Department of Computer Science

Technical University of Cluj-Napoca

Software Design

Laboratory project

Name: Bogdan-Mihai Doia
Group: 30431

Teaching Assistant: Teodora-Melania Furcovici

Contents

1 Project Specification **3**
1.1 Domain Model Diagram . . . 3

2 Use-Case Model **4**
2.1 Users and Stakeholders . . . 4
2.2 Use-Case Identification . . . 4
2.3 UML Use-Case Diagram . . . 5

3 Architectural Design **6**
3.1 Conceptual Architecture . . . 6
3.2 Package Diagram . . . 7
3.3 Class Diagram . . . 8
3.4 Database (E-R/Data Model) Diagram . . . 9
3.5 Sequence Diagram . . . 10
3.6 Activity Diagram . . . 11

4 Supplementary Specifications **12**
4.1 Non-Functional Requirements . . . 12
4.2 Design Constraints . . . 12

5 Testing **13**
5.1 User Controller API Endpoints . . . 13
5.1.1 GET /api/users/ . . . 13
5.1.2 GET /api/users/register.html . . . 13
5.1.3 POST /api/users/register . . . 13
5.1.4 GET /api/users/login.html . . . 14
5.1.5 GET /api/users/not_logged_in.html . . . 14
5.1.6 POST /api/users/login . . . 14
5.1.7 GET /api/users/normal_dashboard.html . . . 15
5.1.8 GET /api/users/admin_dashboard.html . . . 15
5.1.9 GET /api/users/user . . . 15
5.1.10 GET /api/users/get_user_by_id/{userId} . . . 16
5.1.11 GET /api/users/admin_user_management.html . . . 16
5.1.12 GET /api/users/create_user.html . . . 16
5.1.13 POST /api/users/create_user . . . 17
5.1.14 GET /api/users/update_user.html . . . 17
5.1.15 PUT /api/users/update_user/{userId} . . . 17
5.1.16 GET /api/users/delete_user.html . . . 18
5.1.17 DELETE /api/users/delete_user/{userId} . . . 18
5.1.18 GET /api/users/update_account.html . . . 19
5.1.19 PUT /api/users/update_account/{userId} . . . 19

5.2 Saved Team Controller API Endpoints 19
5.2.1 GET /api/userteams/saved_teams.html 19
5.2.2 POST /api/userteams/saveCurrentTeam 20
5.3 Character Controller API Endpoints 20
5.3.1 GET /api/characters/characters.html 20
5.3.2 GET /api/characters/selectMainDPS.html 20
5.3.3 POST /api/characters/confirmMainDPS 21
5.3.4 GET /api/characters/recommended$_t$*eam.html* 21
5.3.5 POST /api/characters/processSelectedCharacters 21
5.3.6 GET /api/characters/show$_t$*eam.html* 22
5.4 Future Improvements 22

