CodeKataBattle
Design Document

Software Engeneering 2

Alessandro Saccone, Matteo Sissa, Sara Zappia

January 3, 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.

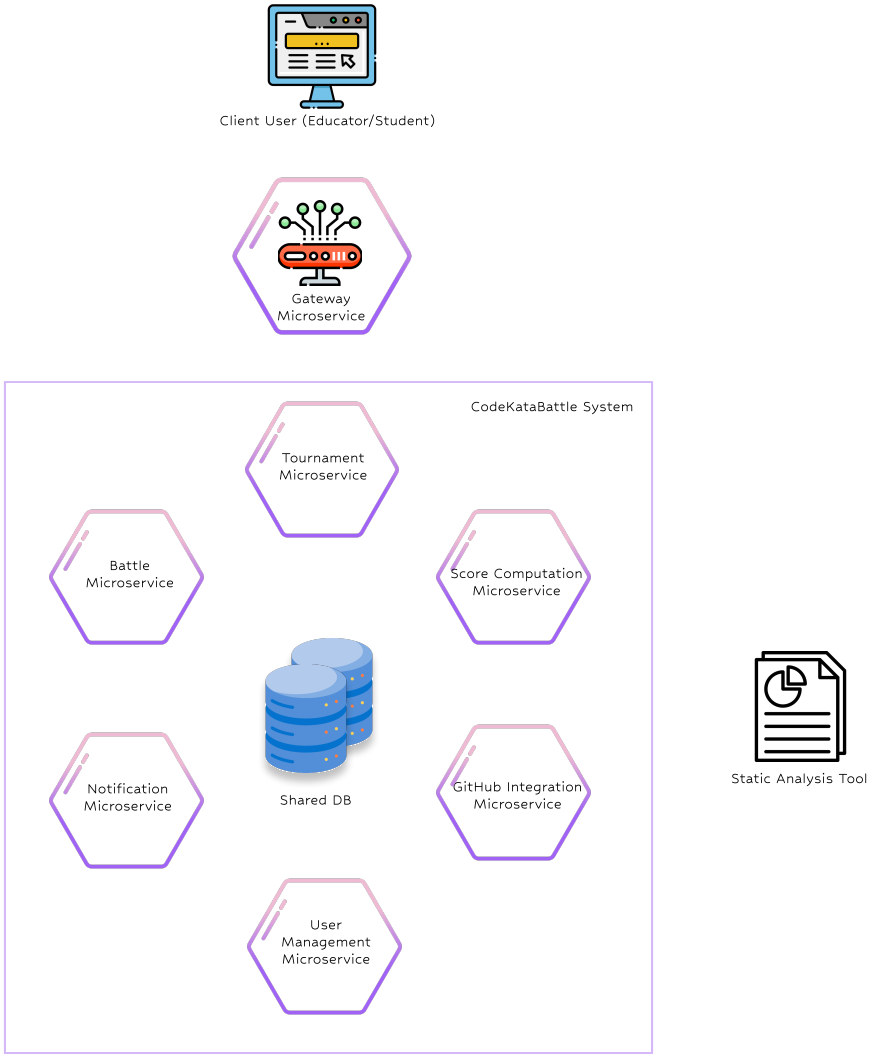


Figure 1: High-level view of the major components

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 CKB 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:

