Code Kata Battle - Eusebio Alberto,
Martini Marcello

Design Document

Deliverable:	DD
Title:	Design Document
Authors:	Eusebio Alberto, Martini Marcello
Version:	1.0
Date:	07-January-2024
Download page:	https://github.com/martinimarcello00/EusebioMartini
Copyright:	Copyright © 2024, Eusebio Alberto, Martini Marcello – All rights reserved

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

The primary goal of this document is to guide the development team in bringing the system to life. It offers a comprehensive overview of the adopted system architecture, delving into the intricate details of its various components and their interactions with one another. Additionally, the Design Document (DD) outlines detailed implementation, integration, and testing plans, meticulously crafted while considering the priority, effort requirement, and stakeholder impact of each individual component.

1.2 Scope

In recent years, the need for practical, hands-on software development training has become increasingly evident in educational environments. The CKB platform is an instrument to support instructors in the teaching process and students in improving their coding skills. The platform is held online and easily accessible by users through an application interface.

Instructors can create katas through the platform and such katas can be addressed by several groups of students in a programming language of choice.

The platform allows Educators to define deadlines on subscription and submission as long as other constraints such as the maximum and minimum number of students per group and other mechanisms for scoring.

The problems are addressed in battles, and organized in tournaments, where the groups of students can participate.

Participation in a tournament can improve a student's rank on the platform and grant them badges, based on the performance measured.

The thresholds for badges, their release, and the creation of other variables are features available only to educators, but their visualization is open to every other user.

The platform relies on GitHub actions for the creation of repositories containing code katas and the automatic testing of the proposed solutions. The students are required to fork the repositories and set up an automatic workflow through GitHub Actions to inform CKB that a commit has been made.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Code kata battle**: a programming exercise proposed by an instructor and tested by the machine. code kata battles are composed of the following elements:
  - A brief textual description of the problem
  - A software program with build automation scripts
  - A set of test cases that will be the base to evaluate the proposed solution
- **API**: stands for Application Program Interface and are a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.
- **Commit**: a name to indicate the action of saving the status of the code in versioning control systems such as Git
- **Push**: the action of uploading the local versioning history to a GitHub repository

- **Pull**: the action of updating the local versioning history with code coming from a GitHub repository
- **Badge**: a digital award that can be obtained through the matching of a set of rules
- **Rule**: a mathematical relation between variables
- **Variable**: a label associated to a measurable quantity
- **Rank**: an incremental value associated with each student registered on the application and updated through the solution of Katas
- **Score**: a natural number comprised between 0 and 100

1.3.2 Acronyms

- **kata**: abbreviation for code kata battle
- **CKB**: used to identify the Code Kata Battle platform

1.3.3 Abbreviations

- **Gi**: goal number i
- **Wi**: world phenomena number i
- **SPi**: shared phenomena number i
- **Ri**: requirement number i
- **UCSi**: use case scenario number i
- **PFi**: product function number i
- **DAi**: domain assumption number i

1.4 Revision history

- Version 1.0 (7 January 2024)

1.5 Reference Documents

This document is strictly based on:

- The specification of the RASD and DD assignment of the Software Engineering II course, held by Professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2023/2024
- Slides of Software Engineering 2 course on WeBeep;

1.6 Document Structure

Mainly the current document is divided into 5 chapters, which are:

1. **Introduction**: The first chapter includes the introduction which explains the purpose of the document, then, a brief recall of the concepts introduced in the RASD is given. Finally, important information for the reader is given, i.e. definitions, acronyms, synonyms, and the set of documents referenced.

2. **Architectural Design**: it includes a detailed description of the architecture of the system, including the high-level view of the elements, the software components of CLup, a description through runtime diagrams of various functionalities of the system, and, finally, an in-depth explanation of the architectural pattern used.

3. **User Interface Design**: are provided the mockups of the application user interfaces, with the links between them to help in understanding the flow between them.

4. **Requirements Traceability**: it describes the connections between the requirements defined in the RASD and the components described in the first chapter. This is used as proof that the design decisions have been taken with respect to the requirements, and therefore that the designed system can fulfill the goals.

5. **Implementation, Integration and Test Plan**: it describes the process of implementation, integration, and testing to which developers have to stick in order to produce the correct system in a correct way.

2 Architectural design

2.1 Overview: High-level components and their interaction

This section gives an overview of the architectural elements that compose the system, their interaction, and a description of the replication mechanism chosen for the system to make it distributed.

2.1.1 Three-Tier Application Structure

Presentation Tier

At the forefront of the system is the Presentation tier, where users interact with the platform. The web pages generated by the Web Server are delivered to users, providing an intuitive interface for effortless navigation and interaction. This tier serves as the gateway for users to submit their code for evaluation and receive timely feedback.

Application Tier

The heart of the platform lies in the Application tier, which orchestrates the core functionalities. Whenever a user commits code, the Application server is invoked via API calls called by the GitHub Action. This server, in turn, triggers another service responsible for creating an isolated test environment. The user's solution is evaluated within this environment using test cases sourced from GitHub and preset Static Analysis Tools. Finally, the scores are returned to the Application Servers and the test environment is destroyed.

Data Tier

The Data tier acts as the repository for essential information. Leveraging a clustered Database Management System (DBMS), user and platform data such as battles or tournaments data are stored securely. This distributed approach not only enhances data retrieval performance but also contributes to fault tolerance by avoiding a single point of failure.

2.1.2 External services

GitHub serves as a reliable storage solution for code-related artifacts, while the **Email Server** plays a role in communication and notification, allowing the users to be notified when a tournament or a battle is open.

2.1.3 Load Balancing and Distributed Systems

A load-balancing mechanism is implemented to fortify the platform against potential downtimes and bottlenecks. By distributing incoming requests across multiple servers, the system ensures optimal resource utilization and guards against a single point of failure. This distributed approach aligns with the principles of fault tolerance and scalability.

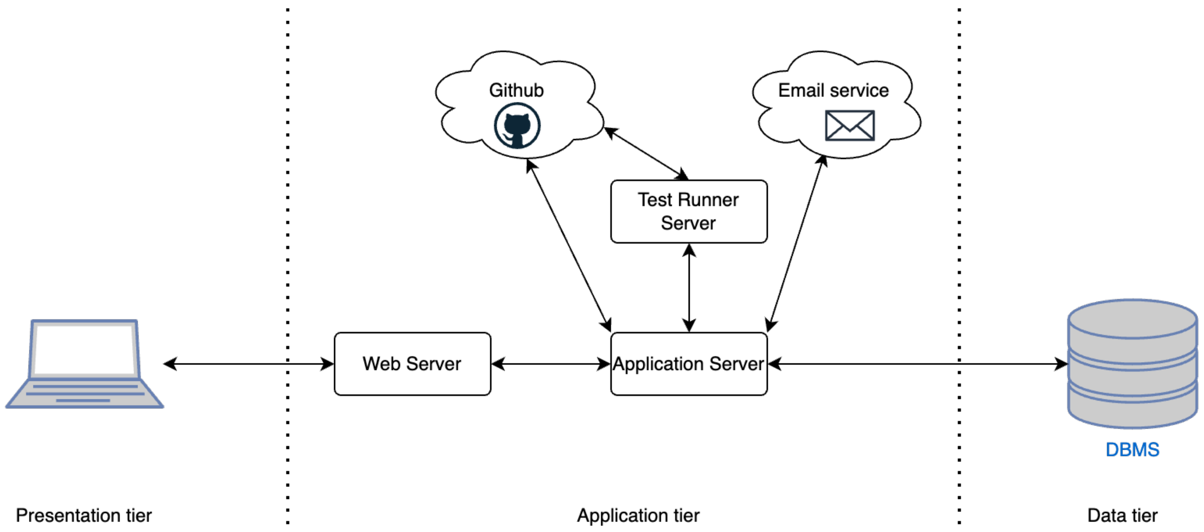


Figure 1: High-level system overview

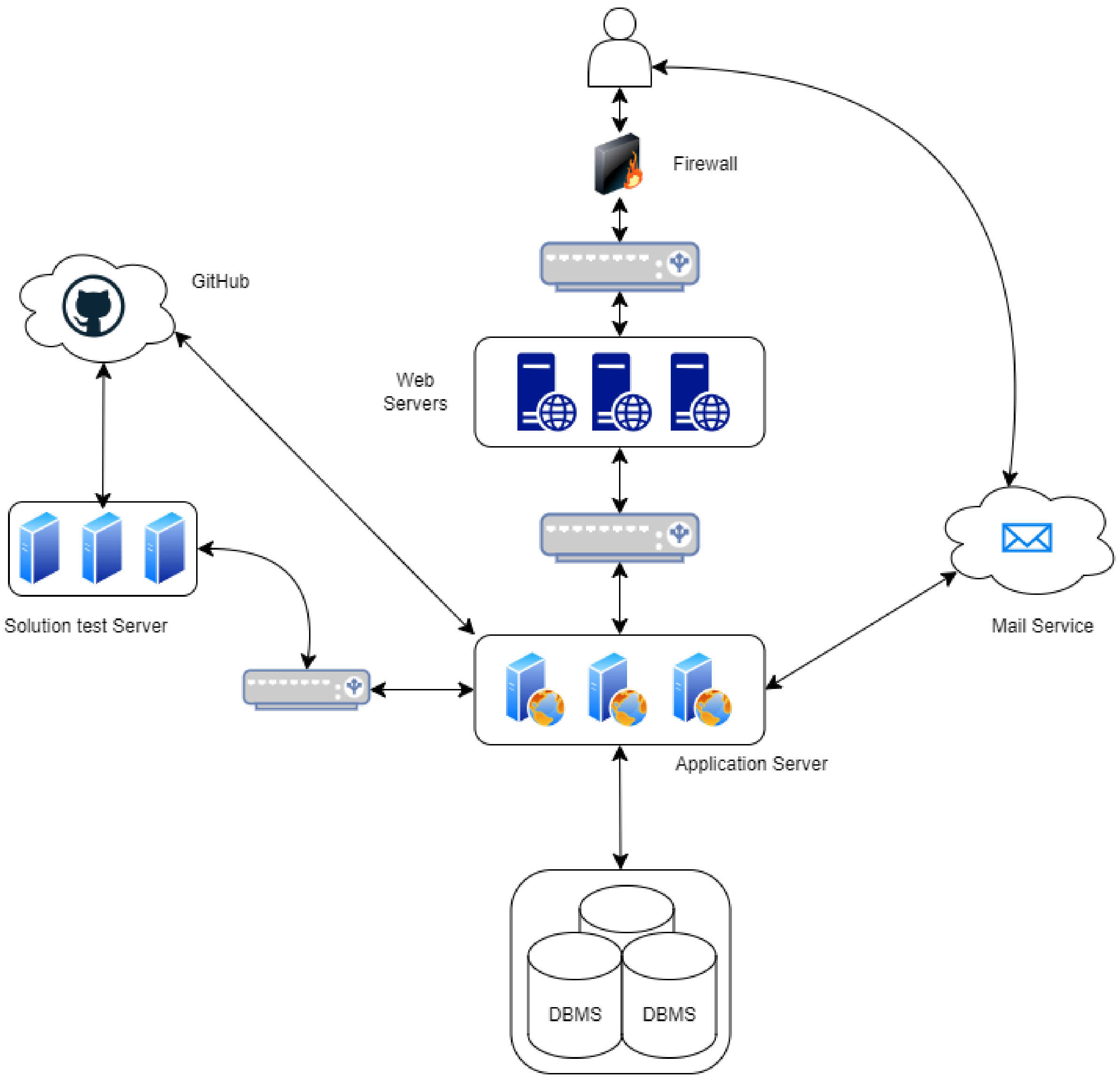


Figure 2: Distributed view

2.2 Component view

Figure 3 represents the component diagrams of the system. All the components in yellow are elements of the application server. The WebServer and the SolutionTestRunner are colored in blue, indicating that, while they're not part of the application server, they are inside the application layer. The elements colored in red, instead, represent the components directly provided to users for interacting with the system, i.e. the ones that belong to the Presentation tier. The only element colored in green is the DBMS, which is the only one belonging to the data tier. Finally, the GitHub and Email provider components are represented in purple to highlight the fact that they are elements provided by a third party, hence not belonging strictly to CKB.

