



UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

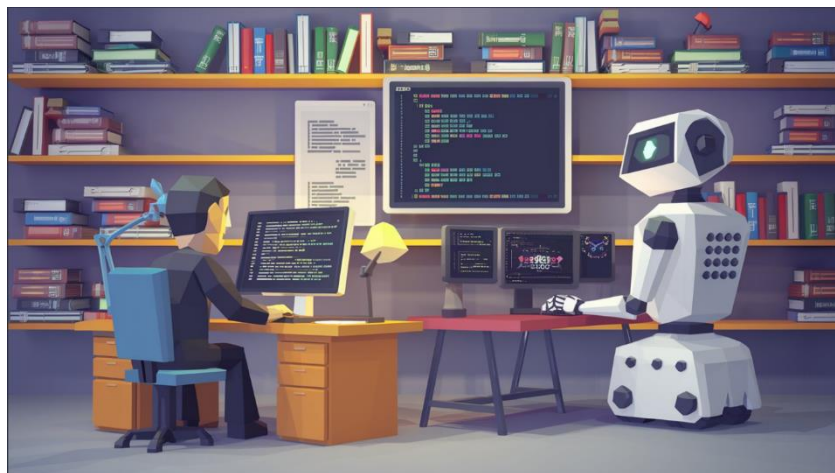
Scuola Politecnica e delle Scienze di base

Corso di Laurea Magistrale in Ingegneria Informatica

SAD Development Documentation:

Analysis and Design

Task T2 - Group 6





Software Architecture Design Course

Teacher: Anna Rita Fasolino

Academic year: 2024-2025

Github Repository: <https://github.com/uo288764/A13>

Members:

-  Marçal Muñoz Salat (m.muñozsalat@studenti.unina.it) Matricola: N. 0001/24345
-  Carlos Sampedro Menéndez (c.sampedromenendez@studenti.unina.it) Matricola: N. 0001/24345
-  Andrés Carballo Pérez (a.carballoperez@gmail.com) Matricola: N. 0001/24363
-  Shreya Malviya (s.malviya@studenti.unina.it) Matricola N. 0001/24300

Date: 07/01/2025

INDEX

| | |
|--|-----------|
| 1 Introduction | 4 |
| 1.1 Description of the project..... | 4 |
| 1.2 Development methodologies | 4 |
| 1.2.1 Weekly work..... | 4 |
| 1.2.2 Sprint backlog..... | 4 |
| 1.2.3 Used tools | 5 |
| 2. Domain Analysis | 6 |
| 2.1 Preliminary Analysis | 6 |
| 2.2 Requirement Specification | 6 |
| 2.2.1 Functional Requirements..... | 6 |
| 2.2.2 Non-Functional Requirements | 6 |
| 3 Project documentation..... | 7 |
| 3.2 Technologies and Development Environment | 7 |
| 3.2.1 Spring Boot..... | 7 |
| 3.2.2 Thymeleaf | 7 |
| 3.2.3 Bootstrap | 7 |
| 3.2.4 Maven | 8 |
| 3.2.5 Extensions – Spring Boot Tools and Spring Boot Dashboard | 8 |
| 3.3 Project architecture | 8 |
| 3.3.1 Class diagram | 9 |
| 3.4 External Database Integration | 11 |
| 3.5 Integration T on T5 | 12 |
| 3.5.1 CORS restrictions | 12 |
| 3.6 Visual Design | 13 |
| 3.7 Update of the DB after playing a game..... | 14 |
| 4. Analysis of the Impact of the Requirements on the Existing Project | 14 |
| 4. 1 Description of the Affected Architectural Components and the Impact of the Modifications: | 14 |
| 5 Design Solutions to Satisfy the Requisites | 16 |
| 5.1 Design Decisions Made | 16 |
| 5.2 Description of new or modified interfaces (e.g., REST APIs with usage examples). | 16 |
| 5.3 Changes in the database (updated data model). | 18 |
| 5.4 Problems identified and resolved during development. | 19 |
| 6. Final Deployment Diagram | 20 |
| 7. Description of Tests Performed | 21 |

| | |
|--|-----------|
| 7.1 Solved issues | 21 |
| 7.2 Solved Issues | 21 |
| 7.2.1 Saving new games | 21 |
| 8. Future Developments | 22 |
| 8.1 Resolve the issue of game completion | 22 |
| 8.2 Validation and Debugging of LeaderboardService | 22 |
| 8.3 Improvement in Backend-Frontend Communication..... | 22 |
| 8.4 Review the API Configuration in T5 | 22 |
| 8.5 Real-Time Monitoring..... | 22 |

Working Methodology

1 Introduction

1.1 Description of the project

Once the player has logged into the app, they have the option to view their statistics in the relevant tab. This shows a ranking of all the people who have logged in and played the game, displaying various parameters and giving players a rank in the overall ranking.

The initial version given does not store the data of the game in any place. There is no calls to database in order to store the actual state of the ended game (whether the user has won or not) neither a total count of the history of games the user has played.

1.2 Development methodologies

1.2.1 Weekly work

The first step was to understand and analyse the domain of the problem to be solved, as well as to identify the difficulties that could arise during the execution of the task. To carry out this analysis process, meetings were organised at a frequency of 1 or 2 times a week, in order to give a first organisational structure to the team.