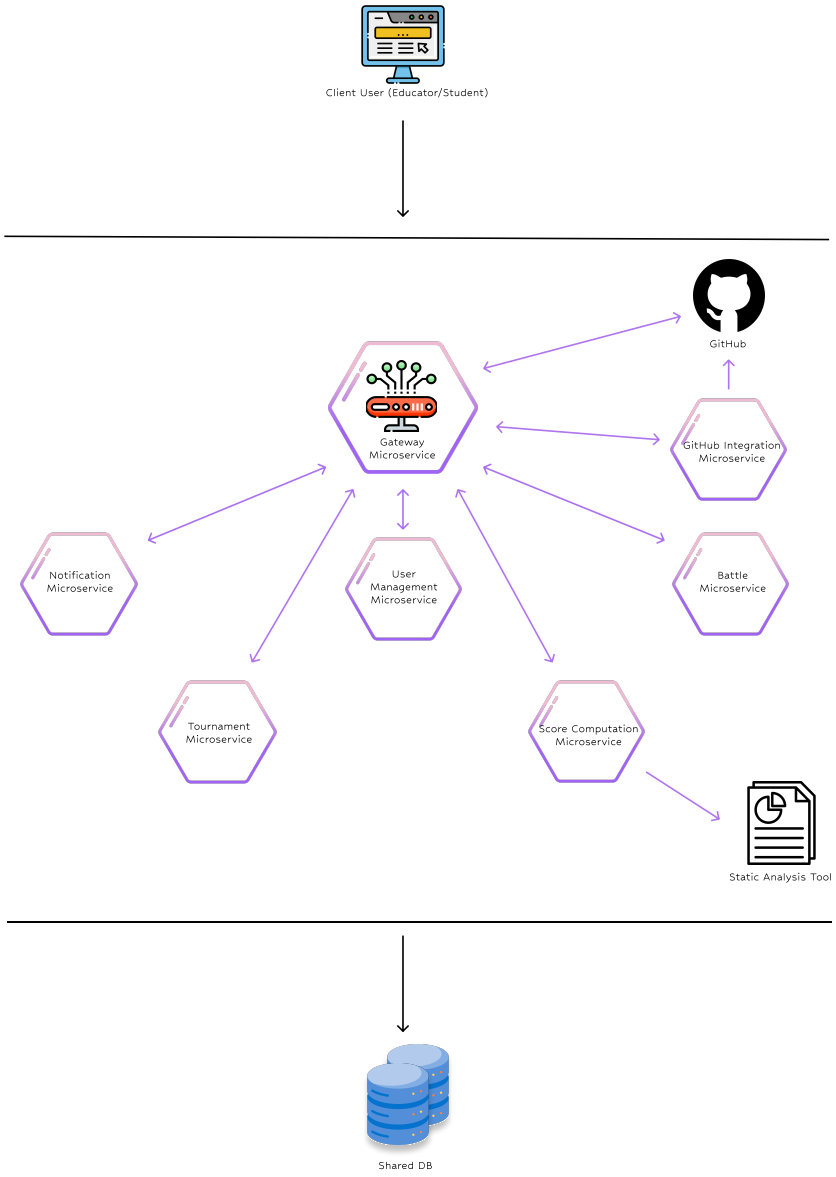


Figure 2: Overview of microservices and main interactions

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.

