 - Tournament Microservice

The following interface is a RESTful APIs offered by the Manager component that handles Tournaments' data.

TournamentAPI

Return Type	Signature	Description
void	addTournament(tournamentName, tournamentInfo)	Adds a new tournament to the ones available on the CodeKataBattle platform. Notice that tournamentInfo is a complex entity that comprises a set of mandatory elements for the creation of a new tournament. Tournament names are the identifiers for battles and have to be unique in CodeKataBattle .
void	closeTournament (tournamentName)	Closes the tournament with the tournamentName specified. The tournament ranking is published and badges assigned.
List<Tournament>	getAllTournaments()	Retrieves a list of all the tournaments available on the CodeKataBattle system in the moment the method is called.
void	grantPermissions (tournamentName, username)	This method allows to grant permissions to publish battles to an educator with the specified username in the tournament identified by the tournamentId.

TournamentAPI

Return Type	Signature	Description
void	addStudent(tournamentName, username)	Adds the student with the specified username to the list of students that are subscribed to the tournament.
List<Student>	getStudents(tournamentName)	Retrieves a list of all the students that are subscribed to the tournament with the tournamentName specified.
List<Educator>	getEducators(tournamentName)	Retrieves a list of all the educators that have permissions to publish battles inside the tournament with the tournamentName specified.
String	getCreator(tournamentName)	Returns the username of the Educator that created the tournament with the tournamentName specified.
List<Tournament>	getAllTournaments()	Returns a list of all the tournaments available on CodeKataBattle.
List<Tournament>	getTournaments(username)	Returns a list of all the tournaments the student with the specified username is subscribed to.
List<Badge>	getBadges(tournamentName)	Retrieves a list of all the badges that have been defined for the tournament with the tournamentName specified.

TournamentAPI

Return Type	Signature	Description
List<Float>	getRanking(tournamentName)	Retrieves the ranking of all students subscribed to a specific tournament with the tournamentName specified.
void	updateRanking(tournamentName)	Requires to update the ranking of a tournament with new data that is now available.
List<Battle>	getBattles(tournamentName)	Retrieves a list of all the battles that have been published in the tournament with the tournamentName specified.
List<File>	getTournamentCreationForm()	Returns all the necessary material to display to an educator the user interface to insert data for a new tournament to be created.
List<File>	getTournamentHomePage(tournamentName)	Returns all the necessary material to display to an user of CodeKataBattle the home page of a tournament, with all the information related to it.

Tournament - EventPublisherI

Return Type	Signature	Description
void	publishEvent(Event)	Publishes an event on the event bus.

Tournament - EventConsumerI

Return Type	Signature	Description
Event	consumeEvent()	Reads an event from the event bus.
List<Event>	consumeAllEvents()	Reads all the events published on the event bus.

4 - Battle Microservice

The following interface is a RESTful APIs offered by the Manager component that handles Battles' data.

BattleAPI

Return Type	Signature	Description
void	addBattle(battleName, tournamentName, battleInfo)	Adds a new battle to the tournament specified with tournamentName. Notice that battleInfo is a complex entity that comprises a set of mandatory elements for the creation of a new battle. Battle names are the identifiers for battles and have to be unique in CodeKataBattle .
void	terminateBattle(battleName)	Terminates the battle named battleName; the final ranking is drawn and published.
void	addTeam(battleName, teamName, List<Student>)	Adds a new team to the battle called battleName, composed of the students specified in the second parameter of the function. In case of a single student, the teamName is the username of the solo player by default.
String	getCreator(battleName)	Returns the username of the educator that created the battle named battleName.
List<Team>	getAllTeams(battleName)	Retrieves a list of all the teams that participate in the battle named battleName.

BattleAPI

Return Type	Signature	Description
List<Student>	getStudents(battleName)	Retrieves a list of all the students participating in the battle named battleName.
List<Student>	getStudentsInTeam(battleName, teamName)	Retrieves a list of usernames of the students that are part of a specific team named teamName inside the battle battleName.
List<Float>	getRanking(battleName)	Retrieves the ranking of the battle named battleName.
void	updateScore(battleName, teamName, newScore)	Updates the maximum score assigned to a specific team and the ranking is changed accordingly.
Float	getTeamScore(battleName, teamName)	Retrieves the maximum score assigned to a team in a battle so far.
Date	getCreationTime(battleName)	Returns the exact date and time when the battle called battleName was created (useful for computing scores).
List<File>	getBattleCreationForm()	Returns all the necessary material to display to an educator the user interface to insert data for a new battle to be created.
List<File>	getBattleHomePage(battleName)	Returns all the necessary material to display to an user the home page of the battle named battleName.

Battle - EventPublisherI

Return Type	Signature	Description
void	publishEvent(Event)	Publishes an event on the event bus.

5 - Notification Microservice

The following interface is a RESTful APIs offered by the Manager component that handles Notification data.

NotificationAPI

Return Type	Signature	Description
List<Notification>	getNotifications(username)	Retrieves all the notifications that are stored for the user with the specified username.
void	newTournament(tournamentName)	This function is called every time a new tournament is created and all the notifications for all the students subscribed to CodeKata-Battle have to be fabricated.
void	endedTournament(tournamentName)	This function is called every time a tournament is closed and all the notifications for all the students subscribed to the tournament have to be fabricated.
void	newBattle(battleName)	This function is called every time a new tournament is created and all the notifications for all the students subscribed to the tournament in which the battle resides have to be fabricated.
void	endedBattle(battleName)	This function is called every time a battle terminates and all the notifications for all the students subscribed to the battle have to be fabricated.

NotificationAPI

Return Type	Signature	Description
void	notifyTeamOffer(offererUser, listTeammates, bName)	This function is leveraged in order to allow the creation of teams for joining battles. When a student wants to join a battle with other students as a team, s/he selects the students s/he wants to invite and they have to receive a personal notification.
void	permissionsGranted(grantingEd, allowedEd, tournamentName)	This function is called every time a notification has to be created for an educator that received permissions from another educator to publish battles in a tournament (tournamentName).
List<File>	getNotificationUI()	Returns all the necessary material to display to a student the user interface to see the notifications that have arrived to him/her on CodeKataBattle.

Notification - EventConsumerI

Return Type	Signature	Description
Event	consumeEvent()	Reads an event from the event bus.
List<Event>	consumeAllEvents()	Reads all the events published on the event bus.

6 - GitHub Integration Microservice

The following interface is a RESTful APIs offered by the Manager component that handles GitHub integration with CodeKataBattle.

GitHubIntegrationAPI

Return Type	Signature	Description
URL	getRepositoryLink(battleName)	Retrieves the link to the remote GitHub repository created by CodeKataBattle for the battle named battleName.
URL	createRepository(battleName)	Creates a new GitHub repository for the battle named battleName and returns its link.
void	removeRepository(battleName)	Deletes the GitHub repository for the battle named battleName, when the battle is closed.
void	newCommit(battleName, username)	This method is offered to receive notifications from GitHub when a new commit is performed by a student in a repository that stores solutions for a battle named battleName.
List<File>	pullSources(battleName, username)	Downloads the latest source code files for the user specified by username in the context of the battle named battleName.

GitHub Integration - EventConsumerI

Return Type	Signature	Description
Event	consumeEvent()	Reads an event from the event bus.
List<Event>	consumeAllEvents()	Reads all the events published on the event bus.

7 - Score Computation Microservice

CalculatorAPI

Return Type	Signature	Description
Float	calculateScore(sourceCode, buildScripts, testCases, timePassed, evaluationCriteria)	This method implements the main functionality of the microservice, which is computing the score of a specific solution submitted by a student and return it.

2.6 Selected Architectural Styles and Patterns

This section delves into our architectural decisions, explaining the reasoning behind each choice. We aim for a system that not only meets current demands but also allows for flexibility and evolution. A comprehensive view empowers both developers and stakeholders with a clear understanding of how these choices contribute to the overall success of the software engineering work.

Microservices Architecture

The adoption of a microservices architecture is driven by several factors, contributing to the versatility and adaptability of our application:

- *Scalability*: Microservices allow independent scaling, enabling the scalability of sub-services without scaling out the entire system, fostering a versatile and responsive application.
- *Fault Tolerance*: Unlike monolithic approaches with inter-dependencies, microservices minimize the impact of a single module failure, ensuring the application remains largely unaffected.
- *Deployment and Productivity*: Microservices facilitate continuous integration and delivery. The modular structure enhances team productivity and team work, with each member responsible for a specific service.
- *Maintainability*: Microservices reduce the effort spent on understanding separate pieces of software, enhancing overall system maintainability.

RESTful APIs

The adoption of the REST architectural style for designing and implementing our APIs offers a scalable, efficient, and interoperable communication model. This choice is motivated by several relevant advantages:

- *Simplicity and Intuitiveness*: RESTful APIs follow a straightforward design using standard HTTP methods.
- *Scalability*: The stateless nature of REST allows for easy scalability, promoting a loosely coupled and scalable architecture.
- *Flexibility and Modularity*: RESTful APIs encourage a modular and resource-centric design, promoting clean separation of concerns. Each microservice in the architecture focuses on a specific section of the data domain and leaves the rest to the other components.
- *Interoperability*: Leveraging standard HTTP protocols makes REST platform-agnostic, facilitating interoperability across different technologies and devices.
- *Statelessness*: RESTful APIs' statelessness simplifies server logic, improves reliability, and allows for easy recovery from failures, contributing to a more robust and fault-tolerant system.