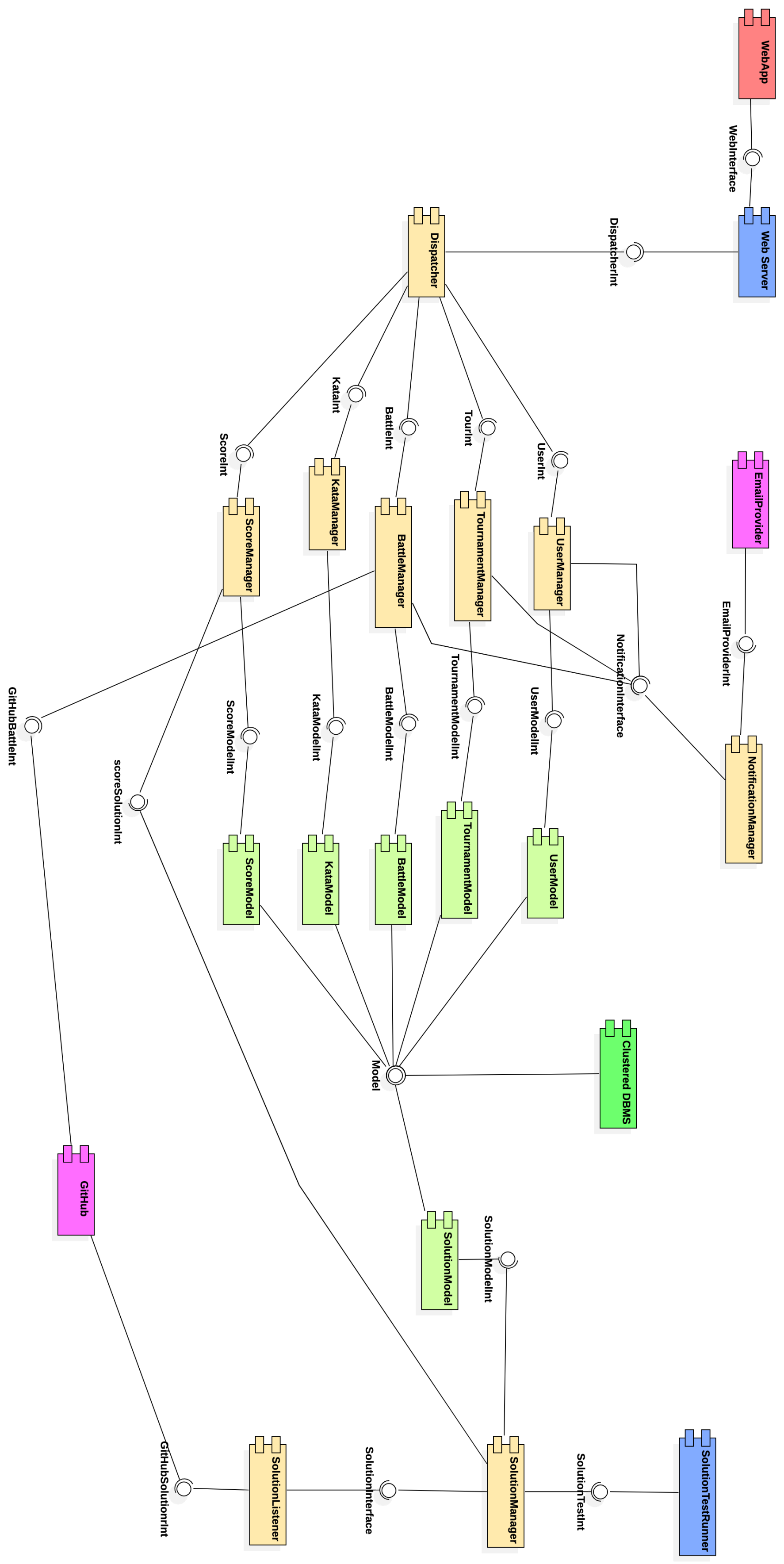


Figure 3: Components diagram

- The **WebServer** component is designed to satisfy all the requests from the CKB platform accessed by all the users from a web browser. It sends the requests to the dispatcher that is part of the Application server which, after forwarding them to the competent components, returns the rendered web pages that are finally forwarded to the users.
- The **WebApp** is the web interface used by the users of the platform. The WebApp communicates with the WebServer via the HTTPS protocol by sending requests to the web server and waiting for the HTML response.
- The **Dispatcher** takes in all incoming requests from the web app, directs them to the appropriate component, and after receiving the responses, sends them back to the clients.
- The **EmailProvider** is an external service that exposes an API from which the system can send emails to the users of the CKB platform. The service is connected to the NotificationManager component.
- The **NotificationManager** component allows the platform to send notifications to the users. It implements the API of the EmailProvider external service and can be used from other components to send emails to the user for instance when a tournament is created the system sends an email to all the students. The methods implemented in the component can be used by other components of the platform.
- The **UserManager** component is responsible for user management on the platform. Its main tasks are handling the login requests of the users, checking whether a user is allowed to sign up to the platform (through the email domain check), and assigning the right authorization and role to the new users. The component generates the email verification token that is sent through email to verify the email address of the user.
- The **TournamentManager** component is in charge of handling the tournament. Instructors can create and close tournaments, and they have the option to assign other instructors to manage a tournament. Once a tournament is created, students will get a notification via email and can sign up for it.
- The **BattleManager** component is responsible for managing the Battle inside the platform. Instructors can create battles inside a tournament and then students can enroll in them. This manager handles the student teams for each battle, following the rules set by instructors when they make the battle.
- The **KataManager** component is in charge of handling the Katas inside a battle. It manages the process in which the instructor creates a Kata and ensures that all the provided test cases and build scripts are correct. The uploaded files will then be used to create a container image that will be used by the SolutionTestRunner to create the isolated environment where the solution will be tested.
- The **SolutionManager** component handle the solutions uploaded by the students for each kata. It coordinates the SolutionTestRunner and ensures that each request committed in a repository will be tested and the execution parameters will be stored. The SolutionListener triggers it which gives access to the platform to Github.
- The **SolutionListener** is a component that exposes an API used by the GitHub Action to inform whenever a new solution is committed in the students' repositories. Every time a new solution is committed, the listener pulls the solution from the repository and sends it to the SolutionManager which will execute the tests through the SolutionTestRunner.
- The **GitHub** external service is used by the platform for creating the repositories for each battle that students will fork, and as storage for students to put the solutions to Katas.

- The **SolutionTestRunner** is a server that is part of the system but external w.r.t. the Application server. It is in charge of creating an isolated environment with a specific amount of resources to securely test the solution provided by the students. This system is controlled by the SolutionManager component which triggers the jobs by specifying which container image should be used to test the solution and the solution pulled by GitHub. The SolutionTestRunner will return the execution parameters and the output produced to the SolutionManager.
- The **ScoreManager** component is in charge of scoring the solutions of the students based on the execution parameters. The components allow the instructor to manually score a solution and let users see the ranks of a tournament.
- The **UserModel**, **TournamentModel**, **BattleModel**, **KataModel**, **ScoreModel** and **SolutionModel** are components used to store the data that are part of the persistent state of the system in the database. Each model acts as a table in the database and an entity in the system, creating an Object-Relational Mapping (ORM). Furthermore, each component provides all the methods that are necessary to access the data and it also implements the logic to manipulate them.

2.3 Deployment view

This chapter describes the deployment view for the CKB platform ((figure 4). This view describes the execution environment of the system, together with the geographical distribution of the hardware components that execute the software composing the application.

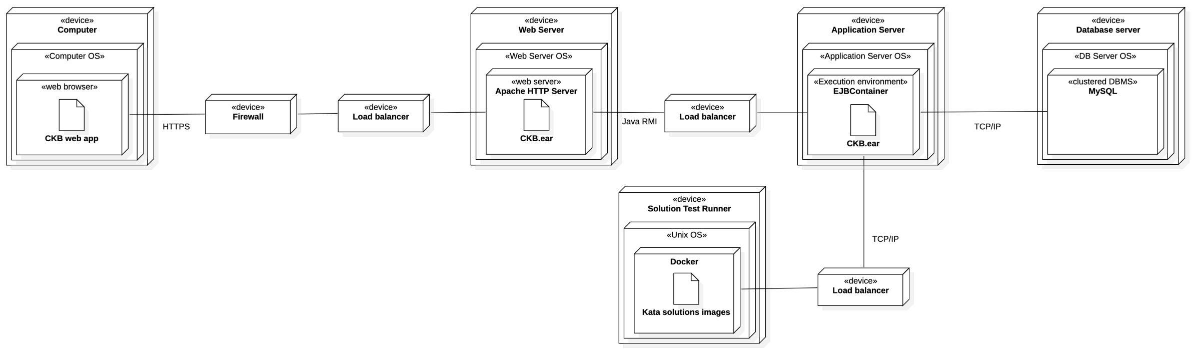


Figure 4: Deployment diagram

Further details about the elements in the figure 4 are provided in the following.

- A **computer** is a user's personal computer. The only software required is a web browser, which is necessary to access the CKB web application. To establish an HTTPS connection with the system, the request first passes through a firewall and then is forwarded by the load balancer to an available web server.
- The **firewall** is a device that monitors the packets incoming to the system, if a packet is potentially dangerous it is not forwarder. It is placed before the load balancer right after the users' computers so that the only packets that enter the network are considered safe.
- A **load balancer** is a device that distributes incoming requests across a pool of servers, such as application servers, web servers, or database servers. This load-balancing process aims to improve the overall system's performance, scalability, and reliability. In the CKB system, three load balancers distribute the incoming requests for the web server, the application server, and the solution test runner server.

- The **web server** receives all the requests sent from the users of the CKB platform via a web browser. It then forwards the received request to the application server for processing, using Java RMIs. Furthermore, once the application server returns the results of the required computations, the web server renders the web page containing such data and forwards it to the user. In the alternative, if the received request from the user asks for a static web page (e.g. the login form or the registration page), the web server returns it right away without even using the application server.
- The **application server** receives all the requests sent from users via web browsers that interact with the CKB web app. It is the most important element of the business tier it is designated for data elaboration and management. It is the only element of the architecture that can communicate with the database and it does so via TCP/IP. Its performance is improved by the use of a load balancer and some replicas of the application server so that the system is reliable and can scale.
- The **database server** represents a cluster of databases managed via the MySQL NDB Cluster CGE. This will guarantee virtually synchronous data replication on all the members of the cluster. Therefore, even if the system can be slowed down by the complexity of the replication mechanism, it will grant a very high availability and always provide up-to-date information to the application server. Furthermore, the last version supports up to 4 data replicas to enhance the data reliability.
- The **Solution Test Runner** is a pool of servers managed by a load balancer that is responsible for running in an isolated environment the students' solutions. Each machine runs Docker, an operating system for containers. Each time a new solution is provided, the Application server triggers via docker API the creation of a container (an isolated environment that contains a base OS and all the tools to run the solution) in the Solution Test Runner. The use of Docker guarantees the security of the system because all the solutions are run in an isolated environment with a preallocated amount of resources.

2.4 Runtime view

In this chapter are presented all the runtime views associated with the use cases described in the RASD relative to the CKB platform. Runtime views show the interactions between the various components to carry out the functionalities offered by the platform.