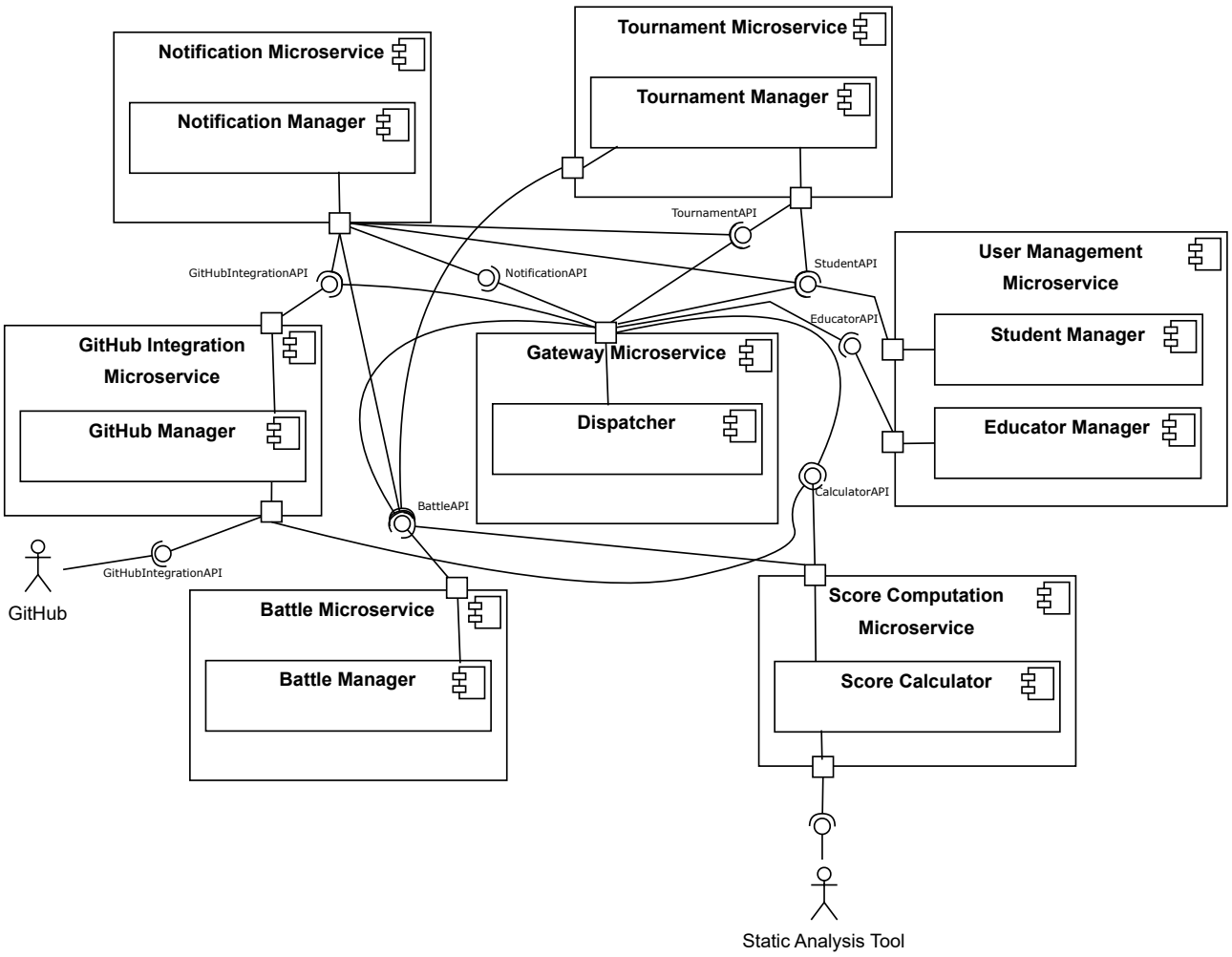
2.2 Component View

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.