Model-View-Controller (MVC)

The adoption of the Model-View-Controller architecture lays the foundation for a well-organized, maintainable, and scalable application. This choice is motivated by several factors:

- *Modular Organization*: MVC divides the Manager components inside the various microservices into three interconnected components: Model, View, and Controller. This enhances code readability, facilitates easier debugging, and allows for independent development of each component.
- *Separation of Concerns*: MVC enforces a clear separation of concerns, simplifying development, making it easier to maintain, extend, and modify individual components without affecting the others.
- *Reusability and Extensibility*: The modular structure of MVC promotes code reusability, allowing components to be reused in different parts of the application or in future projects.
- *Testability*: MVC enhances the testability of our application, allowing for independent testing of each component, ensuring that each part of the application functions correctly.
- *Improved Collaboration*: MVC's division of responsibilities facilitates collaboration among teammates operating on this project, fostering efficiency.

Event-Driven Architecture (EDA)

Beside the fact that the adoption of Event-Driven Architecture (EDA) enhances real-time responsiveness, and scalable communication, its use in the CodeKataBattle system is specifically driven by the desire to generate asynchronous notifications. Through the Notification Consumer, Tournament Publisher, and Battle Publisher, we effectively mitigate the overhead of requesting additional computation for each action involving Notification, Tournament, and Battle Managers. In this way, the Battle and Tournament Publishers are able to create new events when a battle or a tournament begins or ends, and the Notification Consumer can asynchronously read these events and fabricate all the necessary notifications for the users of CodeKataBattle. All of this, without requiring neither the Battle, nor the Tournament microservices to stop their ongoing tasks. Other advantages of the architecture are:

- *Decoupled Components*: EDA promotes decoupling of components by relying on events as the means of communication. This results in a more flexible architecture, allowing for independent development, testing, and deployment.
- *Scalability*: EDA facilitates scalability by enabling the distribution of components and services. This asynchronous and distributed nature supports the scalability of our application, allowing it to handle increased loads and adapt to varying workloads.

2.6.1 Database Management

The data layer of the CodeKataBattle application is composed of a shared database, which provides persistent data access to all the microservices. This choice derives from the fact that although the different microservices making up the CodeKataBattle system have different concerns, they still operate on pieces of information that are frequently related (tournaments and battles, users and badges, battles

and GitHub repositories...). A shared and common place to store the data of the application facilitates the accomplishment of data consistency and data sharing between different processes (microservices).

More specifically, managing the shared database involves the design of internal partitions to divide data among various microservices. Each partition is replicated to guarantee reliability and availability. Protocols such as multi-Paxos or Raft, which manage replicated state machines, can be considered as implementation choices; 2PC could be used across partitions, assuming the reachability of the majority of nodes, to ensure the atomicity and consistency of transactions.

3 User Interface Design

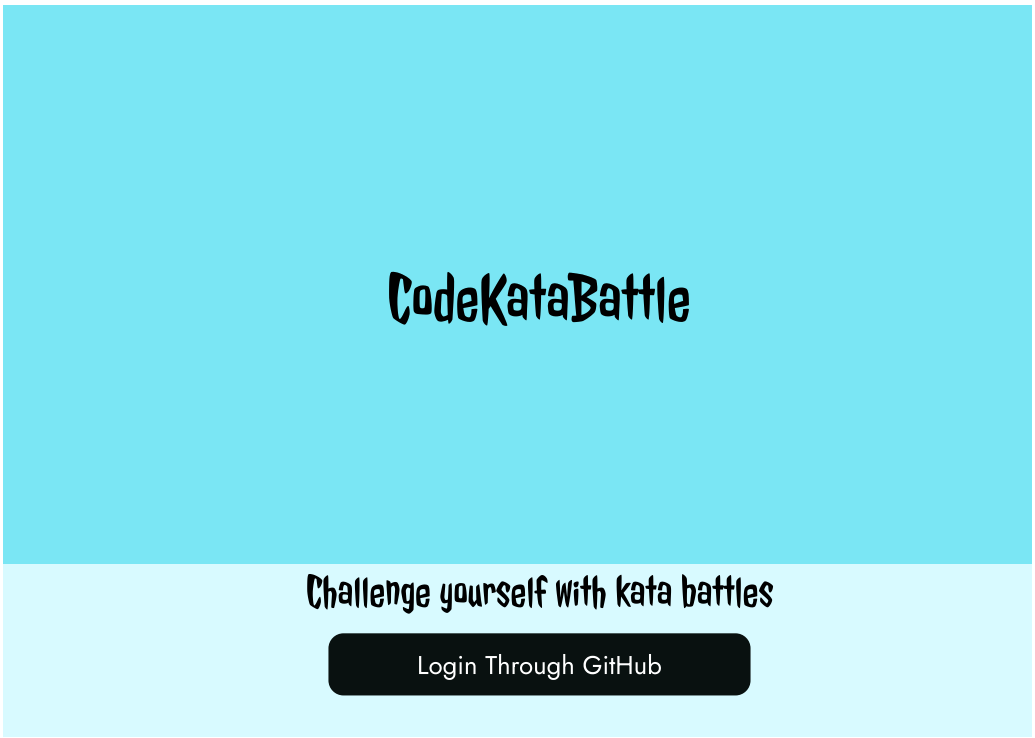
This section is a refinement of the chapter ”3.1.1 User Interface” of the Requirements Analysis and Specification Document (RASD) related to the CodeKataBattle project. At the level of detail that is allowed for a design document, it is possible to delve deeper into the various interactions that the platform offers to the external user (either a student or an educator).

The images inserted in this section serve as mockups of the user interfaces provided by the system and therefore they have to be interpreted as general guidelines for the designers and developers to grasp the application should look like at the end of the implementation process.

In order to facilitate the understanding of the reader, the following description is going to take first the point of view of an Educator using CodeKataBattle and then there will be a switch to the student perspective.

3.1 Login Phase

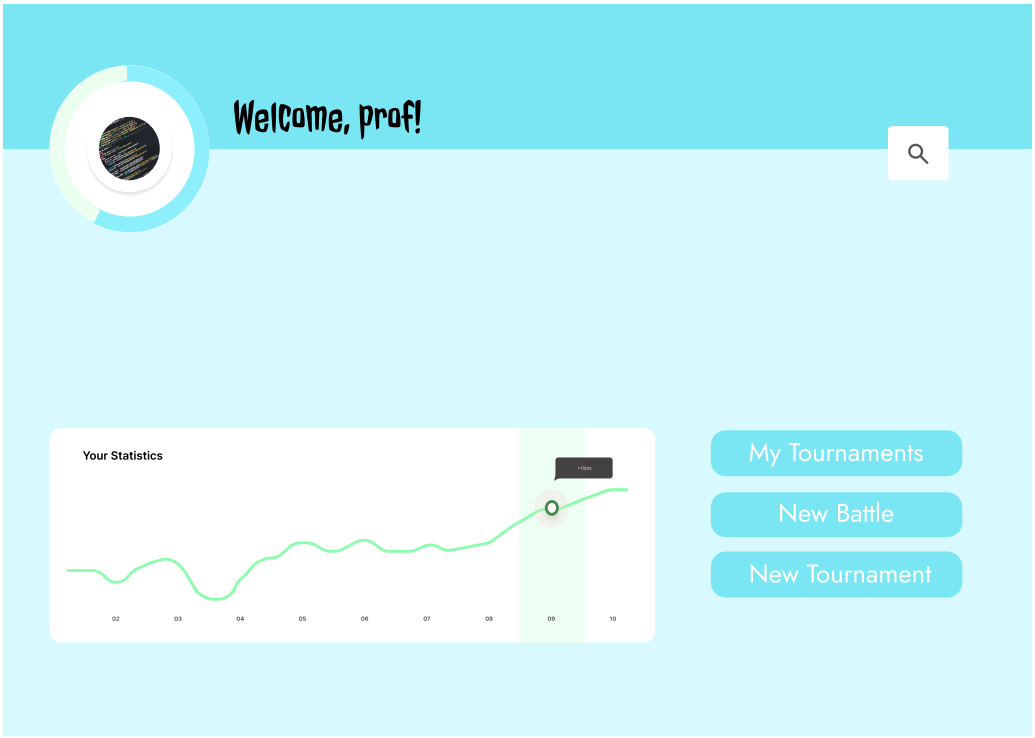
The initial step for accessing CodeKataBattle is of course logging in the system, and this is possible thanks to the initial user interface provided by the application. It is illustrated in the following image:



By clicking on the button ”Login Through Github” the user will be redirected to a web page where s/he will be able to insert his/her personal GitHub credentials.