The meetings were held in two modalities: face-to-face and remote. Face-to-face meetings were preferred to be reserved for discussing the main problems and finding solutions to the most complex parts of the work, while remote meetings were used to take stock of the situation and evaluate the progress of the project.

1.2.2 Sprint backlog

The organizational meetings were key to defining the sprint backlog, initially dividing tasks among team members, and setting deadlines for each objective, based on the review dates set by the professors.

In fact, the deadlines were established on a biweekly basis during the period from November to December. However, as the project presentation date approached, it

was decided to schedule more weekly meetings with the aim of intensifying the work and increasing productivity, focusing on refining the final deliverables.

1.2.3 Used tools

GitHub

To efficiently share code and carry out incremental development divided into multiple stages, the GitHub hosting service was used. This platform facilitated teamwork on the code and optimized the application of the pair programming technique.

Visual Paradigm

The Visual Paradigm modeling tool was useful during the design phase, helping the team define requirements and use cases. It also facilitated the creation of various UML diagrams, which are included in the corresponding sections of the document.

Visual Studio Code

It is worth highlighting the use of the Visual Studio Code editor, which was used by the entire team to write and compile the code. The team took advantage of various extensions that facilitated obtaining the desired result, highlighting the integration of the Spring Boot framework through the “Spring Boot Dashboard” extension, which allowed them to meet the requirements defined in the initial stages. In addition, the GitHub extension integrated into the editor allowed the different development branches to be coordinated directly from the code-editing environment.



```
17 import lombok.AllArgsConstructor;
18 import org.springframework.web.bind.annotation.RequestParam;
19 import org.springframework.web.bind.annotation.PutMapping;
20 import org.springframework.web.bind.annotation.PathVariable;
21
22
23 @RestController
24 @RequestMapping("/api/user")
25 @AllArgsConstructor
26 public class UserController {
27     private final UserService userService;
28
29     @PostMapping("/scores")
30     public ResponseEntity<String> postScores(@RequestBody UserDto userDto) {
31         try {
32             String result = userService.postScores(userDto);
33             return ResponseEntity.ok(result); // Return HTTP 200 with the success message
34         } catch (Exception e) {
35             return ResponseEntity.badRequest().body("Error: " + e.getMessage());
36         }
37     }
38
39     @GetMapping("/getAllResult")
40     public ResponseEntity<List<User>> getMethodName() {
41         try {
42             List<User> users = userService.getAllScores();
43             return ResponseEntity.ok(users);
44         } catch (Exception err) {
45             return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
46         }
47     }
48
49     @PutMapping("/updatescore")
50     public ResponseEntity<User> updateUser(
51         @RequestParam String email,
52         @RequestParam boolean win) {
53         try {
54             User user = userService.updateUserScore(email, win);
55             return ResponseEntity.ok(user);
56         } catch (Exception err) {
57             System.err.println("Exception: " + err.getMessage());
58             return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
59         }
60     }
61
62
63 }
```

2. Domain Analysis

Domain Analysis This chapter focuses on understanding the problem domain and identifying requirements that will guide the development of the application. A proper analysis of the domain ensures that the system is structured efficiently and meets user expectations and project goals.

This analysis is key to ensuring that the system components are well-defined and capable of meeting the established objectives for project development.

2.1 Preliminary Analysis

Problem Definition and Scope The domain analysis began by identifying key challenges and defining the scope of the project. The primary objective was to create a system that allows users to log in, play a game, and view their game history and rankings. It was crucial to address the lack of data persistence in the initial version of the application. Therefore, storing game results and user statistics in a database became a top priority.

To gain a better understanding of the scope, meetings were held with the product owner to clarify expectations and gather feedback. The product owner provided a base architecture, which served as a guideline for structuring the new components of the system. Key actors were identified, including players, system administrators, and external services that might interact with the application.

2.2 Requirement Specification

Requirement specification requirements were specified based on the problem analysis, user needs, and stakeholder inputs. These requirements were categorized into functional and non-functional requirements.

2.2.1 Functional Requirements

Functional Requirements The functional requirements define the core features of the system:

- **FR1:** Users must be able to register and create an account.
- **FR2:** Users must be able to log in using their credentials.
- **FR3:** The system must allow users to play a game session and display the result (win/loss).
- **FR4:** Users must be able to view their game history, including details of each session played.
- **FR5:** The system must maintain a leaderboard, ranking users based on their performance.
- **FR6:** Administrators must be able to manage users and game data.
- **FR7:** Users must have the option to reset their password in case they forget it.

2.2.2 Non-Functional Requirements

Non-Functional Requirements The non-functional requirements describe the quality attributes of the system:

- **NFR1: Scalability:** The system must support a growing number of users without performance degradation.
- **NFR2: Security:** User data, including credentials, must be securely stored using encryption techniques.
- **NFR3: Usability:** The user interface must be intuitive and responsive, providing a seamless experience across devices.
- **NFR4: Data Integrity:** The system must ensure data consistency and prevent data loss in case of system failure.
- **NFR5: Performance:** The system must have a response time of less than 2 seconds for 95% of user requests.

3 Project documentation

3.2 Technologies and Development Environment

As mentioned in section 1.2.3, the Visual Studio Code editor was used to write and compile the code. Although originally a lightweight code editor, the team utilized it as a development environment by adding extensions to integrate frameworks and other technologies, turning it into a suitable IDE.