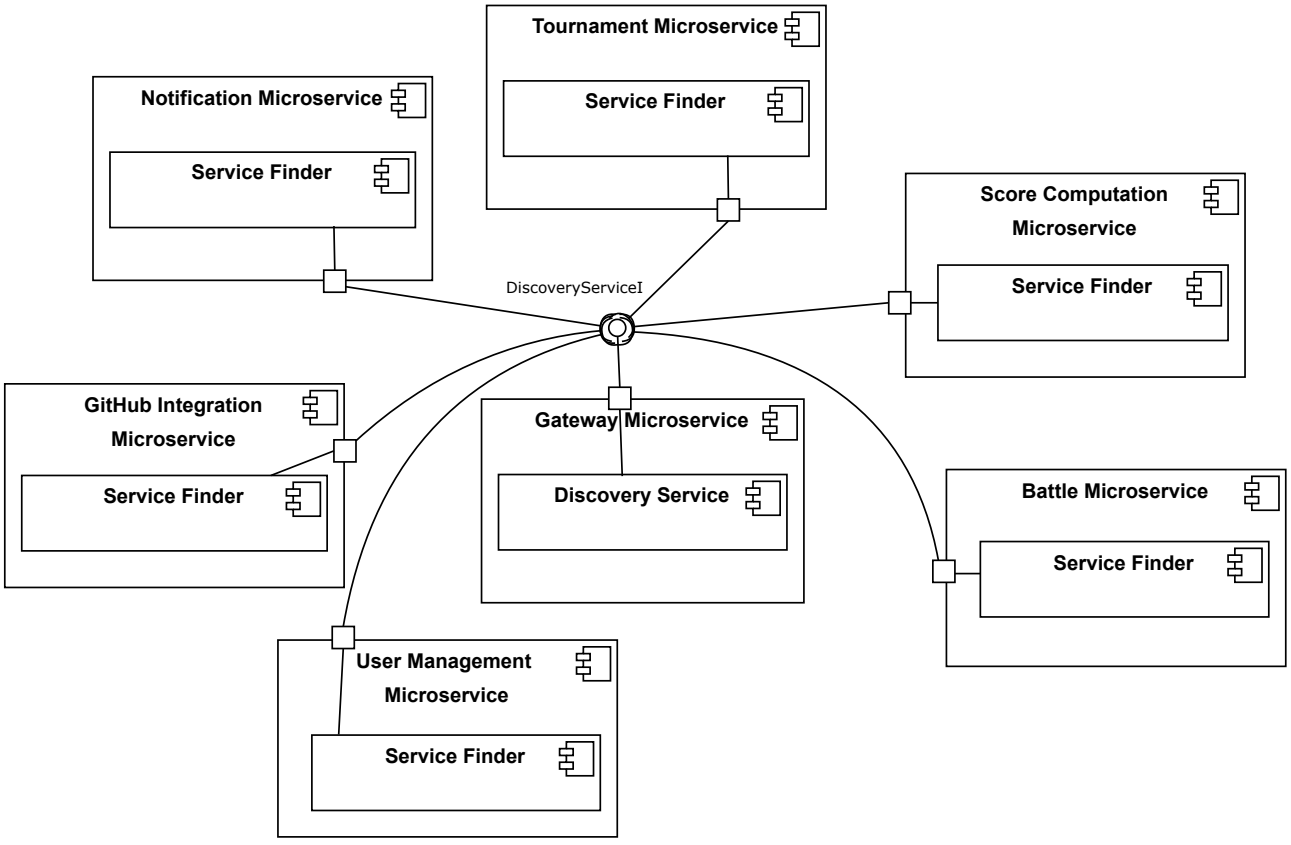
This diagram has to be read and interpreted also accounting for the event-driven paradigm that is adopted in the project. Some of the interactions and flows of events inside the system happen in an asynchronous manner with an event-driven approach that is further described in the following views. That's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

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.