3.2 Educator personal profile

As stated in the introduction, the description now takes the point of view of an educator. The first page displayed after the login is the personal profile of the educator, as it is possible to observe in the following illustration:



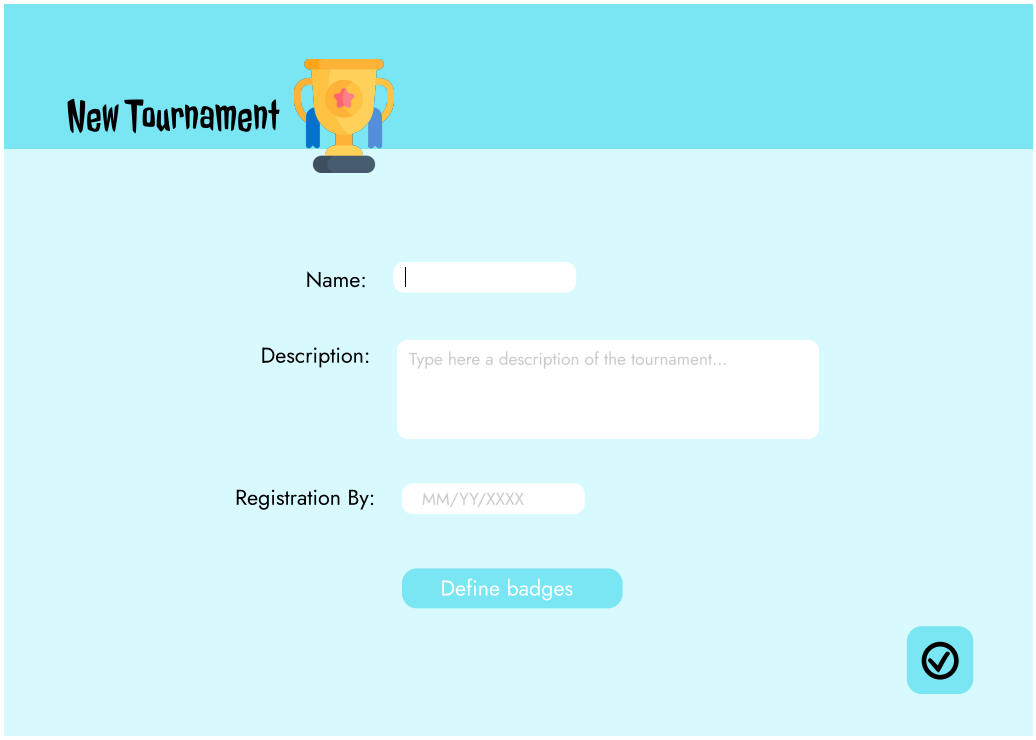
This page offers various functionalities:

- My Tournaments button allows an educator to see the list of tournaments that s/he created or the ones s/he has permissions to publish battles in. The Tournament microservice is responsible for providing this list.
- New Battle button allows for the creation of a new battle. When the educator clicks on it, the Tournament microservice is involved because a list of tournaments in which the educator is allowed to publish battles has to be supplied. The educator is then able to select a tournament out of that list (in which s/he wants to publish the battle) and automatically the page for creating the new battle pops up (as described later on).
- New Tournament button allows for the creation of a new tournament. The request is passed to the Tournament Microservice which will reply with the tournament creation form (as shown later on).

Let's assume now the educator clicks on the "New Tournament" button. The following section shows the interface that pops up in this case.

3.3 Creation of a new tournament

In order to create a new tournament, the tournament creation form has to be filled in by the educator. A sketch of how this form would look like is reported here:



The most important pieces of information for a tournament are clearly visible in the image, as the educator is able to set the tournament name, the tournament description and the registration deadline. There is also a dedicated section for the badges, that will be assigned at the end of the tournament.

Once the educator fills in all the form, s/he can confirm with the button in the bottom right side of the interface. As s/he does it, the tournament home page appears on the screen, as illustrated in the next section.

3.4 Tournament home page

Whenever a student or an educator searches for a tournament and clicks on it, the tournament home page pops up. This is how the page is designed in order to provide all the relevant information related to a tournament:

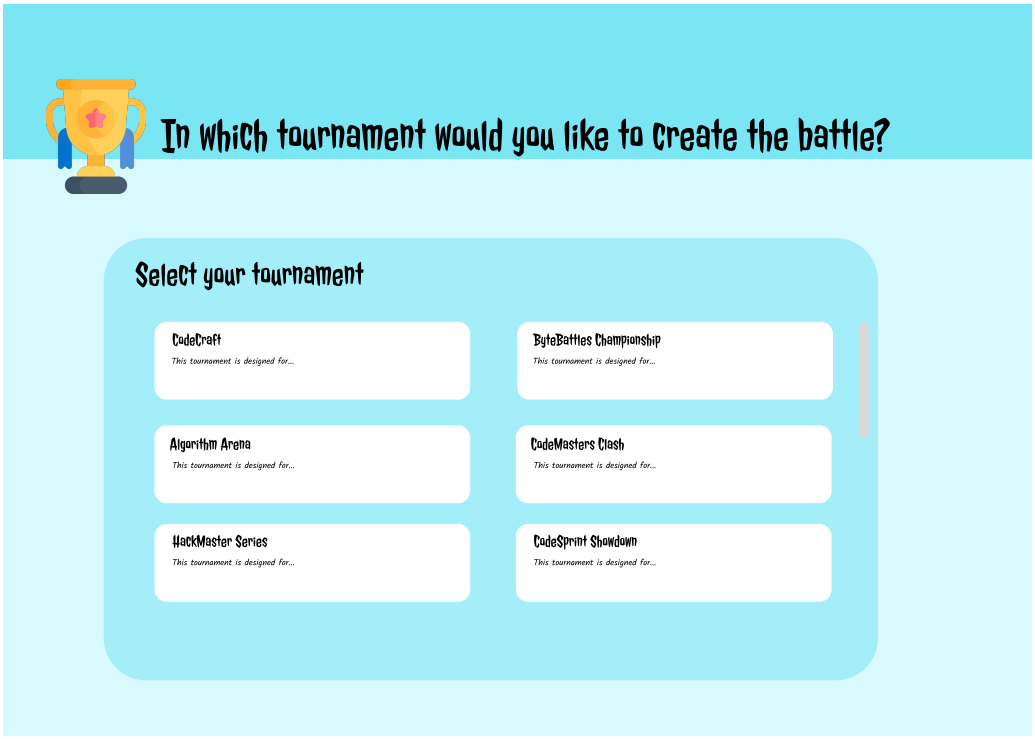
It is important to mention the fact that this interface is flexible based on the type of user that is accessing the tournament. The view provided here is the most general one. In case an educator is accessing the tournament, the ”Join Tournament” button wouldn't be there, as well as the ”My Battles” section. In the same way, if a student already joined a tournament and wants to open the tournament home page, the ”Join Tournament” button won't be present on the screen. These are details that will be handled by the components of CodeKataBattle in charge of building the correct user interface.

As a general description of this interface, it is possible to say that in the top right corner of the page, there is a button that allows a student to join the tournament. As the student clicks on it, the request is dispatched to the Tournament Microservice that will take care of verifying if the registration deadline for the tournament hasn't passed yet and consequently add the student to the set of students subscribed to the tournament.

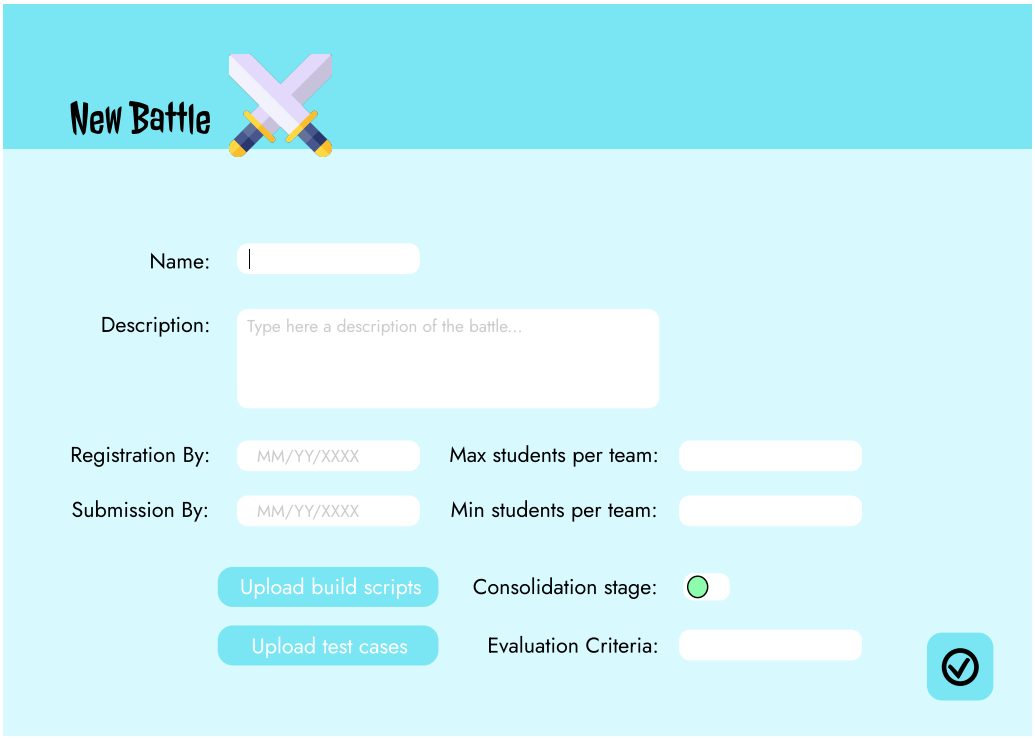
On the left, a list of the battles published within the tournament is available for students to join them and for educators to see them. On the right hand side, two panels appear. Going top to bottom, the first one is a list of battles that the student joined, the second one is the tournament ranking, which displays all the students participating in the tournament as an ordered list of names.