3.2.1 Spring Boot

Spring Boot is an open-source framework based on Java that simplifies the development of Java applications. It provides a pre-configured infrastructure that reduces development complexity, allowing developers to focus on core business logic. Spring Boot offers features such as dependency management, auto-configuration, embedded web servers, and tools for development and testing. It is widely used to create scalable, robust, and easy-to-maintain Java applications.

3.2.2 Thymeleaf

Thymeleaf is an open-source template engine for Java that allows the creation of dynamic HTML pages. It is designed to integrate with Java-based web applications, such as those developed with Spring Boot. Thymeleaf supports the creation of HTML templates by adding expressions and special attributes, enabling dynamic page content. It allows binding Java model data to HTML elements, facilitating the generation of interactive web pages. Additionally, it offers advanced features such as support for internationalization, layout management, and form handling, making it a popular choice for building user interfaces in Java web applications.

3.2.3 Bootstrap

Bootstrap is an open-source front-end framework that simplifies the creation of responsive and intuitive websites. It provides a wide collection of pre-defined components such as buttons, forms, navigation bars, and a grid system for layout design, allowing developers to quickly build modern and well-designed interfaces. Bootstrap uses HTML, CSS, and JavaScript to offer consistent tools and styles that facilitate the development of adaptive websites that automatically adjust to different devices and screen sizes. It is highly customizable and offers numerous themes and additional plugins to extend its functionality.

3.2.4 Maven

Maven is a Java project management tool that simplifies both dependency management and the build process. Its integration into the project was highly beneficial, as it allows developers to declare the required libraries in the “pom.xml” configuration file, automatically downloading them. Additionally, Maven promotes a standardized project structure, which facilitates collaboration among team members. With its predefined lifecycle, the application can be easily compiled, tested, and packaged. Moreover, Maven supports plugins that extend its functionality, such as code quality control and automatic documentation generation.

3.2.5 Extensions – Spring Boot Tools and Spring Boot Dashboard

The Spring Boot Tools and Spring Boot Dashboard extensions provide useful tools that enhance and simplify the development of Spring Boot applications within Visual Studio Code, enabling developers to work more efficiently.

In particular, Spring Boot Tools offers features such as code auto completion, static analysis, debugging, and Spring Boot-specific dependency management. It also allows applications to be executed directly from Visual Studio Code, streamlining the development and testing process.

On the other hand, Spring Boot Dashboard allows developers to visualize and manage running Spring Boot applications, both locally and on remote servers. It provides details about the status of applications, enabling start, stop, and restart operations, as well as log viewing and performance metrics monitoring.

3.3 Project architecture

The architecture of the developed application is based on the Model-View-Controller (MVC) structural pattern. This choice was made considering the advantages that this pattern offers for structuring web applications, using Spring Boot as the main framework.

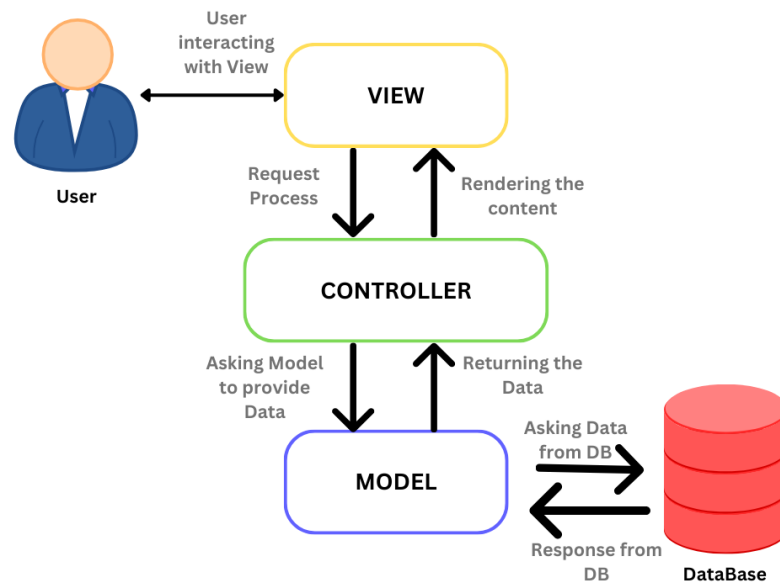
The MVC pattern separates the application’s responsibilities into three main components:

- Model: Responsible for data management and application logic.
- View: Provides the user interface, enabling data visualization and user input collection.
- Controller: Coordinates user interactions, manages control logic, and updates both the model and the view as needed.

Additionally, the MVC pattern promotes application scalability and is widely compatible with major web development frameworks and technologies, offering a rich ecosystem of tools and resources that simplify the development process.