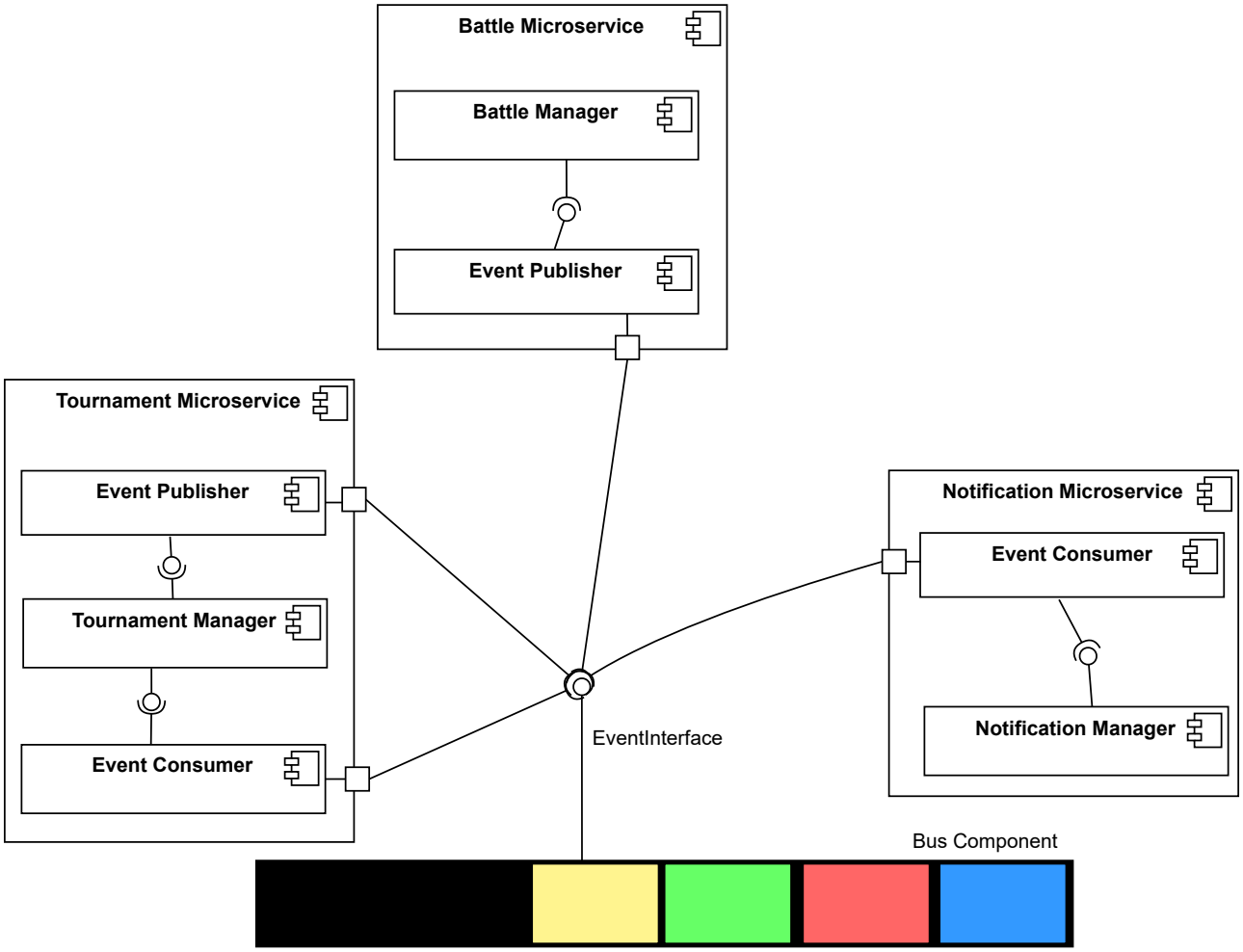
Here’s the diagram:



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.