Login User

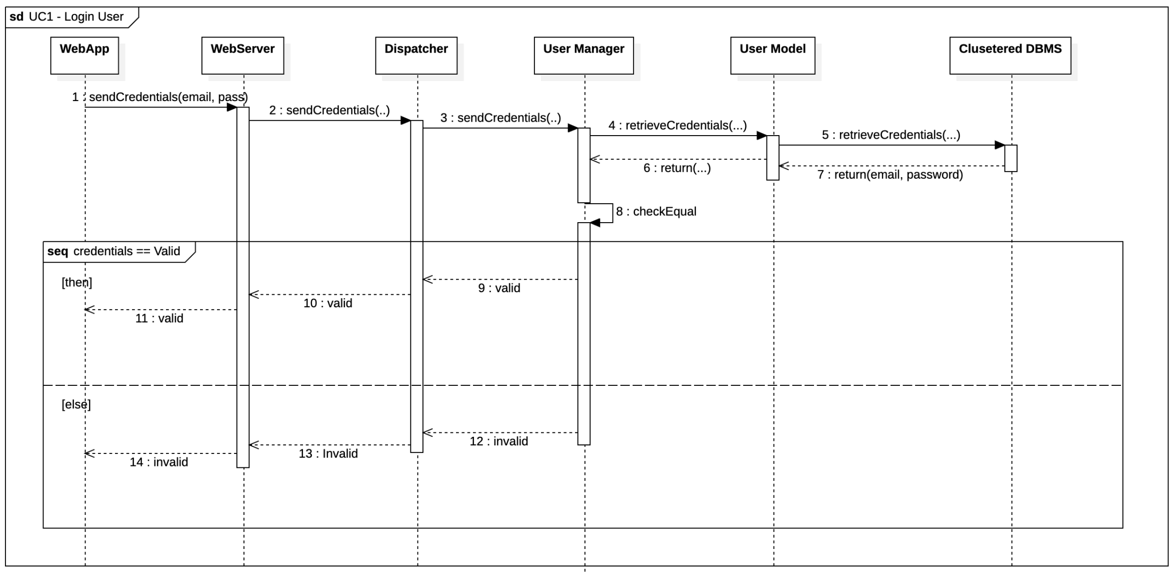
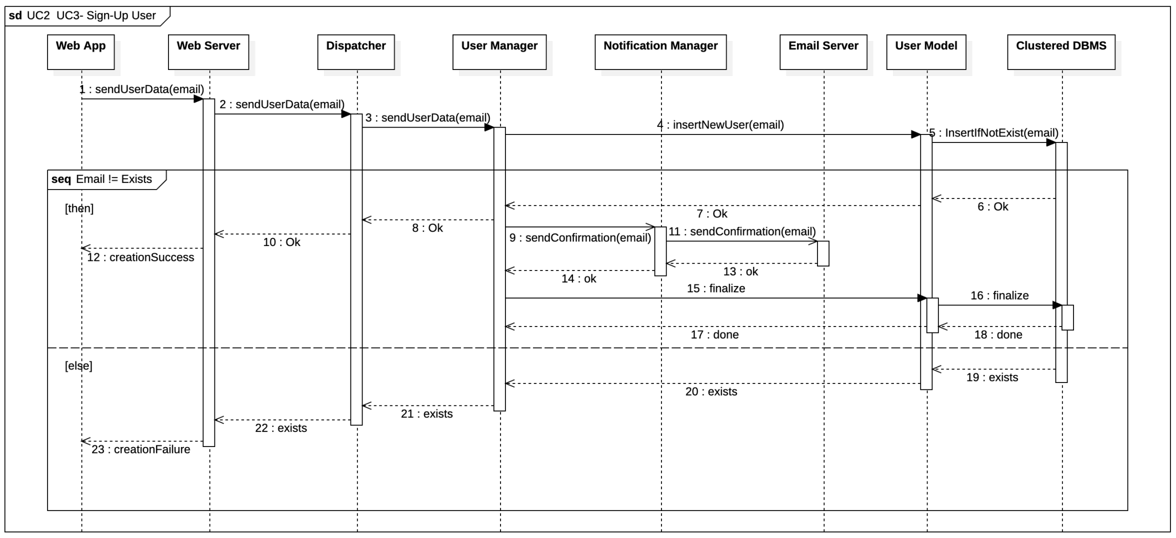


Figure 5: Runtime view: Login User

The user about to login into the application will enter the access credentials(email and password). Those credentials will be sent directly to the **User Manager**, which will interact with the database through the **User Model** to retrieve the credentials associated with the user, in order to check them. The user is enabled to log in if and only if the credentials correspond.

User sign-up



During sign-up time, the users will provide their email addresses to the system inside a form for registration. The system will initialize a process for the registration of the new user that will be finalized whenever the confirmation email is successfully answered. After that, the procedure to register the user is finalized.

User confirms the email address

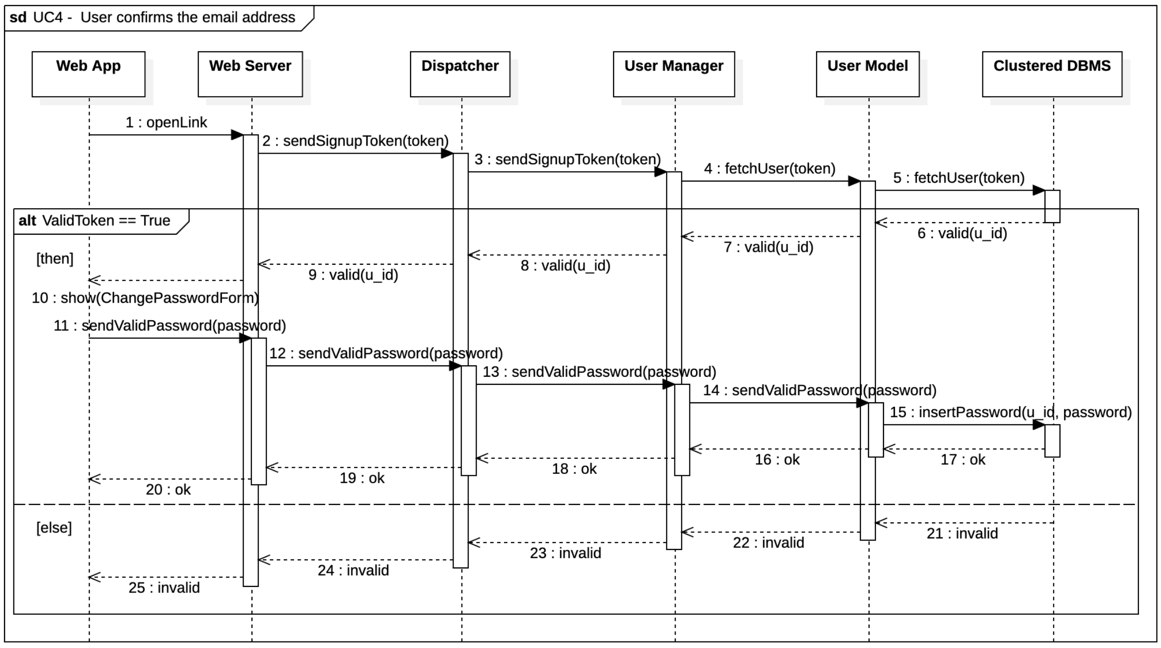


Figure 6: Runtime view: User confirms the email address

The confirmation of the email address happens through a confirmation token that is used by the user to initially identify the user. By using such a token, the User is enabled to set up a password, whose criteria are illustrated by the **Web App**. After setting up the password, the registration procedure is completed.

Tournament creation

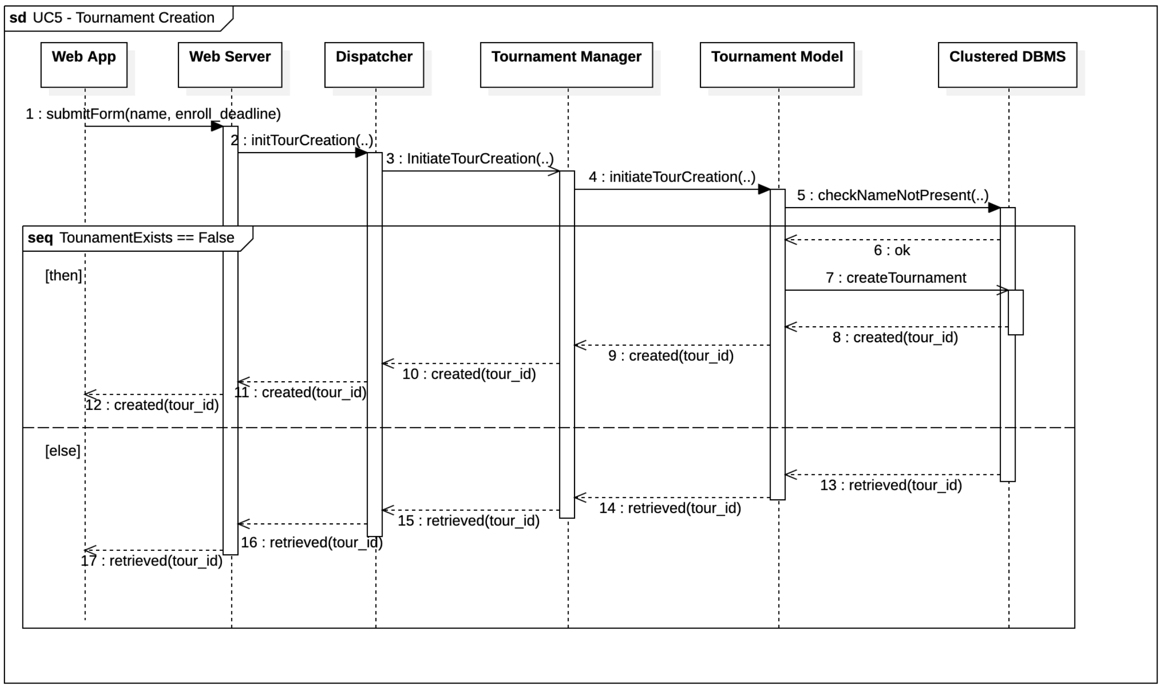


Figure 7: Runtime view: Tournament creation

A tournament creation is initiated by the **Web App** interface by an instructor. Only such type of user is enabled to present a tournament and therefore has a way to perform such action. The tournament presentation is done by specifying a unique name and an enrollment deadline.

Close tournament

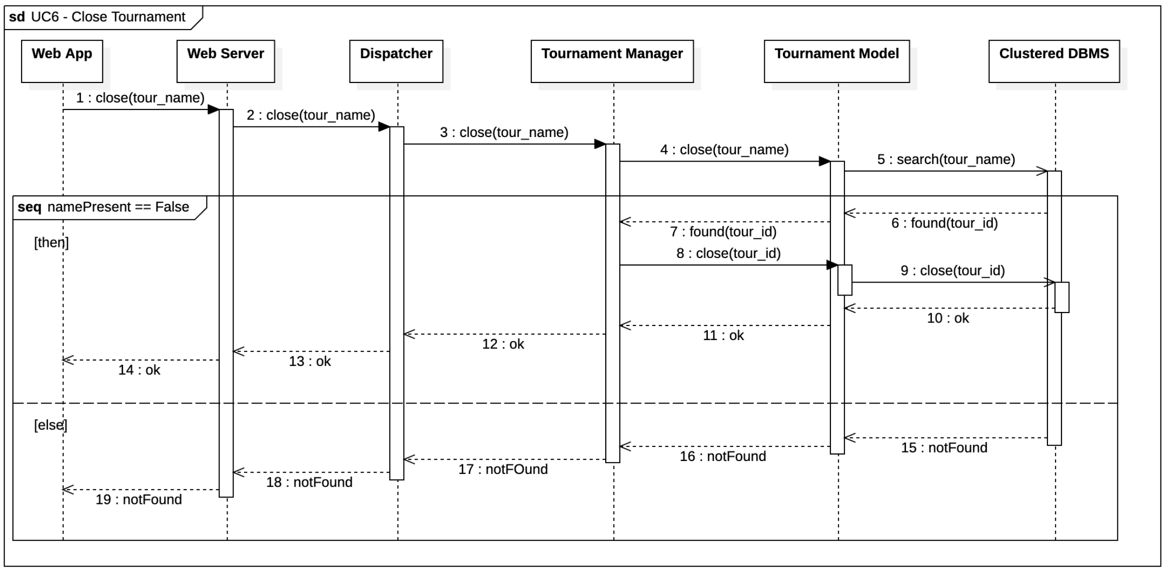


Figure 8: Runtime view: Close tournament

Similarly to the tournament creation, whenever a tournament must be closed, it will be done so by an instructor by specifying an existing name for such a tournament. The name will be then used to retrieve the tournament ID, uniquely identifying the tournament object.