3.5 Creation of a new battle

Let's now go back to the educator's home page (educator profile). Let's suppose the educator clicks on the "New Battle" button. The first response of CodeKataBattle will be showing a list of the tournaments in which the educator has permissions to publish battles. The following image shows this interface:



As the educator selects the tournament, a new page shows up, displaying the battle creation form.



All the pieces of information that are needed for the creation of a new battle have their respective

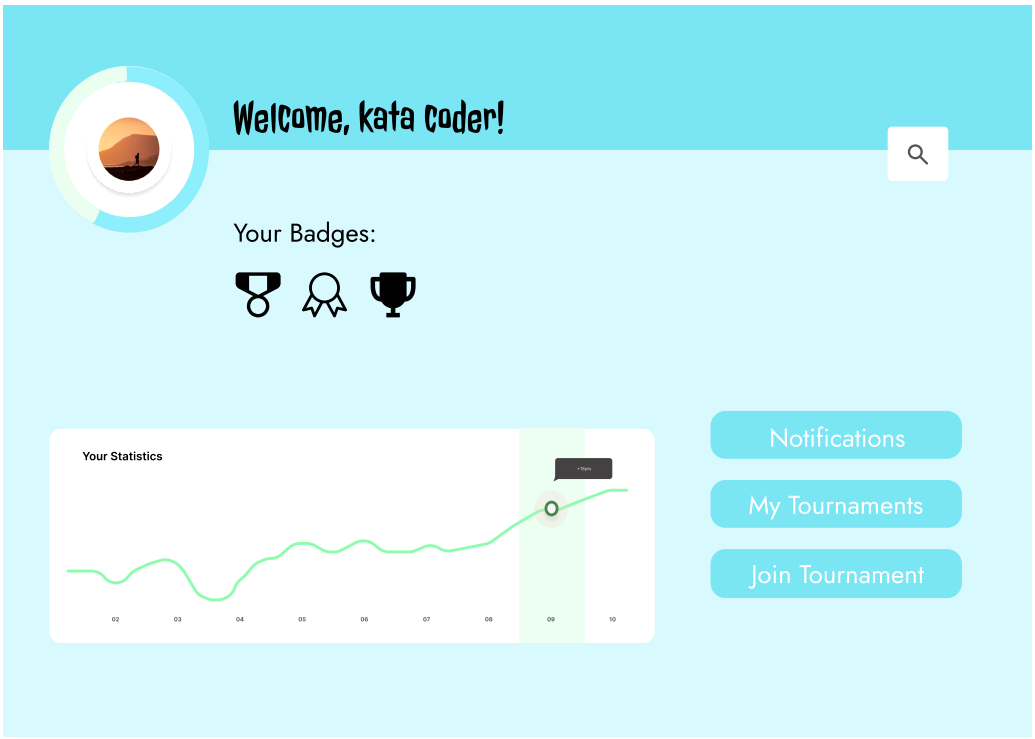
fields in this user interface. Once all these parts have been correctly defined, the educator can confirm his/her choices with the button in the bottom right corner of the interface. As the confirmation button is pressed, the Battle Microservice will be immediately provided with all the data that the educator provided through this interface, in order to allow the Battle Manager to create a new battle and store this data in a persistent manner.

It is now possible to switch to the student's point of view to illustrate the user interfaces that the CodeKataBattle system provides.

First off, the interface for signing in CodeKataBattle is the same, as both educators and students are required to log in through their GitHub account.

3.6 Student personal profile

As the login phase is successfully concluded CodeKataBattle displays the student's personal profile. The following figure sketches the look of this user interface:

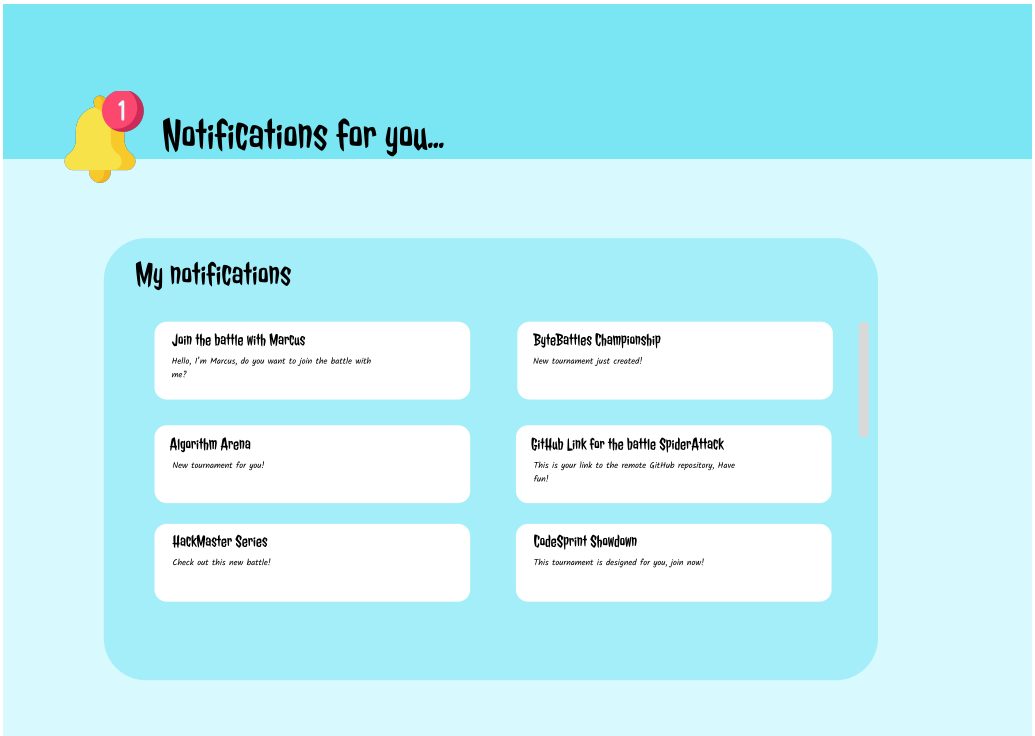


In the central part, a list of the badges earned by the student is shown, and it is visible to all the users of CodeKataBattle navigating to this student's profile. The buttons on the right hand side offer the major functionalities that the student requires in order to use the application.

When the student clicks on the "Notifications" button, CodeKataBattle responds with a new interface that shows all the notifications that arrived for the student in question. The Notification Microservice is responsible in this case for retrieving the list of notifications related to a specific student and building the corresponding interface, which is shown in the following section.

3.7 Notification interface

Here's a mockup of the Notification interface for a student:



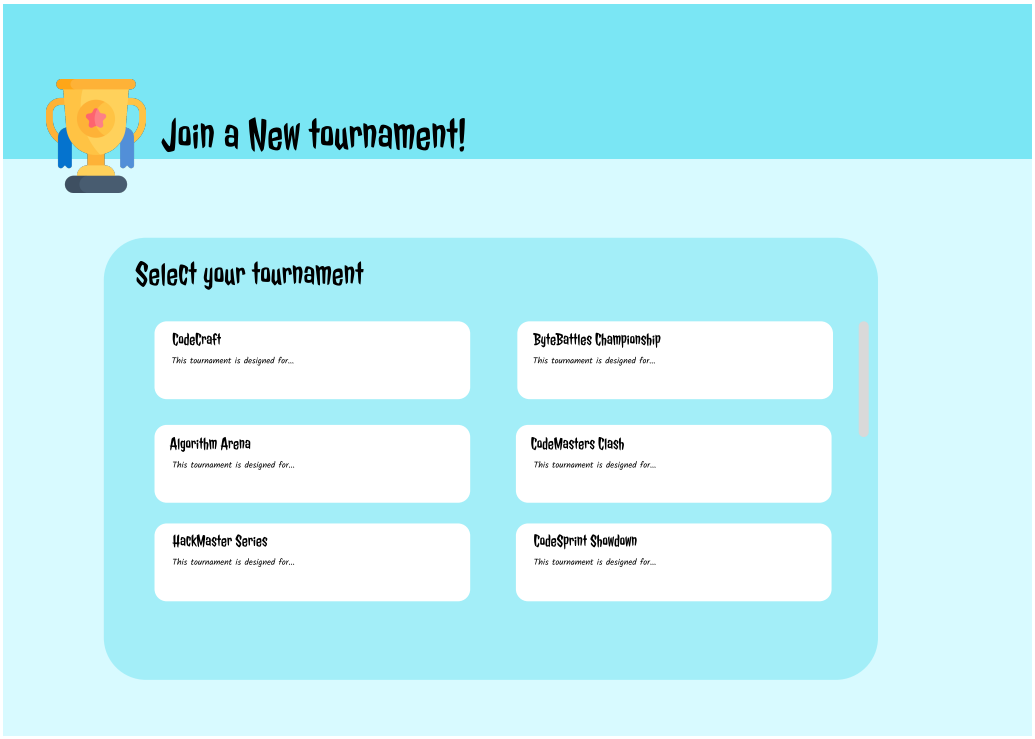
The illustration is pretty straightforward as it is composed of a single central pane that gathers all the notifications for the student. Among the various notifications that can reach a student there are:

- Notification for the creation of a new tournament.
- Notification for the request of another student to join a battle together as a team.
- Notification for the creation of a new battle.
- Notification that carries the link to the remote GitHub repository for a battle the student is subscribed to.
- Notification for the termination of a tournament.
- Notification for the termination of a battle.