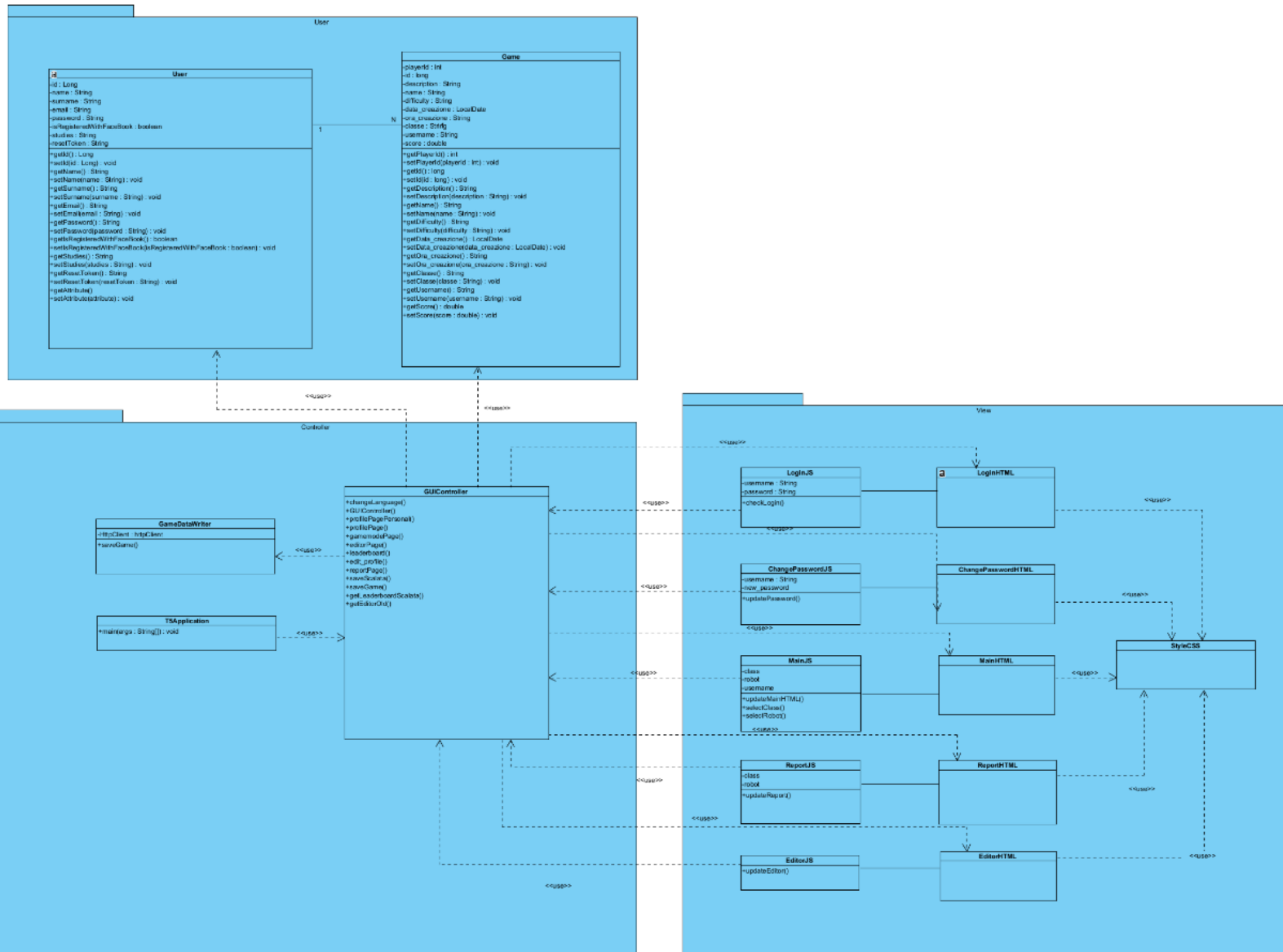
The MVC pattern was chosen over the BCE pattern (Boundaries, Controllers, Entities) because the latter reduces flexibility in managing user interfaces and increases

maintenance complexity. Modifications or extensions in software developed with BCE require a deep understanding of interactions between its components. Moreover, the concept of Entities does not fit well with the data management approach used in the project (T5), and the need to implement an efficient user interface made MVC the most suitable option.



3.3.1 Class diagram

The class diagram provides a structured visual representation of the system's classes and the relationships between them. It plays a key role in defining the software architecture, facilitating both its development and maintenance, and serves as an effective means of communication between developers and stakeholders. The diagram details the properties and methods of each class, highlighting their main responsibilities. This enables a clear understanding of how different classes interact and provides a solid foundation for system development. In the next image it is represented the main class diagram of the project:



The diagram consists of four packages:

- **Model:** Contains the classes that represent the system's data and business logic.
 - **Player:** Designed as a singleton class, ensuring there is only one instance of the player throughout the system.
 - **Game:** Maintains a 1:N relationship with the Player class, allowing a player to be associated with multiple games.

- View: This package contains the classes responsible for the user interface, including the necessary JS, HTML, and CSS components for the front-end.
- Controller: Manages the control logic and the interactions between the view and the model.
 - GUIController: Handles the different pages of the application through various functions and coordinates the interaction between data files and model objects.

The T5application class serves as the main entry point of the application. Through its main method, it starts the GUIController, which manages the application's pages through <<use>> relationships with other classes.

This design ensures a clear separation of responsibilities, facilitates system maintenance, and enables adequate scalability.

3.4 External Database Integration

A new folder named 'T' was created to integrate an external MySQL database hosted on Aiven.io. This integration involved configuring the application to connect to the MySQL database by specifying the necessary connection properties, such as the database URL, username, and password. This setup enabled the application to interact seamlessly with the external database, facilitating efficient data management and retrieval.