Battle creation

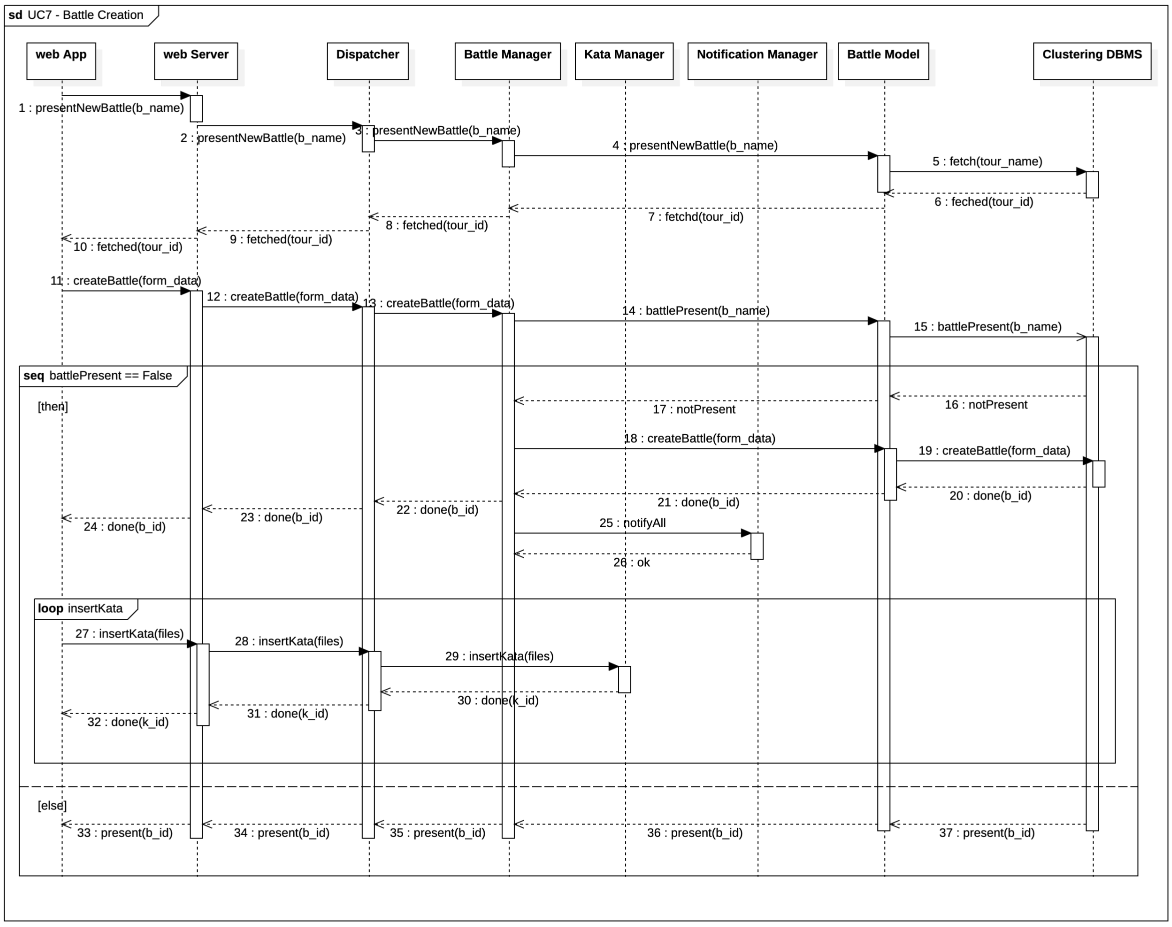


Figure 9: Runtime view: Battle creation

Whenever a new Battle is created by an instructor, he/she will use the **Web App** interface to insert the data of the battle. A battle is uniquely identified by its name and is to be referred to the most recent tournament still open associated with the instructor. The tournament ID is initially fetched by the application, and then the battle metadata are presented to the system. If the battle is not already present, the system will notify all the students enrolled in the tournament about the creation of the battle through the **Notification Manager** and will enable the instructor(s) to present new katas.

Battle close

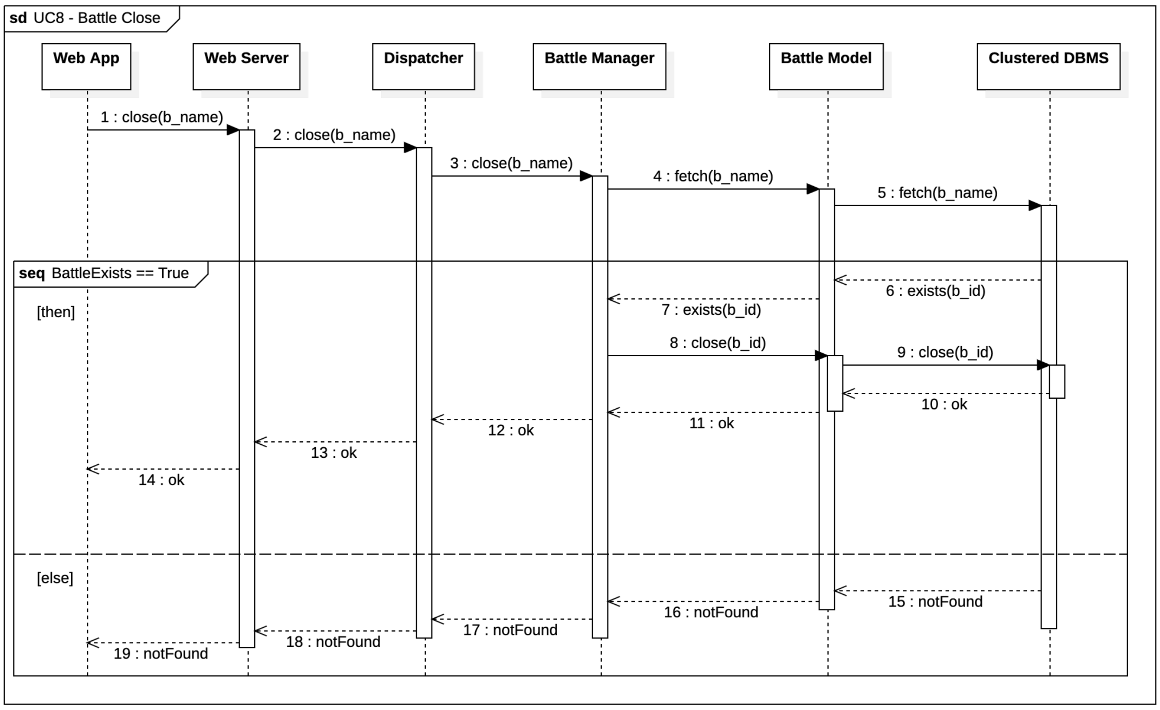


Figure 10: Runtime view: Battle close

Closing a battle is very similar to closing a tournament. A request is transmitted through the layers until it reaches the **Battle Manager** that will handle it. It does so by fetching the unique object identifier of the battle through the name and then forwarding the request to the database.

Join a Battle

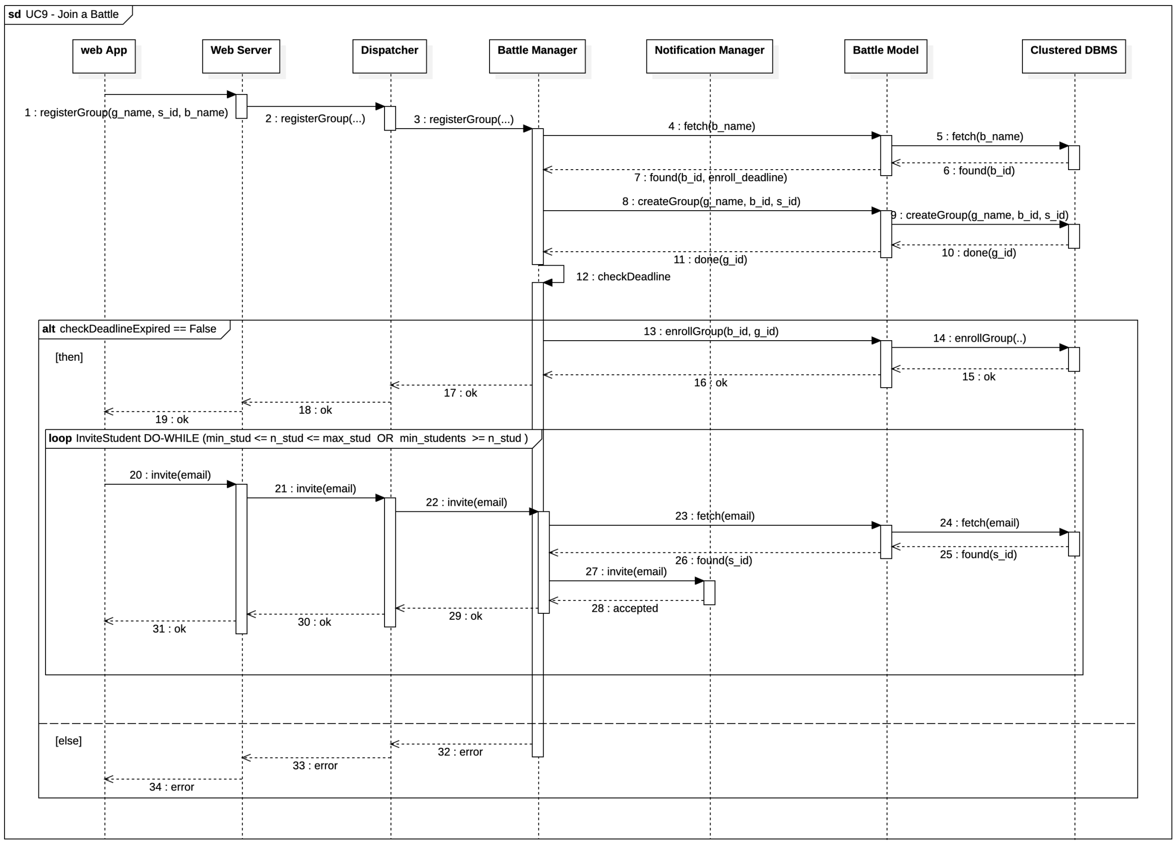


Figure 11: Runtime view: Join a Battle

Students will join a battle in groups and will do so by trusting their team leader to present the group to the application. She/he will do so by sending a request through the **Web App** to the **Battle Manager** specifying the name of the group and the name of the battle they want to join. If the request is performed before the deadline, the system will allow the students to create the group. It will be then the Team Leader to invite his or her Team Members through the application and the **Notification Manager**.