Going back to the student's personal profile, let's see what happens when the "Join Tournament" button is clicked.

3.8 Joining a tournament

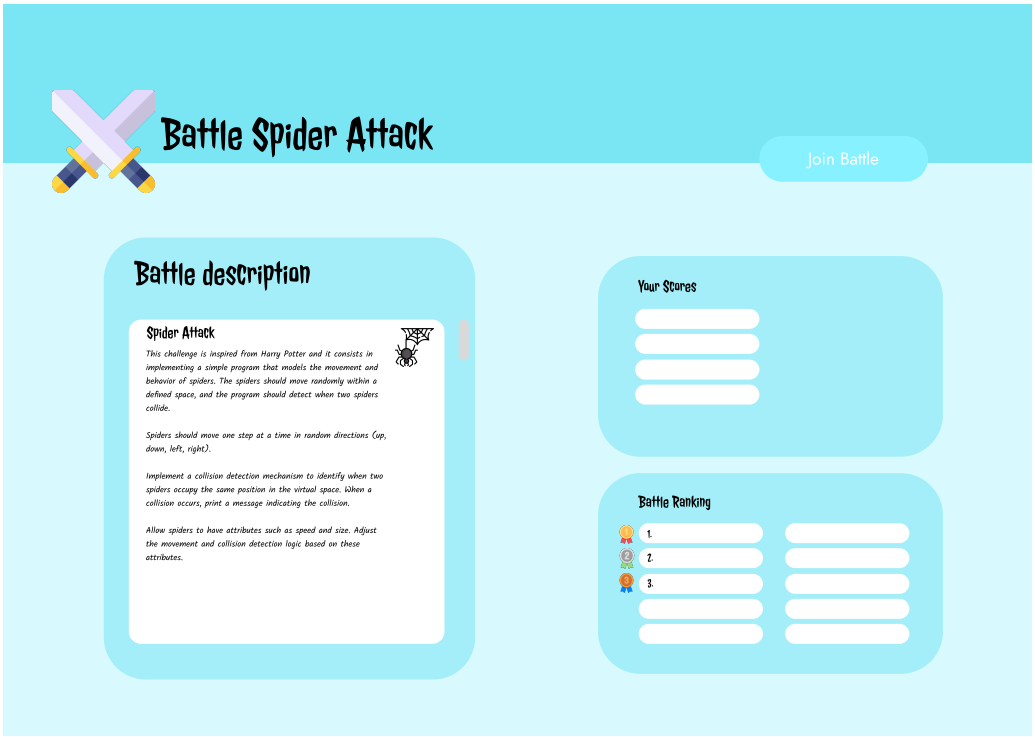
As the ”Join Tournament” button is clicked, CodeKataBattle provides to the student a list of tournaments that are available in that moment inside the platform. The Tournament Microservice is responsible for providing such list of tournaments and building the relative user interface.
This is the look of the interface for joining a new tournament:



The student can browse all the tournaments available on CodeKataBattle from here. When the student clicks on one of the tournaments, the tournament home page shows up (as illustrated in section 3.4) and the button ”Join Tournament” will be visible if the registration deadline for the tournament hasn’t passed yet.

3.9 Joining a Battle

In order to join a new battle, the student has to start from his personal profile and click on the button ”My Tournaments”. The Tournament Microservice will be queried as a consequence and a list of tournaments to which the student is subscribed is displayed. This step is necessary because it is not possible to subscribe to a battle in a tournament without having previously joined the tournament itself. From the list of tournaments, the student will select the one in which s/he wants to join a battle, and the tournament home page will pop up. The tournament home page contains the list of battles for that specific tournament. From here, the student can browse all the battles available. When s/he clicks on one of these battles, the battle home page is shown. The following image gives a representation of the battle home page:



In the top right corner, the button to join the battle is displayed if the battle’s registration deadline hasn’t passed yet. In the left hand side, the battle description is provided, to make the student understand what kind of coding challenge the battle focuses on. On the right hand side, there is a list of scores associated to the student’s code solutions for that battle (initially empty) and the battle ranking (ranking of teams).

4 Requirements Traceability

In the following, a traceability matrix is reported to facilitate the mapping between the requirements, as presented in the Requirements Analysis and Specification Document (RASD), and the components of the CodeKataBattle system that are involved in meeting those requirements.

The various components that are mentioned here can be found in section 2.2 Component View of this Design Document, with a more detailed explanation of their role.

ReqID	Requirement Description	Components Involved
R1.1	The system allows students to log in the platform using their GitHub account.	Gateway Microservice (Dispatcher). User Management Microservice (Student Manager).
R1.2	The system notifies all students on the platform every time a new tournament is created.	Notification Microservice (Notification Manager, Notification Event Consumer). Tournament Microservice (Tournament Manager, Tournament Event Publisher).
R1.3	When the student requires it, the system shows the list of tournaments available on the platform for subscription (whose registration deadline hasn't passed yet).	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager).
R1.4	When the student requires it, the system shows the list of tournaments the student is subscribed to.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager).
R1.5	The system allows students to subscribe to tournaments, by the end of the registration deadlines set for the tournaments.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager).
R1.6	The system notifies all students subscribed to a tournament every time a new battle within that tournament is published.	Notification Microservice (Notification Manager, Notification Event Consumer). Battle Microservice (Battle Event Publisher). Tournament Microservice (Tournament Manager).

ReqID	Requirement Description	Components Involved
R1.7	The system allows students subscribed to a tournament to join battle(s) within that tournament, by the end of the registration deadline set for the battle(s).	Gateway Microservice (Dispatcher). Battle Microservice (Battle Manager).
R1.8	The system allows students subscribed to a tournament to join a battle of the tournament either on their own or creating teams with other students.	Gateway Microservice (Dispatcher). Battle Microservice (Battle Manager).
R1.9.1	The system creates a new GitHub repository dedicated to a battle right after the registration deadline for that battle passes.	Battle Microservice (Battle Manager). GitHub Integration Microservice (GitHub Manager).
R1.9.2	The system sends the link of the GitHub repository dedicated to a battle to all the students subscribed to that battle.	Notification Microservice (Notification Manager, Notification Event Consumer). GitHub Integration Microservice (GitHub Manager). Battle Microservice (Battle Manager).
R1.9.3	The system accepts notifications from GitHub in order to know when a new commit is performed by a student in any GitHub repository forked by the one created by the system itself.	GitHub Integration Microservice (GitHub Manager).
R1.9.4	The system pulls from the GitHub repositories of ongoing battles the new students' code solutions, every time it receives a notification from GitHub of a new commit in those repositories.	GitHub Integration Microservice (GitHub Manager).

ReqID	Requirement Description	Components Involved
R1.10	The system calculates and then publishes the new score of a team's solution every time it receives a notification from GitHub of a new commit performed by a member of the team.	GitHub Integration Microservice (GitHub Manager). Score Computation Microservice (Score Calculator). Battle Microservice (Battle Manager).
R1.11	The system notifies all students participating in a battle when the final ranking of teams is available on the platform.	Battle Microservice (Battle Manager, Battle Event Publisher). Notification Microservice (Notification Manager, Notification Event Consumer).
R1.12	The system notifies all students subscribed to a tournament when the tournament is closed by the educator who created it.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager, Tournament Event Publisher). Notification Microservice (Notification Manager, Notification Event Consumer).
R2.1	The system allows educators to log in the platform using their GitHub account.	Gateway Microservice (Dispatcher). User Management Microservice (Educator Manager).
R2.2	The system allows educators to create new tournaments.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager, Tournament Event Publisher).
R2.3	The system allows the educator that created a tournament to grant the permission of publishing battles within his/her tournament to other educators on the platform.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager, Tournament Event Publisher).