APIs Developed in Task T

To enhance functionality, three main APIs were developed:

1. Retrieve User Scores:

- Endpoint GET /api/user/getallResult
- Provided the capability to fetch all results, offering a comprehensive overview of the data stored within the system.

```
@GetMapping("/getallResult")
public ResponseEntity<List<User>> getMethodName() {
    try {
        List<User> users = userService.getAllScores();
        return ResponseEntity.ok(users);
    } catch (Exception err) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

2. Update User Scores:

- Endpoint: PUT /api/user/updatescore
- Enabled the updating of user scores based on specific parameters, such as the user's email and game outcome, thereby allowing for dynamic adjustments to user data.

3. Post New Scores:

- Endpoint: POST /api/user/scores
- Designed to retrieve user scores, allowing clients to access the performance metrics of individual users

These developments significantly improved the application's data handling capabilities, ensuring robust interactions with the MySQL database and providing essential endpoints for data retrieval and manipulation.

3.5 Integration T on T5

The integration between T and T5 was pivotal in enabling a dynamic and user-friendly leaderboard. This integration not only ensured efficient communication between the two system components but also allowed for the creation of a functional interface for the user. In T5, the frontend components were responsible for managing the user interface and interactions, while the logic and data management were delegated to T. This was achieved by APIs exposed by T, as we have mentioned above, which provided the necessary endpoints to retrieve, display, and update the leaderboard data efficiently.

From a technical standpoint, the frontend of T5 interacted with the APIs of T via Axios, a popular JavaScript library for making HTTP requests. Axios facilitated communication between the client and server by providing a simple, modern interface for making asynchronous requests (such as GET, POST, PUT, DELETE) to the API endpoints.

We can see how this function in T5 uses Axios to asynchronously fetch leaderboard data from T. This function makes a GET request to the endpoint to retrieve the data. Once the data is received, it is logged for verification and returned for further processing.

```
const fetchLeaderboardData = async () => {  
  try {  
    const response = await axios.get('http://localhost:8080/api/user/getallResult');  
    console.log(response.data); // Verifica los datos  
    return response.data;  
  } catch (error) {  
    console.error('Error fetching leaderboard data:', error);  
    return [];  
  }  
};
```

3.5.1 CORS restrictions

During the development of the integration between T and T5, one of the challenges we encountered was related to CORS (Cross-Origin Resource Sharing). CORS is a security mechanism implemented by browsers to prevent unauthorized access to resources from a different origin. In simpler terms, it restricts web applications running on one domain (origin) from making requests to another domain unless explicitly permitted by

the server. This restriction exists to protect users and their data from malicious cross-origin attacks.

In our case, since the frontend (T5) and the backend (T) were hosted on different origins, the browser blocked requests from T5 to access the APIs exposed by T. These CORS issues initially disrupted communication between the two components, preventing seamless data retrieval and updates.

Challenges such as CORS restrictions were resolved by implementing the `@CrossOrigin` annotation in Task T's controllers. This annotation explicitly allowed requests from the frontend's origin, enabling smooth communication between the backend and frontend hosted on different origins.

```
@CrossOrigin(origins = "http://localhost")
@GetMapping("/getAllResult")
public ResponseEntity<List<User>> getMethodName() {
    try {
        List<User> users = userService.getAllScores();
        return ResponseEntity.ok(users);
    } catch (Exception err) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

3.6 Visual Design

The leaderboard interface was designed with careful attention to detail, leveraging the Bootstrap framework to create a responsive and user-friendly layout. The page's structure included intuitive components such as a search bar, pagination controls, and options for selecting the number of rows displayed per page, ensuring that users could easily navigate through the data.

To further enhance interactivity, the search functionality allowed users to filter players by email, dynamically showing results as they typed. Pagination and the ability to

| RANK ↑ | EMAIL | GAMES WON | GAMES PLAYED | WIN RATE |
|--------|--------------------|-----------|--------------|----------|
| 1 | marshall@gmail.com | 19 | 28 | 67.86% |
| 2 | shreya@gmail.com | 7 | 12 | 58.33% |
| 3 | carlos@gmail.com | 5 | 8 | 62.5% |
| 4 | carba@gmail.com | 0 | 0 | 0% |
| 5 | carba@gmail.com | 0 | 0 | 0% |
| 6 | player1@gmail.com | 0 | 0 | 0% |
| 7 | player2@gmail.com | 0 | 0 | 0% |
| 8 | player3@gmail.com | 0 | 0 | 0% |
| 9 | player4@gmail.com | 0 | 0 | 0% |
| 10 | player5@gmail.com | 0 | 0 | 0% |

select rows per page were seamlessly integrated with JavaScript to ensure scalability for large datasets.

3.7 Update of the DB after playing a game

After various attempts to implement it in different ways, the issues encountered will be noted in the "Problems" section. These issues were primarily caused by the incorrect placement of T outside T5, resulting in the inability to call it from the GameController. We eventually decided to use a new service, along with a helper user, to call it within the gesticipartita method in the GameController.

The Leaderboard service is responsible for receiving and handling requests. In gesticipartita, the player ID is provided, and using T23, the user (T5) is retrieved, along with their email, name, surname, and the win Boolean. If the user who just completed the game is already in the database, the updateScore function is called. If not, the insertScore function is invoked, and a new user record is added to the database. This record includes the user's email, name, and surname, with initial values for games played, games won, win rate, and rank (set to 0 or a random number, since these values will be updated later by T).

Unfortunately, with this implementation, the game does not seem to finish. In a previous case with a similar implementation, the game completed successfully, and the server received a query like the following:

```
SELECT u1_0.id, u1_0.email, u1_0.first_name, u1_0.games_played, u1_0.games_rate,
u1_0.games_won, u1_0.last_name, u1_0.rank FROM newuser u1_0
```

For some unknown reason, the game does not finish in the current implementation, which could potentially be due to a conflict with T23.