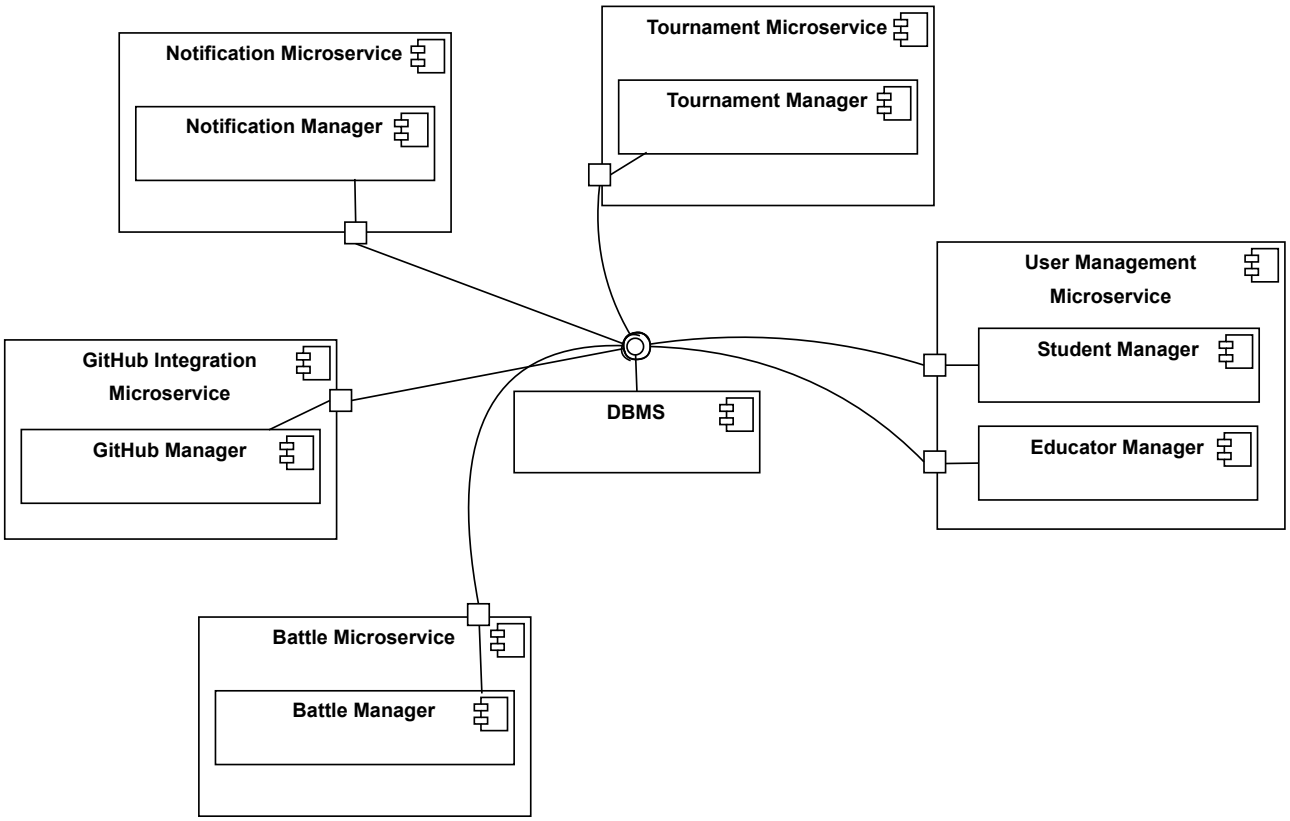
The Tournament Microservice, for instance, has both the Event Publisher component and the Event Consumer component. Indeed, the Tournament Microservice has to publish an event every time a new tournament is created or closed. At the same time, the tournament has to be able to read from the bus when a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

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

The 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).