ReqID	Requirement Description	Components Involved
R2.4	The system allows educators to create new battle(s) in the tournaments they have the permissions to do so.	Gateway Microservice (Dispatcher). Battle Microservice (Battle Manager, Battle Event Publisher).
R2.5	The system doesn't take into account any new code solution for a battle after the submission deadline of that battle.	Battle Microservice (Battle Manager). GitHub Integration Microservice (GitHub Manager).
R2.6	After the submission deadline of a battle and only if a consolidation stage had been requested by the educator at battle creation time, the system sets a time frame for the educator that created the battle to allow him/her to assign personal scores to the students' code solutions.	Gateway Microservice (Dispatcher). Battle Microservice (Battle Manager).
R2.7	The system takes into account the personal scores assigned by the educator during the consolidation stage of a battle only if the educator assigned a score to all teams within the imposed time frame.	Battle Microservice (Battle Manager).
R2.8	At the end of a battle and after the consolidation stage (if requested), the system automatically calculates and publishes the final rank of all teams that participated in that battle.	Battle Microservice (Battle Manager).
R2.9	When a tournament is closed, the system automatically publishes the rank of all students that participated in the tournament.	Tournament Microservice (Tournament Manager).

ReqID	Requirement Description	Components Involved
R3.1	The system allows students subscribed to the same tournament to invite each other in order to join battles together as a team, respecting the minimum and maximum number of students permitted for a team in the battle.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager). Battle Microservice (Battle Manager). Notification Microservice (Notification Manager).
R3.2	The system allows students to accept or reject invitations from other student asking to join a battle together as a team.	Gateway Microservice (Dispatcher). Battle Microservice (Battle Manager). Notification Microservice (Notification Manager).
R3.3	The system allows students to set the name of their team when they join a battle with other students.	Gateway Microservice (Dispatcher). Battle Microservice (Battle Manager).
R3.4	The system calculates and then publishes the score assigned to a new code solution of a team in a battle, every time GitHub notifies the platform of a new commit performed by a member of such team.	GitHub Integration Microservice (GitHub Manager). Score Computation Microservice (Score Calculator). Battle Microservice (Battle Manager).
R3.5	The system constantly keeps updated the total ranking of teams for a battle, which evolves based on the new code solutions that are uploaded by students on the GitHub repository.	GitHub Integration Microservice (GitHub Manager). Score Computation Microservice (Score Calculator). Battle Microservice (Battle Manager).
R3.6	The system calculates and publishes the final ranking of teams when a battle ends.	Battle Microservice (Battle Manager).

ReqID	Requirement Description	Components Involved
R3.7	The system allows only the educators and students involved in a battle to see the partial or final ranking of teams for that battle.	Gateway Microservice (Dispatcher). Battle Microservice (Battle Manager).
R3.8	The system automatically calculates and keeps updated the total ranking of students subscribed to a tournament, based on the scores each student received in the battles he participated in.	Tournament Microservice (Tournament Manager, Tournament Event Consumer). Battle Microservice (Battle Manager, Battle Event Publisher).
R3.9	The system allows all users on the platform to see the partial or final rankings of tournaments.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager).
R3.10	The system allows educators to define customary badges (rewards) for the students participating in their tournaments, at tournament creation time.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager).
R3.11	The system assigns badges (rewards) to students that participated in a tournament, when the tournament is closed by the educator that created it.	Gateway Microservice (Dispatcher). Tournament Microservice (Tournament Manager). User Management Microservice (Student Manager).
R3.12	The system allows all users on the platform to search for the personal profile of a student and see the corresponding account.	Gateway Microservice (Dispatcher). User Management Microservice (Student Manager).
R3.13	The system allows all users on the platform to see the list of badges owned by a student for the tournaments s/he participated in.	Gateway Microservice (Dispatcher). User Management Microservice (Student Manager).

5 Implementation, Integration and Test plan

5.1 Overview

This section illustrates the implementation, integration and test plan for the CodeKataBattle platform.

The implementation plan is concerned with the order or schedule to follow to build the various components that make up the system. The integration plan shows how these components have to be assembled together in order to inter-operate and achieve a common result. Finally, the test plan is concerned with the way the various components are going to be tested when incorporated into the overall system.

The test plan that has been chosen for this application aligns with a thread strategy. The main reason for this choice is the fact that there is no real hierarchy that can be designed or recognized among the components of CodeKataBattle. Indeed, the design of the application as a set of microservices collaborating and communicating with each other, makes CodeKataBattle look more like a cluster of components, without a real natural ordering. Thus, the test plan will focus on identifying a series of incremental user-visible features that can be offered and seen by the stakeholders as the development process goes on. This way of handling the testing procedure is also beneficial to the stakeholders in the sense that the thread strategy fosters and enhances collaboration between developers and committers, as it allows developers to show incremental results of the implementation process.

As for the implementation plan, this has to match and align completely with the test plan, and this comes from the previous choice of handling the testing process with a thread strategy. Indeed, since the thread strategy requires to incrementally build user-visible portions of the software and then test them, the order in which components will be implemented follows the order in which components are integrated and then tested.

In the following sections of the document, a list of the major features that are going to be sequentially built and tested is provided, as well as a more detailed explanation of the build and test plan for each one of them.

In order to make the description as clear as possible, illustrations will be used to show the components involved and their ”use” relationships. More specifically, all the microservices making up CodeKataBattle will be displayed in every graphical representation but only the ones involved in the functionality being implemented will be highlighted with some colors. This to show every time the parts of the system that will be touched and modified and the ones that will be completely ignored in that development phase. The ”use” relationships are shown with dashed lines.

5.2 Implementation Plan

5.2.1 Features Identification

Six major features have been identified for the development of the CodeKataBattle system. The order in which they are listed below is relevant, as it will be the exact same order in which they will be implemented and tested.
All these feature are logical parts of the application that provide some user-visible functionality.

Login phase

The login phase is the entry point of the application. As soon as the user opens CodeKataBattle s/he has to login using his/her personal GitHub account in order to access the functionalities of the system. In this phase, the correct visualization of the login user interface and of the user personal profile are included. Indeed, when the login process is successfully completed, CodeKataBattle shows to the user his/her personal profile, from which further actions can be taken.

Creation of a new tournament

This feature is offered only to educators, who can create new tournaments for students on CodeKataBattle. This feature includes many functionalities that are related.
First off, the creation of a new tournament requires the correct display of the tournament creation form, which is the user interface responsible for asking to the educator all the mandatory pieces of information to correctly generate the tournament.
Then, as soon as the tournament is created, all students on CodeKataBattle have to be notified. Thus, this notification process is incorporated in this second feature.
The educator that created the tournament can also grant permissions to publish battles to other educators on the platform.
Finally, students have to be able to sign up for the tournament before the registration deadline, which is also to be considered here.

Creation of a new battle

Educators can create new battles in the tournaments they have permissions to do so. This feature includes a series of sub-functionalities to be accounted for.
First off, the creation of a battle requires the use of the battle creation form, which provides a way for the educator to input in the system all the mandatory data that is necessary to correctly create the battle.
As soon as the battle is created, all students subscribed to the tournament in which the battle resides have to be notified.
Students must be able to join the battle as a solo player or with other students as a team (also this part is incorporated in this feature), before the registration deadline for the battle passes.
Finally, when the registration deadline for the battle is over, the remote GitHub repository has to be created and all students who joined the battle must be notified with a message containing the link to the remote GitHub repository.

Pushing code on GitHub

This feature is related to the correct collaboration between CodeKataBattle and GitHub when a new code solution for a battle is pushed by any student on their forked GitHub repository.

This feature includes the correct receipt by the system of the GitHub notifications (to alert CodeKata-Battle of the new commit), the download process of the source code from the forked GitHub repository of the student, the analysis of the code solution in order to assign a score to it, the update of the battle ranking accordingly.

End of a battle

When a battle ends, some actions have to be taken.
First of all, if a consolidation stage was requested by the educator at battle creation time, the manual evaluations of the educator who created the battle are required. The CodeKataBattle system will initiate a timer that sets a time frame for the educator to assess the students' code solutions.
After the consolidation stage (if any) the battle can end and the final ranking of teams has to be drawn and published on the platform.
All students subscribed to the battle have to be notified of the battle terminating.
For the tournament that contains the battle, the tournament ranking must be updated with the new results of the battle that just ended.

End of a tournament

First off, the end of a tournament is established by the educator that created the tournament. When a tournament ends, the final ranking of students has to be published and all students subscribed to the tournament must be notified.

5.2.2 Components Integration and Testing

In the following section, the order in which the implementation and testing plan are described totally aligns with the order of features presented in the previous chapter. The aim of this implementation and test plan is to build the CodeKataBattle system in an incremental way, by adding a new feature in every cycle of the development process, and being able to show the improvements to stakeholders to seek validation.
As all the functionalities related to a specific feature have been implemented, integration testing will be performed on those specific components of the CodeKataBattle platform that collaborate to provide the new feature. In this way, no additional modification is introduced until the system is robust and fully tested on the already implemented services.
In the following, the main events occurring in the system and the interactions between the various microservices are reported here, classified based on the features presented in the previous section. The purpose of these descriptions is to highlight the particular functionalities that have to be implemented in each component of CodeKataBattle to provide the overall feature and test it, so to detail which parts of the application have to be improved or ignored in each stage of the development.
It is also important to mention the fact that since all communications from the users to the CodeKataBattle platform pass through the Gateway Microservice, this will almost always be involved in the implementation and integration testing of the below features.