Students Commit a solution on GitHub

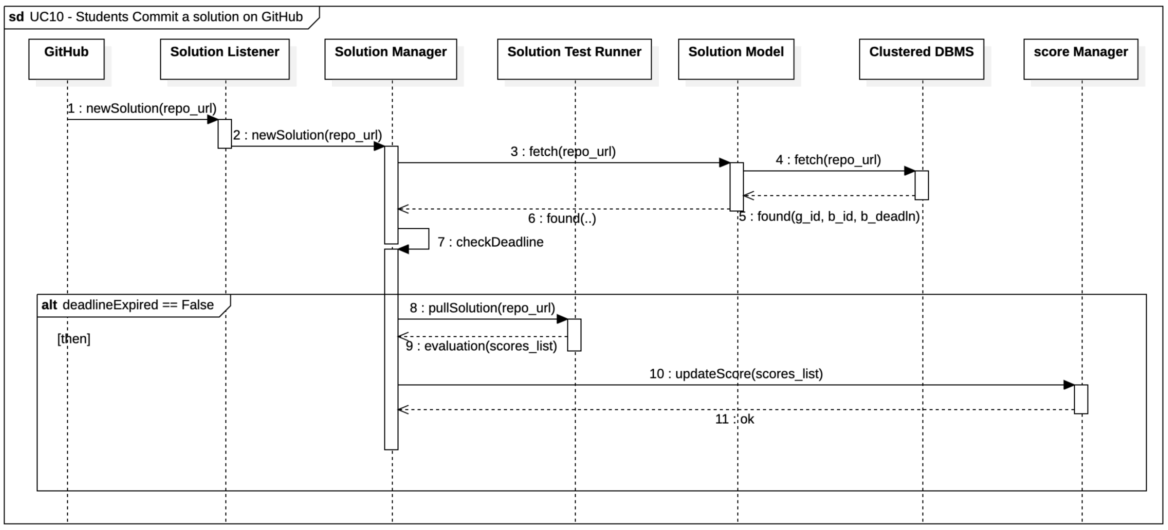

sd UC10 - Students Commit a solution on GitHub
GitHub
Solution Listener
Solution Manager
Solution Test Runner
Solution Model
Clustered DBMS
score Manager
1 : newSolution(repo_url)
2 : newSolution(repo_url)
3 : fetch(repo_url)
4 : fetch(repo_url)
5 : found(g_id, b_id, b_deadln)
6 : found(..)
7 : checkDeadline
alt deadlineExpired == False
[then]
8 : pullSolution(repo_url)
9 : evaluation(scores_list)
10 : updateScore(scores_list)
11 : ok

Figure 12: Runtime view: Students Commit a solution on GitHub

Whenever a push is performed on the GitHub repository, the system will be informed by the proper GitHub Action associated with it, which will inform the **Solution Manager** that will fetch the metadata associated with the repository and will use it to make a request to the **Solution Test Runner** system that will pull the just pushed code from GitHub, will test it and finally will provide the score. Having done so, the scores will be sent to the **Score Manager** for update.

2.5 Component interfaces

This section lists all the methods that each component interface provides to the other components.

1. **WebInterface:**
    - sendCredentials(email, password)
    - sendUserData(email)
    - openLink()
    - show(changePasswordForm)
    - sendValidPassword(password)
    - submitForm(name, enrollment-deadline)
    - created(tourid)
    - retrieved(tourid)
    - close(tournament-name)
    - presentNewBattle(bname)
    - fetched(tour-id)
    - createBattle(formdata)
    - done(bid)
    - present(bid)
    - insertKata(files)
    - done(kid)
    - close(bname)
    - registerGroup(gname, sid, bname)
    - invite(email)
2. **DispatcherInt:**
    - sendCredentials(email, password)
    - sendUserData(email)
    - sendSignupToken(token)
    - valid(uid)
    - sendValidPassword(password)
    - initiateTourCreation(name, enrollment-deadline)
    - created(tourid)
    - retrieved(tourid)
    - close(tournament-name)
    - presentNewBattle(bname)
    - fetched(tour-id)

   - createBattle(formdata)
   - done(bid)
   - present(bid)
   - insertKata(files)
   - done(kid)
   - close(bname)
   - registerGroup(gname, sid, bname)
   - invite(email)

3. **UserInt:**
   - sendCredentials(email, password)
   - checkEqual(email, password)
   - sendUserData(email)
   - sendSignupToken(token)
   - valid(uid)
   - sendValidPassword(password)
4. **TourInt:**
   - initiateTourCreation(name, enrollment-deadline)
   - created(tourid)
   - retrieved(tourid)
   - close(tournament-name)
5. **BattleInt:**
   - presentNewBattle(bname)
   - fetched(tour-id)
   - createBattle(formdata)
   - done(bid)
   - present(bid)
   - close(bname)
   - registerGroup(gname, sid, bname)
   - invite(email)
6. **KataInt:**
   - insertKata(files)
   - done(kid)
7. **ScoreInt:**
   - updateScore(scores-list)

8. **SolutionInterface:**
    - newSolution(repo-url)
    - checkDeadline()
9. **UserModelInt:**
    - sendCredentials(email, password)
    - return(email, password)
    - insertNewUser(email)
    - finalize()
    - fetchUser(token)
    - valid(uid)
    - sendValidPassword(password)
10. **TourModelInt:**
    - initiateTourCreation(name, enrollment-deadline)
    - created(tourid)
    - retrieved(tourid)
    - close(tournament-name)
    - close(tourid)
    - found(tourid)
11. **BattleModelInt:**
    - presentNewBattle(bname)
    - fetched(tour-id)
    - battlePresent(bname)
    - createBattle(formdata)
    - done(bid)
    - present(bid)
    - fetch(bname)
    - close(bid)
    - exists(bid)
    - createGroup(gname, sid, bname)
    - enrollGroup(gid, bid)
    - fetch(bname)
    - found(bid, enroll-deadline)
    - done(gid)
    - checkDeadline()

- fetch(email)
- found(sid)
- fetch(repo-url)
- found(gid, bid, bdeadline)

12. **KataModelInt:**
    - insertKata(files)
    - done(kid)
13. **ScoreModelInt:**
    - updateScore(scores-list)
14. **SolutionModelInt:**
    - newSolution(repo-url)
    - fetch(repo-url)
    - found(gid, bid, bdeadline)
15. **Model:**
    - retrieveCredentials(email, password)
    - return(email, password)
    - insertIfNotExist(email)
    - fetchUser(token)
    - valid(uid)
    - insertPassword(uid, password)
    - checkNameNotPresent(name, enroll-deadline)
    - createTournament()
    - created(tourid)
    - retrieved(trouid)
    - close(tourid)
    - found(tourid)
    - fetch(tour-name)
    - fetched(tour-id)
    - battlePresent(bname)
    - createBattle(formdata)
    - done(bid)
    - present(bid)
    - fetch(bname)
    - close(bid)

- exists(bid)
- createGroup(gname, sid, bname)
- fetch(bname)
- found(bid, enroll-deadline)
- done(gid)
- enrollGroup(gid, bid)
- fetch(email)
- found(sid)

16. **EmailProviderInt:**
    - sendConfirmation(email)
17. **GitHubSolutionInt:**
    - newSolution(repo-url)
18. **NotificationInterface:**
    - sendConfirmation(email)
    - notifyAll()
    - invite(email)
19. **SolutionTestInt:**
    - pullSolution(repo-url)
    - evaluation(scores-list)

2.6 Selected architectural styles and patterns

2.6.1 3-tier Architecture

The CKB platform is developed over a three-tier architecture. Each tier has different roles and divides the platform into three different independent layers:

- The **Presentation tier** is the top-level tier that is responsible for providing the user interfaces. Its main task is to deliver to the platform's users the data received from the Business Login tier and present them.
- The **Business Logic tier** is the middle-level tier, it is between the presentation and data tiers. It contains all the logic of the application and is responsible for providing all the core features of the platform. Furthermore, it passes and processes data between the two surrounding layers.
- The **Data tier** includes the Database system and provides an API, to the business logic tier, for accessing and managing the data stored in the DB.

Adopting this architecture ensures higher flexibility because thanks to the independence of the three layers, each layer can be updated apart from the others. Futhermore, by using a middle layer between the user (presentation layer) and the data layer, the security is improved because users cannot access directly to the data but all the requests to the DB should be made by the business logic tier that manages data authorization access.

2.6.2 Model View Controller (MVC) pattern

The CKB platform is developed by using the Model-View-Controller (MVC) pattern which is composed of three components:

- The **Model** which contains the application's data and provides methods for its manipulation.
- The **View** that contains all the web pages to be rendered with the users' data. This component provides all the methods that the controller will use to return the web pages to the users with dynamic content retrieved from the model.
- The **Controller** is between the Model and View. This component manages the interaction flow of the users, so upon a user's event, it makes some operations that can include data manipulation operations through the model component and at the end returns a different web page by using the view component.

The MVC pattern is used by a lot of web frameworks such as Spring or Django. This pattern increases the decoupling of the components which offers a series of benefits such as reusability, and simplicity of implementation.

2.6.3 Facade pattern

The facade pattern is used in the implementation of the Dispatcher component to hide the complexity of the application server from the web application and to provide the latter with a simpler interface. Furthermore, this solution increases the decoupling of the various components of our system; in particular between the web application (served by the web server) and the application server.

2.6.4 Mediator pattern

The mediator pattern is a behavioral pattern used to reduce coupling between modules willing to communicate with each other. The mediator component sits in between two or more objects and encapsulates how such objects communicate; this way does not require them to know implementation details about each other. The use of this pattern improves the flexibility of development it is possible to change or update the mediator component without the need to change all the other components. In our system, the **NotificationManager** is a mediator component, because it connects the email service used to send emails with the other components of the application server by providing an interface to send emails to the CKB users. If the external email service is to be changed, it will only be necessary to update the mediator component to implement the API of the new chosen service without, however, changing all the components that need to use the service.

2.7 Other design decisions

In this section are presented some of the design decisions made for the system to make it work as expected.

2.7.1 Sandboxes for solutions' testing

To test the solutions provided by students, it is necessary to build and run the solution. To ensure system security, it is essential to run solutions within sandboxes, isolated environments with a pre-allocated amount of resources that are created on demand when a solution is provided for evaluation. The use of sandboxes prevents potentially malicious software from affecting the entire application server and prevents the exploitation of system resources. The Sandboxes can be implemented using Docker[1], a platform that allows the creation of on-demand containers that contain the required tools to analyze the students' solutions and the provided solution. Furthermore, by using Docker it is possible to introduce a job queue, which is essential to manage a large amount of requests. The SolutionManager triggers the creation of a container by using the Docker APIs, the container will test the solution by building and running the provided code and will be destroyed right after the testing of the solution is terminated.

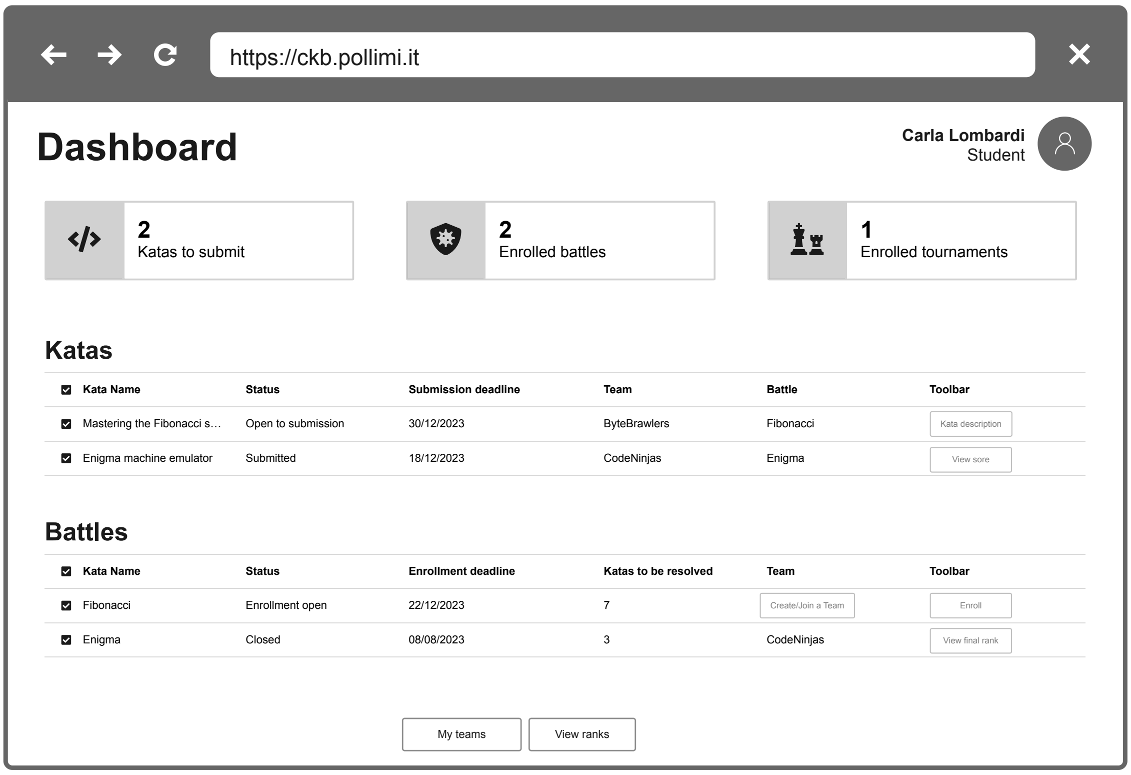
2.7.2 GitHub Action task for solutions' notifications

To notify when a new version of a Kata's solution is committed, the system exposes a REST API that the GitHub Action service can call. When students commit to the solution, the previously configured GitHub Action task will be triggered and will make a GET request to the CKB platform's API exposed by the Solution Listener. The SolutionListener will then pull the new version of the code and send it to the SolutionManager which will trigger the solution. The chosen design will move the task of notifying the application server when the student commits to GitHub, avoiding the Application Server to check regularly the version of the monitored GitHub repositories.