4. Analysis of the Impact of the Requirements on the Existing Project

4. 1 Description of the Affected Architectural Components and the Impact of the Modifications:

The new requirements and modifications significantly impacted various architectural components of the project, requiring changes and the creation of new files. Below are the affected and newly introduced components:

1. Modified Files:

- **leaderboard.html:** Updated to display the leaderboard data from the database.

- **navbar.html:** Adjusted to integrate navigation links related to the leaderboard and ensure access to the Leaderboard button from Home page.
- **GameController (T5App):** Modified to include new logic for scanning and considering ExternalUserDto in
`@SpringBootApplication(scanBasePackages = {"com.g2.factory", "com...", "com.g2.dto"})`

2. Created Files:

- **dto.ExternalUserDto:** New Data Transfer Object to facilitate communication between services.
- **service.LeaderboardService.java:** Created to centralize and manage all HTTP requests related to the leaderboard, handling updates and new player entries.
- **Leaderboard.js:** JavaScript file responsible for fetching, processing, and dynamically displaying data from the leaderboard database.
- **Leaderboard.css:** Stylesheet specifically designed for the leaderboard interface to ensure clarity and visual consistency.

3. Created Files (All within the 'T' Folder):

- **UserController:** A **REST controller** that handles user-related API requests. It provides endpoints for posting scores, retrieving user results, updating scores, and fetching user details via email.
- **UserDto:** A **Data Transfer Object (DTO)** representing user data. It facilitates the transfer of structured user information, including email, rank, games played, and win rates, between client and server.
- **Request:** A **utility class** that contains static methods for mapping between UserDto and the User entity, ensuring consistency and smooth transformations.
- **User: Description:** Represents the **User entity** in the application, encapsulating user-related data such as ID, email, rank, and gameplay statistics.
- **UserRepo:** A **repository interface** extending JpaRepository, responsible for database operations involving user entities, including searching by email and performing CRUD operations.
- **UserImp:** Implements the **UserService** interface, providing the **business logic** for managing user data. It supports operations like saving scores, updating rankings, and retrieving user details.

- **UserService:** Defines the **core interface** for user management. It outlines essential methods for saving user scores, retrieving user rankings, and fetching user details by email.

5 Design Solutions to Satisfy the Requisites

5.1 Design Decisions Made

After attempting to use the **database** and the **T4Service interface**, among other components that belonged to the base application's architecture, and realizing they did not function as expected, we decided, **as a first step, to implement our own database.**

Our **initial idea was to use MongoDB**, but in the end, we opted for **Aiven**, which is an SQL-based database. Due to **difficulties during its integration**, we were unable to incorporate it directly into **T5**, and instead, we placed it in a **separate folder named "T"**.

Once the **API was implemented** to handle the database calls and its functionality was verified, we resumed work on the **leaderboard.html** file to process the database data. This was achieved with a **new JavaScript file named leaderboard.js** and a **dedicated stylesheet called leaderboard.css.**

After completing these steps, we attempted to ensure that **game results were saved at the end of each session.** After numerous attempts, the **final approach chosen** was to create a **LeaderboardService.** This service manages all **HTTP requests** to the database.

The **LeaderboardService** is called from the **GameController**, specifically from its **gesticipartita** function. When a game ends:

1. The service retrieves **user values** using the **T23Service.**
2. It either **updates the player's score** or **adds a new player to the database** if they are not already registered.

Unfortunately, **the latter scenario does not work for reasons still unknown to us.**

5.2 Description of new or modified interfaces (e.g., REST APIs with usage examples).

Taking as example the functioning of T, as it's the main change with regards to the original

1. **src/main/java/com/example/demo:** This package houses the core business logic and REST controller.
2. **controller:** This package contains the UserController class, responsible for handling REST requests related to users.
 - **UserController.java:** The class that handles REST requests.
 - **@RestController:** This annotation marks the class as a REST controller.
 - **@RequestMapping("/users"):** This annotation sets the base URL for all endpoints in this controller to /users.

- Methods with annotations:
 - **@PostMapping:** This annotation maps a POST request to the /users endpoint, typically for creating a new user.
 - **@GetMapping:** This annotation maps a GET request to the /users endpoint, typically for retrieving a list of users.
 - **@GetMapping("/{id}"): This annotation maps a GET request to /users/{id} endpoint, typically for retrieving a specific user by ID.**
 - **@PutMapping("/{id}"): This annotation maps a PUT request to /users/{id} endpoint, typically for updating an existing user.**
 - **@DeleteMapping("/{id}"): This annotation maps a DELETE request to /users/{id} endpoint, typically for deleting a specific user.**

3. dto: This package holds Data Transfer Objects (DTOs) used to exchange data between the controller and other layers.

- **ResponseDto.java:** This class represents the response structure when handling successful requests. It may contain data related to the user and status codes.
- **UserDto.java:** This class represents the structure of user data used for input or output in REST requests. It may include fields like username, email, password, etc.

4. mapper: This package contains mappers that transform data between different formats.

- **Request.java:** This class may represent the request body sent by the client, potentially containing user data to be processed.

5. model: This package holds the domain model classes representing the entities in the system.

- **User.java:** This class represents the User entity, including fields such as id, username, email, and any other relevant attributes.

6. service: This package contains service classes that encapsulate the business logic.