Login phase, implementation and testing

The login phase is the entry point of the application. It is primarily handled by the Gateway Microservice, which receives all incoming requests of users and redirects them to the GitHub authentication page. When GitHub verifies the user's credentials, it will send back a message to the Gateway Microservice to inform it of the successful (or unsuccessful) authentication. At that point, the Gateway Microservice sends the username of the user to the User Management Microservice, which verifies whether the user is a new one on the platform of if s/he already has an account. The User Management Microservice is also responsible for retrieving all the user's data from the DBMS (if the user already had an account) and building the user interface for showing the personal profile of the user based on that data. If the user is a new one, a new profile is created.

Given these interactions between components of the CodeKataBattle platform, it's pretty straightforward to state that the only microservices that have to be partially implemented to serve the login feature are the Gateway Microservice and the User Management Microservice. Here's a schematic representation showing the components involved in this first development cycle and the ones left out.

It has to be noted that inside the Gateway Microservice, only the essential logic to drive the login is going to be implemented, as well as the User Management Microservice will be provided with the logic to handle users' data and build the users' profile interfaces. Another important point is related to the Discovery Service offered by the Gatway Micrsoervice, which is included as a component to be implemented in this section because all the other features rely on it for the location of microservices.

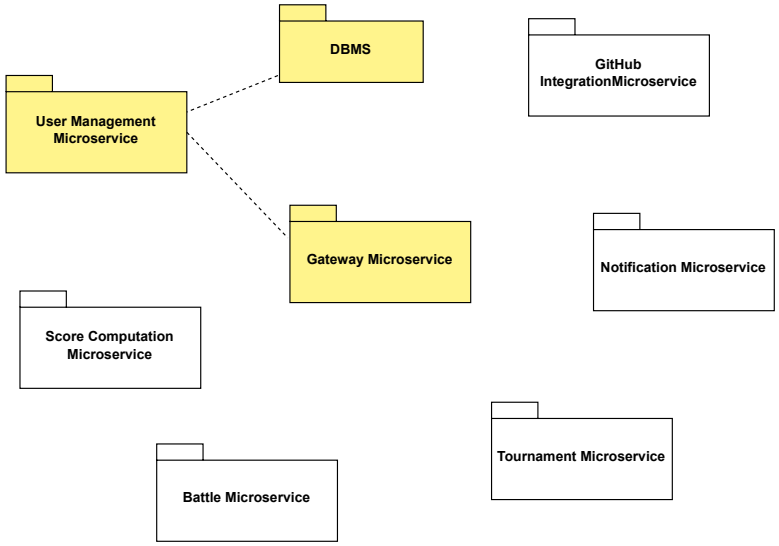


Figure 11: Schematic view of the development and testing of the login phase

Creation of a new tournament, implementation and testing

When creating a new tournament, the Gateway Microservice is involved, as it has to intercept the educator's requests and dispatch them to the right microservice, which is in this case the Tournament Microservice. So, part of the Dispatcher component inside the Gateway Microservice has to be implemented. The request for the new tournament creation is sent to the Tournament Microservice, which returns the tournament creation form (a user interface), which the Gateway shows to the user.

The educator is now able to fill the form out with the tournament data and confirm. The Gateway Microservice will then forward these pieces of information to the Tournament Microservice which has to create the new tournament in a persistent way (by accessing the DBMS) and then returning the new tournament home page (user interface) to the Gateway Microservice.

Finally, this feature of the application also includes the notifications that have to be sent to all students with an account on CodeKataBattle. As specified in this Design Document, the communications with the Notification Microservice are handles in an asynchronous way (event-driven programming). Therefore, at this point of the building plan, the event bus has to be built and the Tournament Microservice has to produce a new event every time a tournament is created.

At the same time, the Notification Microservice is involved, specifically in the process of reading events from the event bus and acting accordingly. At this layer of the implementation, it suffices to implement the logic to respond to the creation of a new tournament, which involves fabricating a new notification for all the students of CodeKataBattle. To do so, the Notification Microservice has to contact the User Management Microservice to retrieve the list of all the users on the platform.

After the tournament has been created on the system, the educator that fabricated the tournament must be able to grant permissions to publish battles to other educators. This feature is internal to the Tournament Microservice.

Finally, students must be able to subscribe to the tournament, which requires the Tournament Microservice to accept requests for tournament participation from the Gateway Microservice.

Here's an illustration of the involved components for this feature.

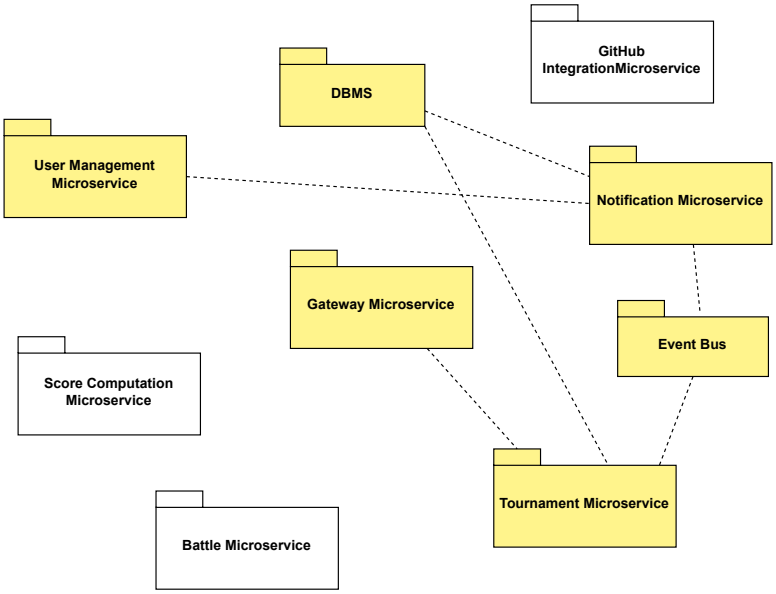


Figure 12: Schematic view of the development and testing of the creation of a new tournament

Creation of a new battle, implementation and testing

In order to create a new battle, the Battle Microservice has to provide the battle creation form (user interface) through the Gateway. The educator can use the battle creation form to feed the system with all the mandatory data necessary to correctly generate a new battle.

Once the educator confirms his/her choices on the battle creation form, the Gateway Microservice must forward this pieces of information to the Battle Microservice, which will take care of instantiating the new battle in a persistent way (accessing the DBMS).

The next step for the implementation of this feature is the notification of all students subscribed to the tournament in which the battle resides. In order to achieve this:

- The Battle Microservice has to publish a new event when the battle is successfully created.
- The Notification Microservice listens to the event.
- The Notification Microservice leverages the Tournament Microservice to obtain a list of all the students subscribed to the tournament in which the battle resides.
- The Notification Microservice produces a notification for all the students in the list

Therefore, the Notification Microservice and Tournament Microservice are involved in this step and have to offer these new functionalities.

Moving on, students must be able to join the new battle before the battle registration deadline passes, either as single players or in a team with students. As for the single player mode, this is achieved easily involving only the Battle Microservice, which will receive the student's request through the Gateway. As for the participation to the battle as a team, students can invite other students to form a team, so the Notification Microservice is involved in this case. The request of a student to invite another student to join a battle together as a team is intercepted by the Gateway Microservice, which dispatches it to the Battle Microservice. The Battle Microservice leverages the Notification Microservice (through the RESTful NotificationAPI) in order to produce an individual notification for the student invited to join the battle in a team.

Finally, the GitHub remote repository dedicated to a battle has to be created when the battle registration deadline passes. In this case, the GitHub Integration Microservice takes care of the task. Once the GitHub repository is created, the GitHub Integration Microservice generates a new event. The Notification Microservices reads the event, contacts the Battle Microservice to obtain a list of all students subscribed to the battle and fabricates the corresponding notifications for all the students on the list.

Here is an illustration of the components involved in the development of this feature, which will be enriched with the functionalities explained above and then tested to see if they integrate and interoperate to provide the feature of creation of a battle.