[1]Official website: www.docker.com

3 User interface design

This section of the document presents the Web platform UI: the views both the students and the instructors will see when they login to the platform. The student dashboard (fig 13) gives an overview of the activities related to the CKB: the student can look at which Battles is enrolled, and can enroll in a battle. Furthermore, he/she can have an overview of the Katas he/she should submit and view the scores of the previously submitted ones. The instructor dashboard (fig 14) gives the instructors an overview of the battles and tournaments created inside the CKB platform. The instructor can create a Battle/Tournament using the "create battle" form (fig 15), in which he/she can specify all the parameters of the battle such as the minimum number of students per team or the tournament in which the battle is part. The instructor, for each battle created, should upload at least one kata.



https://ckb.pollimi.it

Dashboard

Carla Lombardi
Student

2
Katas to submit

2
Enrolled battles

1
Enrolled tournaments

Katas

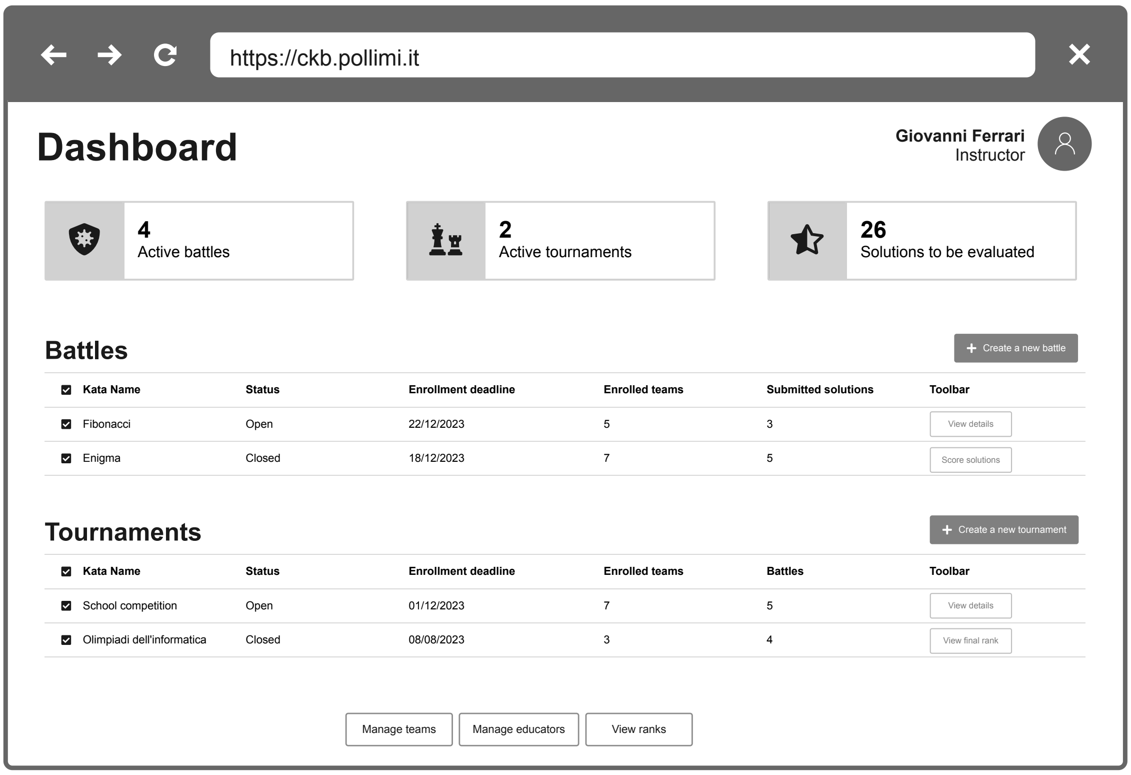
	Kata Name	Status	Submission deadline	Team	Battle	Toolbar
✓	Mastering the Fibonacci s…	Open to submission	30/12/2023	ByteBrawlers	Fibonacci	Kata description
✓	Enigma machine emulator	Submitted	18/12/2023	CodeNinjas	Enigma	View sore

Battles

	Kata Name	Status	Enrollment deadline	Katas to be resolved	Team	Toolbar
✓	Fibonacci	Enrollment open	22/12/2023	7	Create/Join a Team	Enroll
✓	Enigma	Closed	08/08/2023	3	CodeNinjas	View final rank

My teams | View ranks

Figure 13: User interface: Student dashboard



https://ckb.pollimi.it

Dashboard

Giovanni Ferrari
Instructor

4
Active battles

2
Active tournaments

26
Solutions to be evaluated

Battles

Create a new battle

	Kata Name	Status	Enrollment deadline	Enrolled teams	Submitted solutions	Toolbar
✓	Fibonacci	Open	22/12/2023	5	3	View details
✓	Enigma	Closed	18/12/2023	7	5	Score solutions

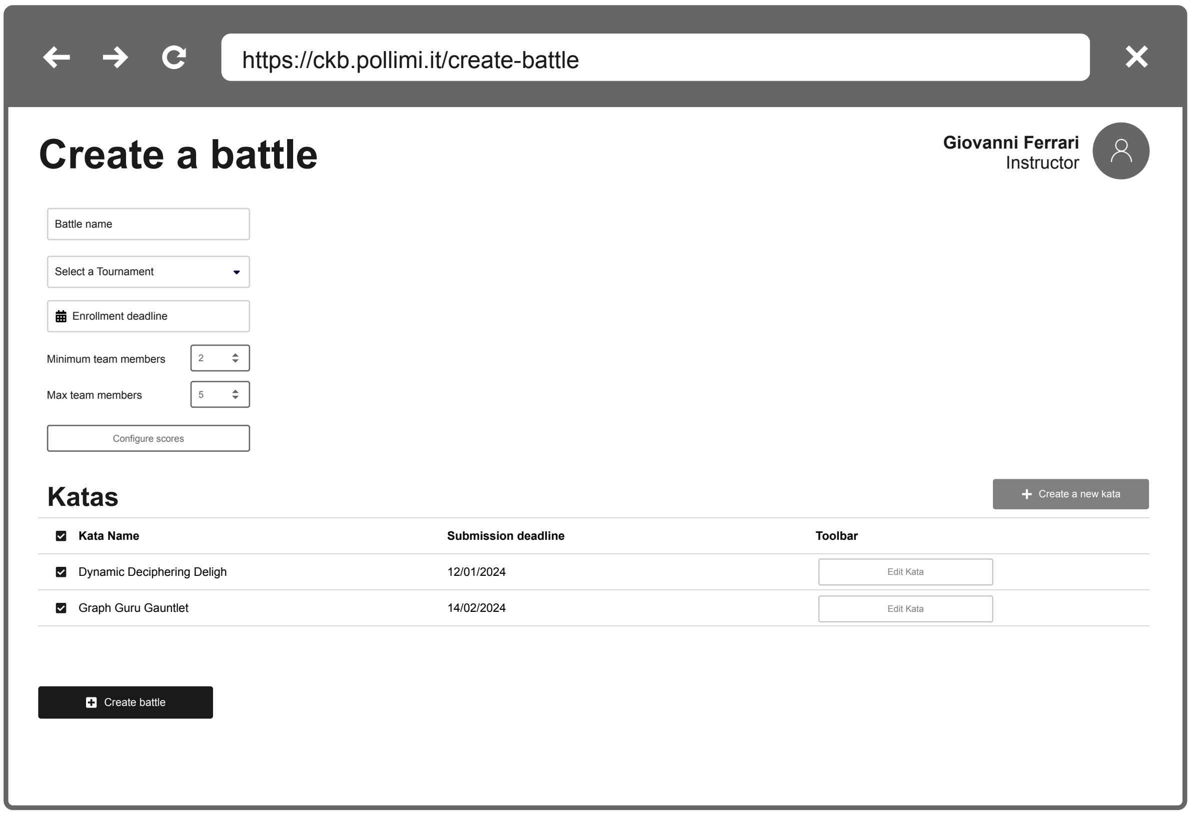
Tournaments

Create a new tournament

	Kata Name	Status	Enrollment deadline	Enrolled teams	Battles	Toolbar
✓	School competition	Open	01/12/2023	7	5	View details
✓	Olimpiadi dell'informatica	Closed	08/08/2023	3	4	View final rank

Manage teams | Manage educators | View ranks

Figure 14: User interface: Instructor dashboard



https://ckb.pollimi.it/create-battle

Create a battle

Giovanni Ferrari
Instructor

Battle name

Select a Tournament

Enrollment deadline

Minimum team members	2
Max team members	5

Configure scores

Katas

Create a new kata

✓	Kata Name	Submission deadline	Toolbar
✓	Dynamic Deciphering Deligh	12/01/2024	Edit Kata
✓	Graph Guru Gauntlet	14/02/2024	Edit Kata

Create battle

Figure 15: User interface: Instructor interface to create a battle

4 Requirements traceability

In this section of the document it is explained, for each requirement defined in the RASD, what design elements are involved for its fulfillment. In the following a series of tables maps such design elements to the respective requirement.

Requirements	R1 The system allows students to sign up. R2 The system allows instructors to sign up. R3 The system allows students to log in. R4 The system allows instructors to log in.
Components	• WebApp • Web Server • Application Server: – Dispatcher – UserManager – UserModel – NotificationManager • Email provider • Clustered DBMS

Requirements	R5 The system allows instructors to create a tournament. R6 The system allows the instructors to insert the deadline for enrollment in the tournament. R7 The system allows the instructors to insert the name for the tournament. R8 The system allows instructors to close a tournament. R12 The system allows instructors to give access to peers to the tournament management.
Components	• WebApp • Web Server • Application Server. • – Dispatcher – TournamentModel – TournamentManager – NotificationManager • Email provider • Clustered DBMSTest

Requirements	R9 The system allows instructors to create a battle in a tournament. R10 The system allows instructors to manage a battle in a tournament. R11 The system allows students to form teams. R14 The system creates a new repository for each created battle. R16 The system allows students to FORK the provided repository. R21 The system allows students to join battles.
Components	• WebApp • Web Server • Application Server: – Dispatcher – BattleModel – BattleManager – NotificationManager • GitHub • Email provider • Clustered DBMS

Requirements	R13 The system provides visibility of tournament ranks to all platform users. R15 The system automatically evaluates submissions based on functional aspects, timeliness, and code quality. R18 The system automatically updates students' rank based on the number of correct solutions provided and the manual evaluation. R20 The system allows instructors to score students' solutions manually.
Components	• WebApp • Web Server • Application Server: – Dispatcher – SolutionManager – SolutionModel – ScoreManager – ScoreModel • Clustered DBMS • SolutionTestRunner

Requirements	R17 The system allows students to provide solutions to katas within the deadline through COMMIT actions. R19 The system allows instructors to provide test cases in different programming languages to katas. R22 The system is informed every time a PUSH action is performed on a repository within the deadline of the battle. R23 The system performs a PULL action every time a PUSH has occurred

Components	• WebApp
	• Web Server
	• Application Server:
	    – Dispatcher
	    – SolutionManager
	    – SolutionModel
	    – SolutionListener
	    – ScoreManager
	    – ScoreModel
	    – KataManager
	    – KataModel
	• Clustered DBMS
	• SolutionTestRunner
	• GitHub

5 Implementation, integration and test plan

5.1 Overview

"The bitterness of poor quality remains long after the sweetness of low price is forgotten." - **Benjamin Franklin_**

With this famous quotation, we introduce this important chapter.