- **User Service.java:** This class provides methods for user-related operations, such as creating, retrieving, updating, and deleting users. It interacts with the repository layer to perform database operations.

7. repository: This package contains interfaces for data access.

- **User Repository.java:** This interface extends a Spring Data repository, providing methods for CRUD operations on the User entity without the need for boilerplate code.

8. configuration: This package may contain configuration classes for setting up various aspects of the application.

- **WebConfig.java:** This class may configure CORS settings, message converters, or other web-related configurations.

9. application.properties: This file contains configuration properties for the Spring Boot application, such as database connection settings, server port, and logging levels.

5.3 Changes in the database (updated data model).

After verifying that the original database in the project did not meet our requirements, we decided to implement our **own database solution**. Initially, our plan was to use **MongoDB**, but we ultimately opted for **Aiven**, an alternative that utilizes **SQL**.

The **database schema** was structured to store user-related data efficiently and includes the following attributes:

```
{
  "userId": 43,
  "email": " valentino.rossi@studenti.unina.it",
  "rank": 15,
  "firstName": "Valentino",
  "lastName": "Rossi",
  "gamesPlayed": 25,
  "gamesWon": 18,
  "gamesRate": 72.0
}
```

- **userId**: Unique identifier for each user.
- **email**: User's email address, serving as a unique reference.
- **rank**: The current rank of the user based on performance.
- **firstName**: User's first name.
- **lastName**: User's last name.
- **gamesPlayed**: Number of games the user has participated in.
- **gamesWon**: Number of games won by the user.
- **gamesRate**: Win rate percentage calculated based on games played and won.

This database is located within the **T/demo** directory. To start the database, the following command must be executed in **Visual Studio Code's terminal**: `mvn spring-boot:run`

Once the database is running, the following **API methods** from the **UserController** can be used to interact with it:

- **String postScores(UserDto userLoginDto) throws Exception;**
 - Adds or updates a user's score in the database.

- **List<User> getAllScores() throws Exception;**
 - Retrieves all user scores from the database.
- **User updateUserScore(String email, boolean win) throws Exception;**
 - Updates a user's score based on the outcome of a match (win/lose).
- **User getUserByEmail(String email);**
 - Fetches a user's details using their email address.

These methods are **defined in the UserService interface**, and their **logic is implemented in UserImp**.

Finally, you can interact with the database via **HTTP requests** using tools like **Postman**. For example, to verify that the service is running correctly, you can use the following endpoint:

GET <http://localhost:8080/api/user/getallResult>

This request will return a list of all user scores in JSON format, ensuring that the database and API are functioning as expected.

5.4 Problems identified and resolved during development.

During the development process, several challenges were identified and resolved to ensure smooth functionality across the system:

1. **Database Connectivity Issues:**
 - *Problem:* The original database was not connectable, causing failures in data retrieval and updates.
 - *Solution:* A new external database was created and successfully integrated into the application. This new database allowed proper data retrieval.
2. **Empty Leaderboard Display:**
 - *Problem:* The leaderboard, even when accessed, appeared empty despite having existing user data in the system.
 - *Solution:* The logic responsible for fetching and displaying leaderboard data was corrected. The integration with backend services was improved, ensuring that the leaderboard accurately displayed user data, including scores and rankings.
3. **lista_utenti impossible to add fields:**
 - *Problem:* The lista_utenti component was not functioning properly. It was to add any extra user data that was going to be needed for a correct implementation of the Leaderboard, such as score, games played, or games won.
 - *Solution:* To resolve this, the fetching of data was done from our DB was integrated into the leaderboard.js.
4. **T4Service methods:**

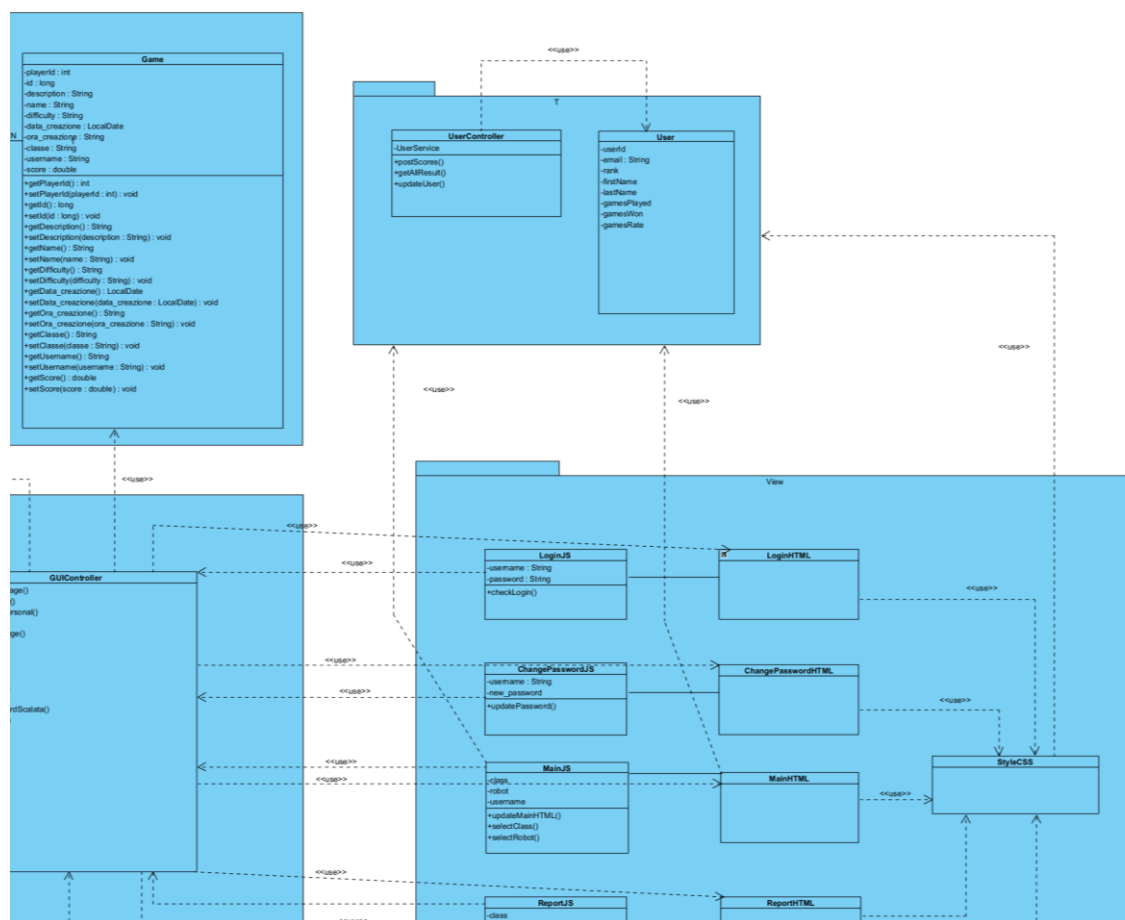
- *Problem:* At first, we tried using `getGames` however T4Service was not working.
- *Solution:* The user data, such as games played, wins, and other statistics, were stored in the new database.