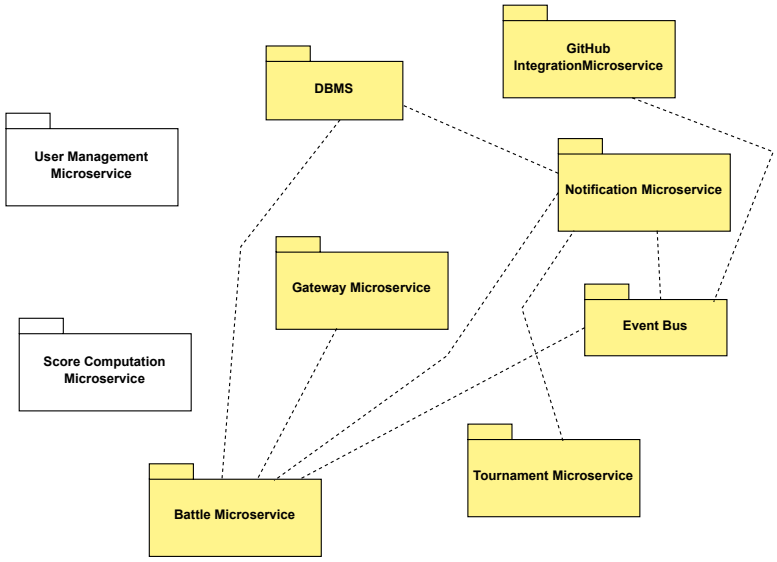


Figure 13: Schematic view of the development and testing of the creation of a new battle

Pushing Code on GitHub, implementation and testing

When a student pushes a new code solution on GitHub, many components of the CodeKataBattle platform are involved in a cohesive response to this event.

First, the Gateway Microservice must be always listening to notifications coming from GitHub. Once a new notification arrives, the Gateway Microservice leverages the GitHub Integration Microservice, which takes care of downloading the source code from the correct remote repository and then employs the Score Computation Microservice (through the CalculatorAPI) to compute the score to assign to the solution.

The Score Computation Microservice exploits the interface exposed by the Battle Microservice in order to update the score associated to a team with the new solution provided on GitHub. The ranking of the battle has to be udpated accordingly.

Here's an illustration of the components involved in this specific feature:

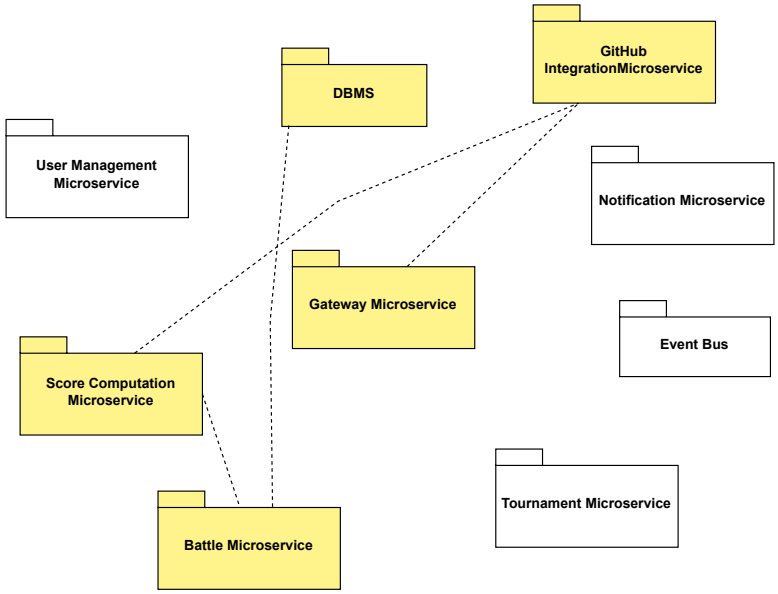


Figure 14: Schematic view of the development and testing of the interactions with GitHub pushing the code

End of a battle, implementation and testing

The end of a battle first requires the consolidation stage to be handled (if any). The logic for this stage can be integrated in the Battle Microservice.

Then, the battle final ranking has to be published and a notification must be sent to all the students involved in the battle. This implies that the battle has to publish an event and the Notification Microservice has to listen to it. The Notification Microservice then contacts the Battle Microservice to request a list of all students taking part in the battle and fabricates all the notifications accordingly.

Finally, the tournament in which the battle that terminated resides must be updated in terms of its ranking. This is achieved by making the Tournament Microservice listen to the events on the event bus.

Here's an illustration of the components involved and left out in this feature:

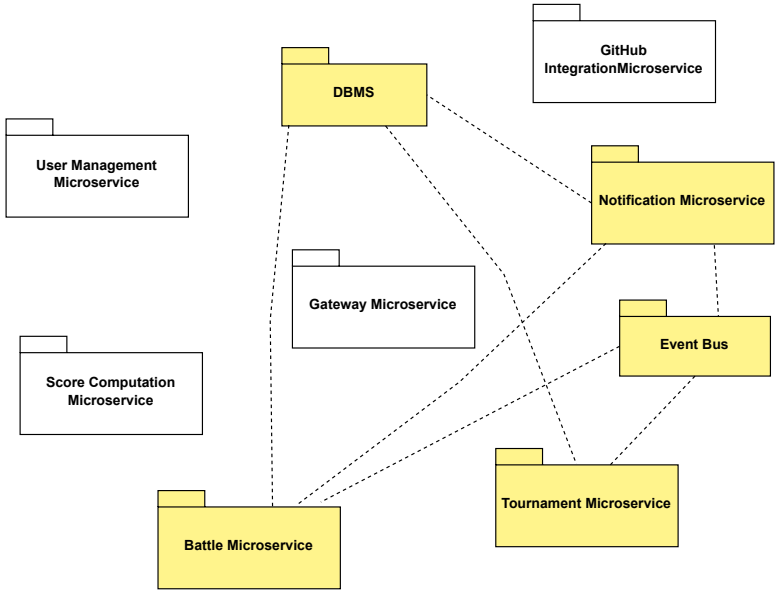


Figure 15: Schematic view of the development and testing of the end of a battle

End of a tournament, implementation and testing

It is only the educator who originally created the tournament to be able to close it. This request is intercepted by the Gateway Microservice, which contacts the Tournament Microservice to close the tournament. In turn, the Tournament Microservice will publish an event on the event bus, so that the Notification Microservice can read it, contact again the Tournament Microservice to obtain a list of all students subscribed to the tournament and fabricate a new notification for all of them.

Here's an illustration of the components involved in this feature and of the ones ignored.

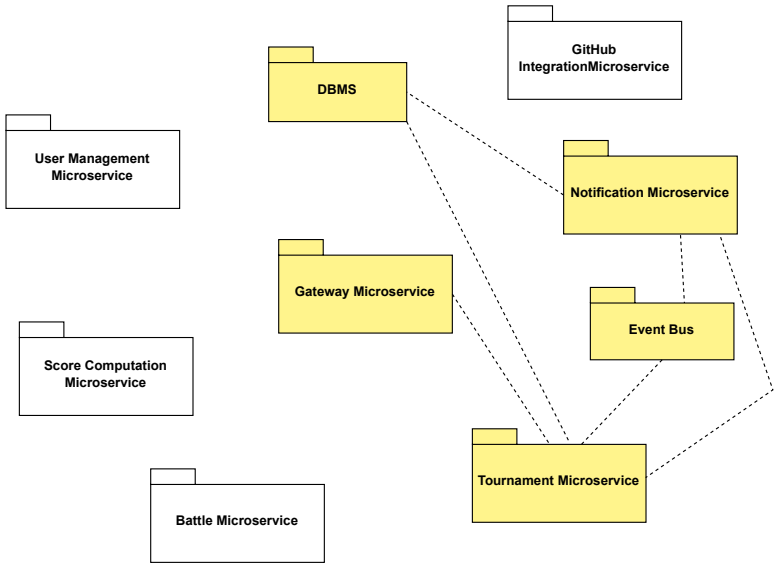


Figure 16: Schematic view of the development and testing of the end of a tournament

5.3 System Testing

System testing is a critical phase where the complete integrated system is put to the test. The testing environment should closely mirror the actual production setup to ensure a complete evaluation of the software's final version. To increase the chances of a successful initial software release, we perform both functional testing and non-functional one (performance, load and stress testing).

Functional testing checks if the software meets the requirements outlined in the CodeKataBattle 's Requirements Analysis and Specification Document (RASD), as described, for example, in the use cases: after implementing the software, it's crucial to confirm that these requirements have been met.

Performance testing includes non-functional tests like assessing response time, throughput, and identifying any architectural issues; the goal is to determine if adjustments are needed in the architecture of the Document Design (DD) to meet the software's requirements.

Load testing, a subset of performance testing, involves gradually increasing the workload or sustaining loading to ensure that the system can handle the expected number of users.

Stress testing focuses on the system's capacity of recovery after a failure, emphasizing availability over complete reliability, as requested for the software.

For test case generation in the CKB system, we use both manual and automatic methods. Manual testing helps identify specific situations, while automatic testing ensures a comprehensive assessment of various scenarios. To cover the entire system, automated testing employs either concolic execution, and fuzz testing, and search-based strategy.

It's essential to note that simpler tests, like those outlined as asserts in the Alloy model of the CKB system in the RASD, serve as a starting point for system testing.

6 Effort spent

	Alessandro	Matteo	Sara
Introduction	21	28	15
Architectural Design	17	25	15
User Interface Design	3	15	4
Requirements Traceability	1	5	0
Implementation, Integration and Test Plan	15	18	5

Table 16: Time spent on every section of the DD for each student

7 References

- Sequence Diagrams made with StarUML
- Chapter 5 Diagrams made with Draw.io
- All other Diagrams made with Figma