The Integration, Implementation, and Test Plan chapter is a pivotal section within the Design Document, guiding the software development process from design to deployment. It outlines the integration strategy, determining how diverse components converge into a cohesive system. Implementation details are discussed, covering programming languages, coding standards, and version control. The Test Plan section addresses quality assurance through various testing methodologies. Deployment considerations and monitoring strategies are also detailed, emphasizing the importance of meticulous planning for a smooth transition to the operational phase and continuous improvement post-deployment. This chapter encapsulates the essence of transforming a conceptual design into a functional, robust software system.

5.2 Implementation plan

The system will undergo implementation through a bottom-up approach, the features of the system are strongly associated with the components identified in the Component Diagram, therefore is not necessary to undergo a thread approach to testing The bottom-up strategy supports incremental integration, easing bug tracking through the testing of intermediate results—sub-systems formed by component integration—as additional modules are incorporated and is to be preferred to a top-down approach, because allows more resilience to possible changes in requirements during the development phase.

The bottom-up strategy involves identifying system features and the corresponding components responsible for delivering those features. Since different components collaborate to achieve a single feature, it becomes essential to establish an implementation order.

This implementation strategy enables the assignment of different features to independent development teams operating in parallel. However, before this, identifying common components is crucial to prevent redundant production of the same component.

Notice that the majority of the platform's features require communication between the application server and the web server, therefore a portion of the Dispatcher has to be developed for each one of them. In that sense, a thread strategy is used to develop the dispatcher component since a portion of features is added at each implementation test.

5.2.1 Features identification

The features to implement in the system are extracted directly from the system requirements. It is important to notice that some of them require the implementation of entire components, while others require the implementation of more or less small portions of them. In the following, there is a brief recap of the features of the system.

F1. **Sign-in and sign-up** Instructor and Student sign-in and sign-up processes are similar and can be implemented together. The only difference is in the email domain to check at the moment of registration and the authorization that different types of users have inside the CKB platform.

F2. **Tournament, Battle and Kata creation and management** All these entities concur in the core features of the system and must be developed together because strictly related to one another. They can be divided into multiple sub-features that can be developed separately: Tournament creation and close, tournament management, enrollment to a tournament, battle creation and close, battle management, enrollment to a battle, team creation and management, and kata creation.

F3. **Send notification to users** This feature is strictly connected to the email provider of choice and should be created as an API wrapper over the external service.

F4. **Automatically run tests on students' solutions** This feature requires the development of a system that enables the automatic testing of the students' solutions. To do so, it is important to integrate a resource manager and a container system that can create a container when a new solution is provided and then run it in an isolated environment. It is important to choose the right infrastructure (for instance Docker for managing the containers) and then create an image that can be used to create the isolated environment which encapsulates all the tools required for testing the solutions.

F5. **Automatically collects students' solutions** The development of this feature must follow the implementation of the Solution Test Runner because it is a system whose aim is to inform the latter for it to perform pull requests. The Solution Listener will expose the API used in GitHub Actions to inform the CKB platform each time a solution is committed.

F6. **Automatically assign scores to the students** This feature requires the development of two components: a tool inside the isolated environment that based on the provided solution retrieves the execution parameters such as the used resources, the time of execution, and the output of the solution; and the ScoreManager that analyses the parameters and then computes the score. Moreover, the ScoreManager allows the instructors to give manual scores and users to view the updated tournament ranks.

5.3 Component Integration and Testing

In the following section, for each step is specified which components are implemented in the stage, and how the components are integrated and tested (there is an integration diagram for each stage).

For simplicity, and to avoid monolithic graphs, we have decided to split the implementation diagrams based on the feature they refer to, even though a bottom-up approach to development, as the one we have selected, may want the module grounds implemented together, with the support of drivers.

The white folders represent the components, while the drivers are the pieces of software needed for the scaffolding of the infrastructure while it is being implemented. Usually, drivers would be removed when upper components were implemented, but for the simplicity of the graph and since the components may be dependent on multiple other upper components, we have decided to maintain them.

[F 1] Sign-in and sign-up

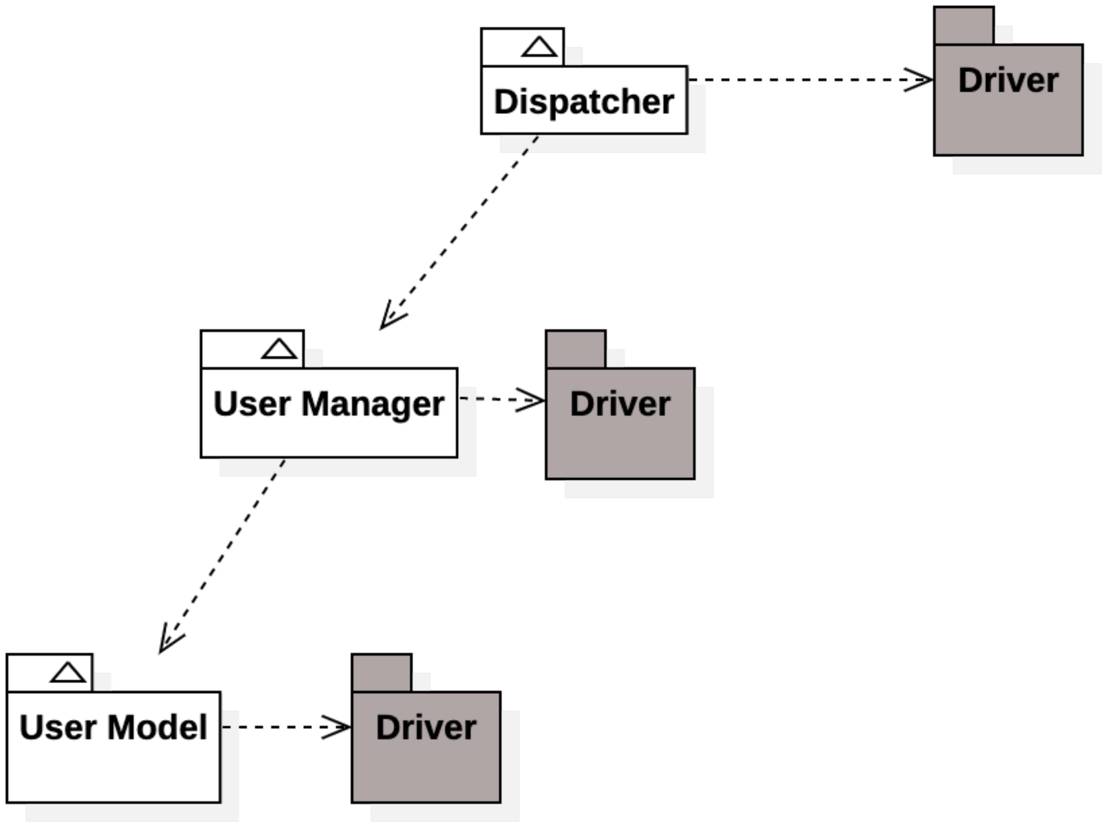


Figure 16: Integration Diagram: Sign-in and sign-up

The following graph shows the implementation strategy for the following feature. The User Model and relative drivers are implemented first, then the User Manager and finally the corresponding thread in the dispatcher.

[F 2] Tournament, Battle and Kata creation and management

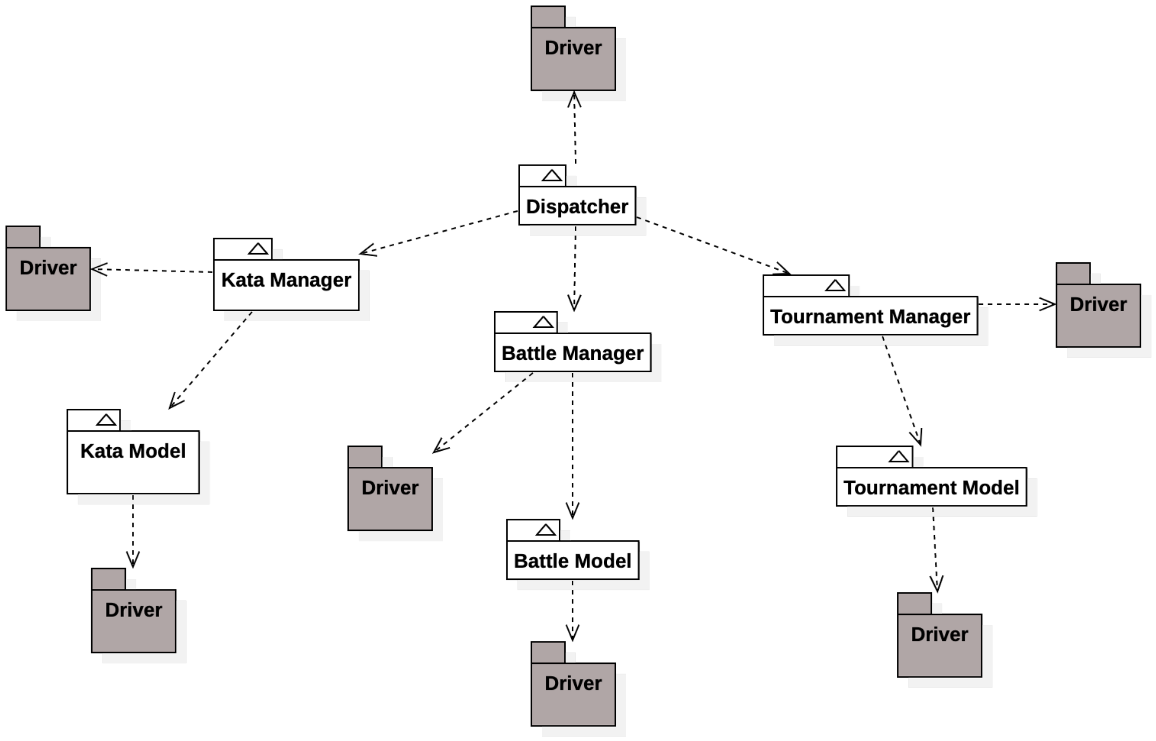


Figure 17: Integration Diagram: Tournament, Battle and Kata creation and management

The following graph shows the implementation strategy for the following feature. The Models and relative drivers are implemented first, then the Managers, and finally the corresponding thread in the dispatcher.

[F 3] Send notification to users

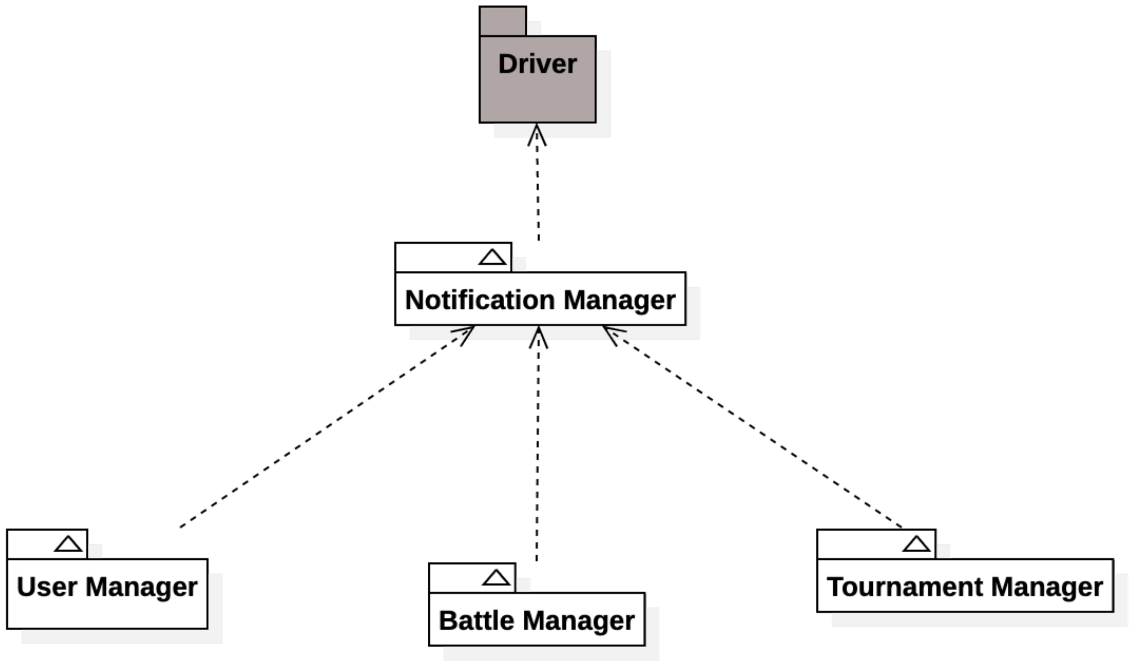


Figure 18: Integration Diagram: Send notification to users

The following graph shows the implementation strategy for the following feature. The Notification Manager is implemented when the User, Battle, and TournamentManager are implemented.