5. Leaderboard Button Missing from Home Screen:

- *Problem:* The leaderboard button was missing from the home screen, which prevented users from accessing the leaderboard.
- *Solution:* editing navbar to make it visible on home.

6. Final Deployment Diagram

The image illustrates the final deployment diagram of the application, highlighting the key components and their interactions. It includes classes for managing users (User) and games (Game) in the system, with detailed attributes and methods. We can see the T package apart from the others that were already created, that is the one that interacts with the new database to store the statistics of each player.



7. Description of Tests Performed

During the development and integration of the new requirements, various tests were carried out to ensure the proper functioning of the modified and newly created components. Below are the most relevant identified issues and their current status:

7.1 Solved issues

1. Navbar not showing the Leaderboard button:
 - Issue: Initially, the navigation button to access the Leaderboard was not displaying correctly.
 - Solution: The navbar.html file was updated to ensure the Leaderboard link was present and accessible from the home page.
2. Leaderboard appearing empty:
 - Issue: When accessing leaderboard.html, data was not being loaded correctly.
 - Solution: leaderboard.js was implemented to make API calls and fetch data from the database, which was then dynamically rendered in the HTML.
3. Database not functioning correctly:
 - Issue: Initially, the original database could not handle the new requirements, and expected queries were not returning the correct results.
 - Solution: A new database was implemented using Aiven (SQL), which was set up inside T/demo.
4. User list not displaying correctly:
 - Issue: The user list and their data were not being reflected properly in the queries.
 - Solution: Methods in LeaderboardService were improved, and adjustments were made in GameController to ensure that the data was fetched and displayed correctly.
5. Games not finishing correctly:
 - Issue: After multiple attempts, games were not ending properly due to errors in database integration.
 - Partial Solution: LeaderboardService3 was created to centralize API calls and handle the score updates.

7.2 Solved Issues

7.2.1 Saving new games

- Issue Description: Despite multiple approaches, the saving of new games to the database was not completed correctly.
- First Attempt: We attempted to implement a manager directly in GameController, using data obtained through ExternalUser (from T), trying to call the methods in UserImp. However, we could never properly access T/demo.
- Second Attempt: We replicated the ExternalUser, ExternalUserService, and related classes directly in T5, but the issue persisted.

- Third Attempt: We tried to replicate the approach used in leaderboard.js, making the API calls from GameController. However, this was not possible due to the separation between frontend (JavaScript) and backend (Java).
- Final Approach: We created LeaderboardService3 to manage all API calls, including getAllResults, getUser, updateScore, and score. We also created a handleUserScore method to check if a user already existed in the database and update their stats or add a new user if necessary.

Although the API seems to respond correctly, the game does not finish, and we are unable to determine why.

8. Future Developments

To address the remaining issues and improve the application's stability, the following future developments are proposed:

8.1 Resolve the issue of game completion

- Investigate in detail why the game does not finish correctly after making the API call.
- It is suspected that this may be related to CORS (Cross-Origin Resource Sharing) issues in the T5 project, which could be blocking the HTTP requests.

8.2 Validation and Debugging of LeaderboardService

- Additional testing should be conducted to verify if the handleUserScore method is working correctly under various conditions.
- Investigate possible blockages in code execution when making calls to the database.

8.3 Improvement in Backend-Frontend Communication

- Optimize the interaction between GameController and the service responsible for managing the database.
- Explore the possibility of directly integrating the API into the game flow to prevent request blockages.

8.4 Review the API Configuration in T5

- Check the CORS settings and access permissions to ensure there are no unnecessary restrictions on HTTP requests.

8.5 Real-Time Monitoring

- Implement real-time monitoring tools to observe the flow of HTTP requests and detect potential errors in communication between services.