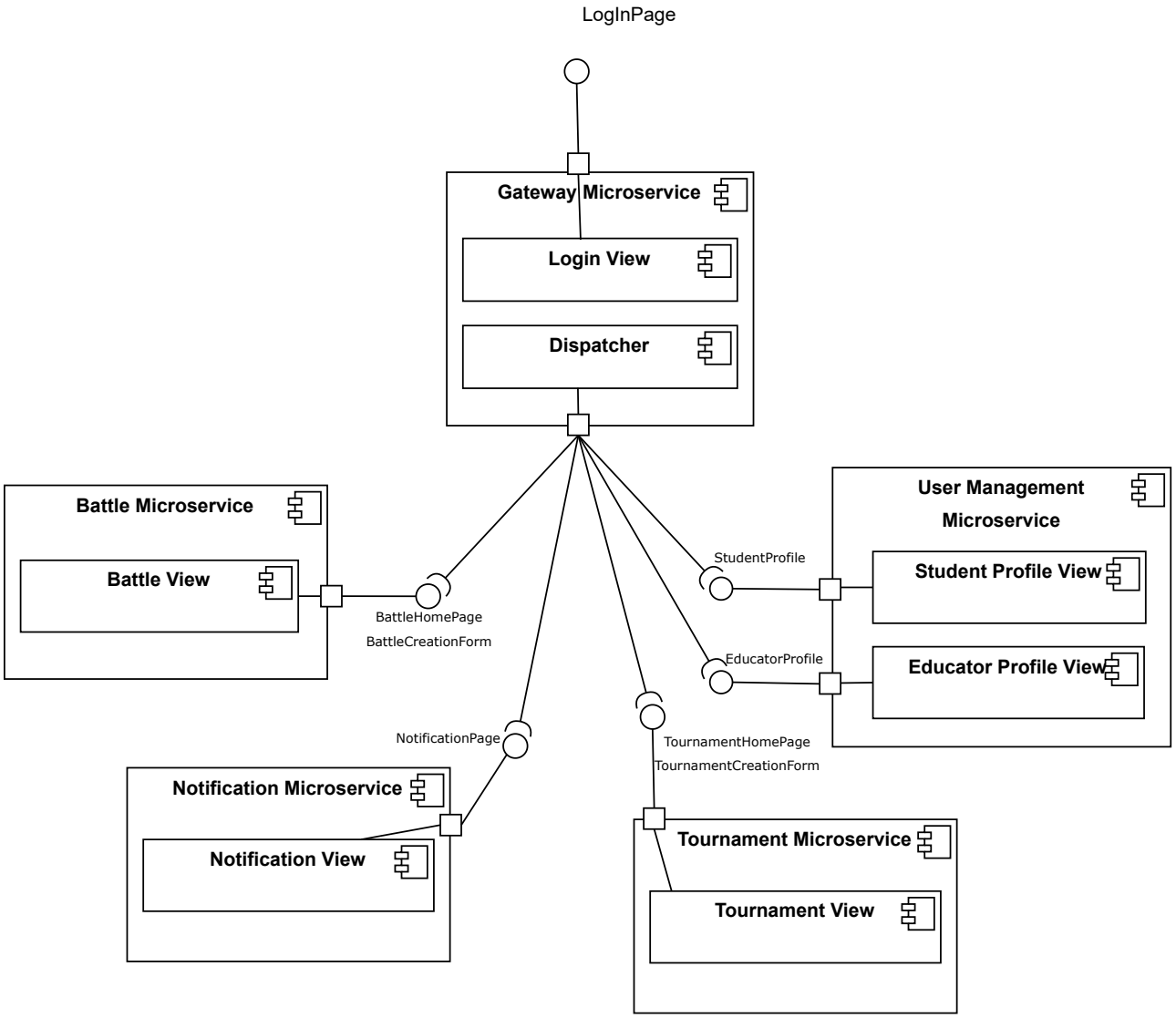
The DBMS exposes an interface that is employed by all microservices to work with the persistent data stored in the DBMS.
Looking at every microservice individually, it is possible to notice that they all are equipped with a ”Manager” component, which is responsible for manipulating and accessing data.
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.



2.3 Deployment View

2.4 Run Time Views

2.5 Component Interfaces

2.6 Selected Architectural Styles and Patterns

The choice of this architectural style is due to many factors, including:

- *Scalability*: Microservices can be scaled independently, allowing for scaling out sub-services without scaling out the entire system. This leads to the versatility of the application.
- *Fault Tolerance*: Unlike the monolithic approach, which has many inter-dependencies creating a single point of failure, in this approach, the application can remain mostly unaffected by the failure of a single module.
- *Deployment and Productivity*: Microservices enable continuous integration and delivery, making it easy to test new ideas, suiting Agile and DevOps working methodologies. Furthermore, it makes it easier to split the work between team members: each team member is responsible for a particular service, resulting in a smart, productive, cross-functional team where the speed of development is largely improved.
- *Continuous Delivery*: Microservices enable continuous delivery, meaning your software can be modified and delivered to your client base frequently and easily due to its automated nature.
- *Maintainability*: Benefits of Microservices Architecture include less energy spent on understanding separate pieces of software or worrying about how a bug fix will affect other parts of the product.

2.7 Other Design Decisions

3 User Interface Design

4 Requirements Traceability

5 Implementation, Integration and Test plan

5.1 Overview

5.2 Implementation Plan

5.2.1 Features Identification

5.2.2 Components Integration and Testing

5.3 System Testing

5.4 Additional Specifications on testing

6 Effort spent

	Alessandro	Matteo	Sara
Introduction	0	12	0
Architectural Design	0	10	0
User Interface Design	0	0	0
Requirements Traceability	0	0	0
Implementation, Integration and Test Plan	0	0	0

Table 2: Time spent on every section of the DD for each student

7 References