[F 4] Automatically run tests on students' solutions

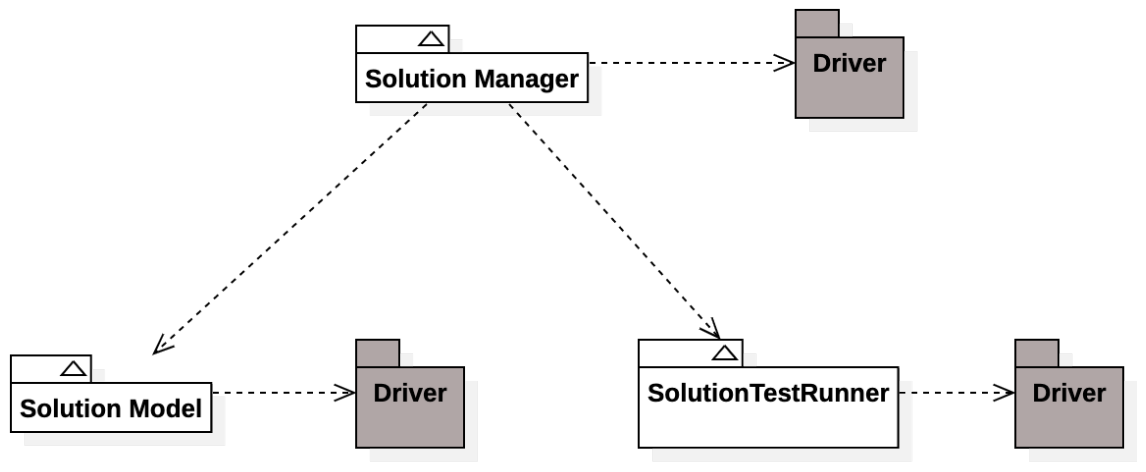


Figure 19: Integration Diagram: Automatically run tests on students' solutions

The following graph shows the implementation strategy for the following feature. The Solution Manager depends on both the Solution Model and the Solution Test Runner which are the first to be implemented.

[F 5] Automatically collects students' solutions

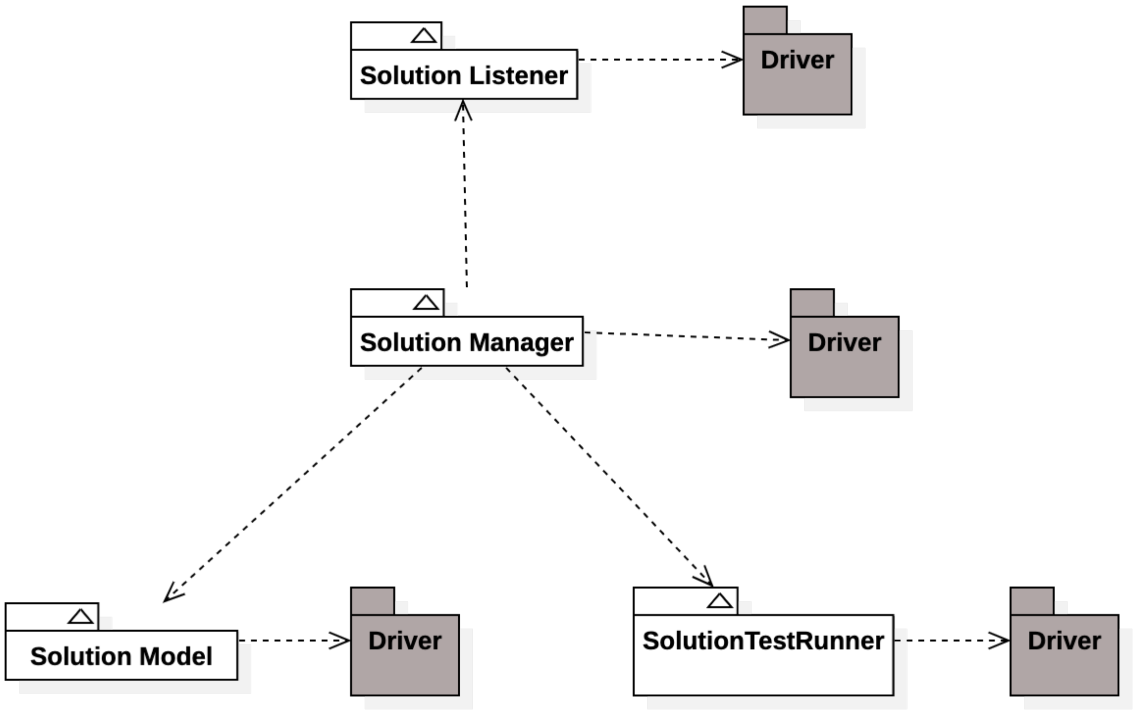


Figure 20: Integration Diagram: Automatically collects students' solutions

The Solution Listener implementation comes immediately after the implementation of the manager.

[F 6] Automatically assign scores to the students

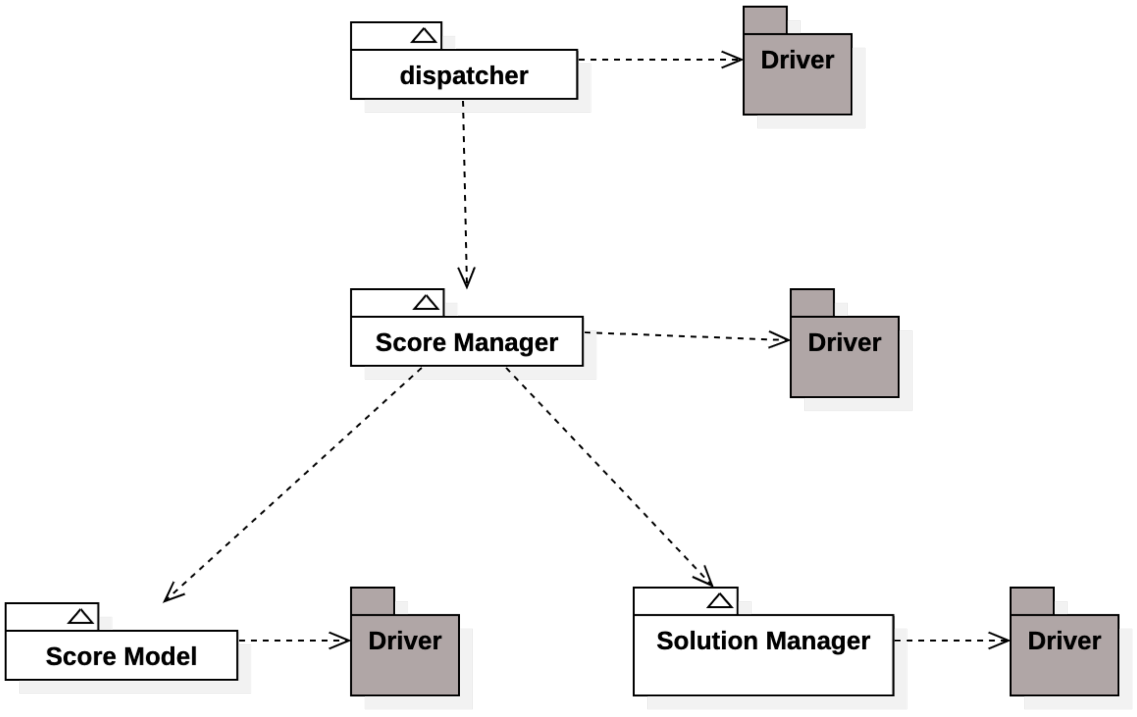


Figure 21: Integration Diagram: Automatically assign scores to the students

Score Manager uses both the Score Model and the Solution Manager, so its implementation must follow the implementation of the two. Finally, the corresponding thread in the dispatcher is created.

5.4 System testing

The CKB platform, before the production development, undergoes a series of testing activities to test the entire system. To do so, it is necessary to plan different tests that have different scopes and granularity (some can test one component, and others can test a feature that requires more components for instance). During the development, first of all, every single component has to be tested on its own to check if it is performing as expected. Since some components require other components to work properly (for instance the UserManager requires the NotifcationManager component to send users the confirmation link to complete the sign-up), to test components in isolation is required to develop Driver components that simulate the behavior of the missing required components. Thus, the component can be tested in isolation from the system and it will be easier to find bugs.

After performing individual component testing, the next step involves integrating and testing the combined outcome. In our scenario, we use a bottom-up integration and testing strategy, as described in the previous chapter.

Once the system is implemented, unit tested, and integration tested, it is required to test the entire system to check if all the features have been developed correctly and if the system and the outcome comply with the functional and non-functional requirements defined in the RASD document. To verify that, it is necessary to do system testing. At this final stage, the software must be as close to the final product as possible. This testing phase doesn't involve only the developers but can include also stakeholders who can evaluate the final product. The system testing is composed of the following steps:

- **Functional testing**: consists of verifying whether the software meets the functional requirements specified in the Requirement Analysis and Specification document (RASD). In this test, the software is used as described by use cases in the RASD to check whether requirements are fulfilled.
- **Performance testing**: this testing step aims to find any bottlenecks (affecting response time, utilization, throughput), inefficient algorithms, and hardware/network issues to improve the performance of the whole system. In a broader sense, it entails identifying inefficiencies that could impact the system's performance. This process requires a predefined workload and a set performance target for testing purposes.
- **Load testing**: this testing phase aims to expose bugs such as memory leaks, mismanagement of memory, and buffer overflows. It also identifies the upper limits of the components. This phase can also be used to evaluate the distributed architecture and its scalability. To do so, the system is tested at increasing workload until it can support it.
- **Stress testing**: To make sure that the system recovers gracefully after failure. To test the latter, it is necessary to try to break the system under test by overwhelming its resources or by reducing resources (for instance, randomly shutting down and restarting ports on the network switches/routers that connect the servers).

5.5 Additional specifications on testing

The system has to be tested whenever a new function is added to the CKB platform, to check the presence of bugs. All the steps described in the paragraph 5.4 should be performed to ensure that the updated component will work as expected inside the system. It is also crucial to continuously gather feedback from the stakeholders because regular feedback can be useful for discovering usability issues and bugs that previous tests haven't found. Some users will be invited to join the alpha version of the system, a version that contains new features and is in the final testing phase, to test the new features and provide feedback regarding discovered bugs and the general usability of the app. This will introduce a continuous validation of the system during the creation process and will allow developers to promptly evaluate the quality of the system and make necessary adjustments if certain aspects of its implementation are not aligned with customer expectations.

6 Effort Spent

Here we report an estimation of the hours spent in the creation of the DD document.

name	**hours**
Alberto Eusebio	20h
Marcello Martini	20h

Table 6: Effort spent in the creation of the document

References

- High-level system overview made with draw.io
- Diagrams made with StarUML
- Mockups made with Miro