6 Bibliography **23**

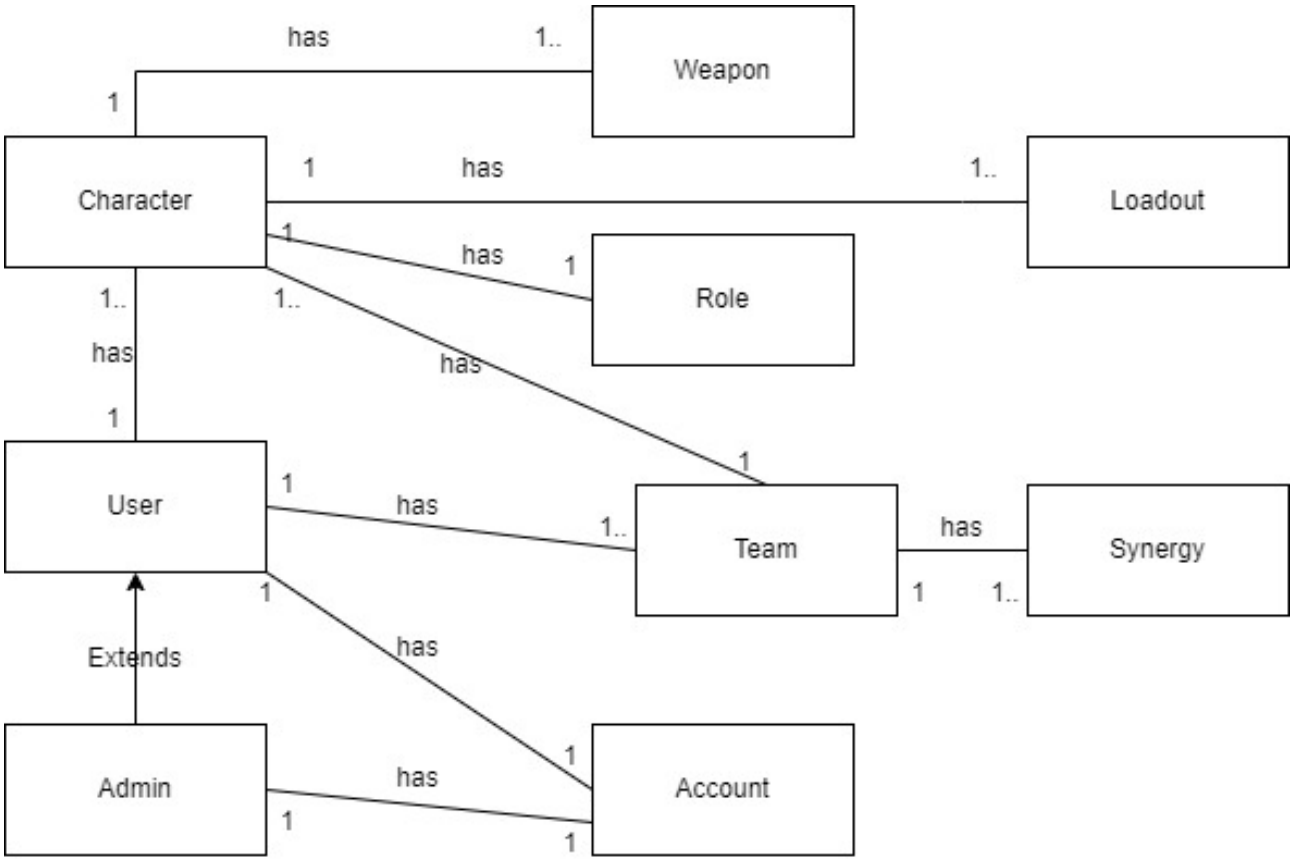
Chapter 1

Project Specification

This project develops a tool specifically for Genshin Impact players to streamline the optimization process for characters, teams, and equipment. Through a user-friendly interface, players input their roster, and the tool generates personalized recommendations for team compositions, weapon choices, and artifact selections.

These tailored suggestions aim to enhance the overall gaming experience by maximizing the synergy and effectiveness of character setups.

1.1 Domain Model Diagram



Chapter 2

Use-Case Model

The "Use-Case Model" chapter explains how players use the software tool. It covers actions like entering player rosters and getting personalized suggestions for teams, weapons, and artifacts. These features make it easier for Genshin Impact players to optimize their characters, improving their gaming experience by making characters work better together.

2.1 Users and Stakeholders

Users

There are two user types: the general user and the admin. The general user can utilize the application to the fullest as expected. The admin also has tools that allow modifying and adding to the application's data, such as characters and their recommendations.

Stakeholders

The stakeholders are the target audience: people who play the game Genshin Impact and are looking for recommendations to optimize their account.

2.2 Use-Case Identification

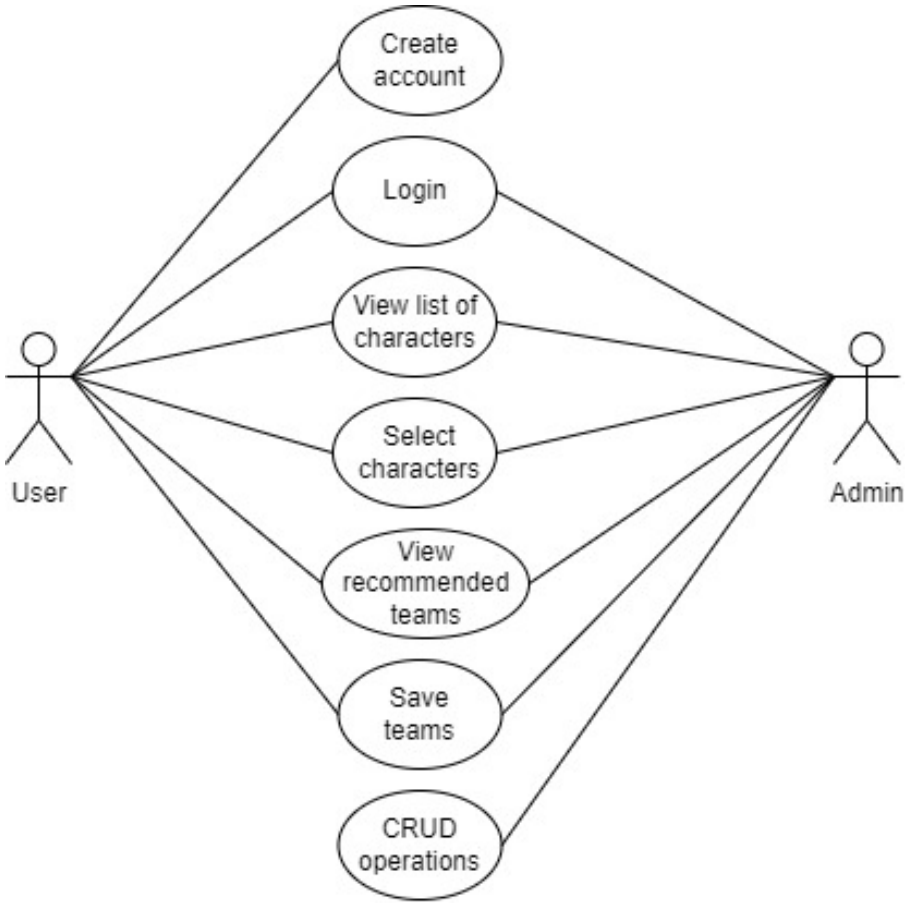
Use case name: Create account
Level: User-Goal
Main actor: User
Main success scenario: The user creates a new account by providing the required information if it doesn't already exist.
Extension: If the account already exists, the creation will not be completed.

Use case name: Select characters
Level: User-Goal
Main actor: User
Main success scenario: The user selects the characters they possess to inform the application of their available choices.
Extension: If the user does not select any characters, the application prompts them to choose at least 4 characters.

Use case name: Update recommendations
Level: Subfunction
Main actor: Admin
Main success scenario: The admin successfully modifies what the app will recommend to the user based on the selected characters, including team compositions and character-specific information.
Extension: If the admin enters incompatible data, the app signals an input error.

Use case name: Save team
Level: User-Goal
Main actor: User
Main success scenario: The user saves the team recommended by the app based on the selected characters and can view it later.
Extension: If the team has already been saved by the user, the save operation will fail.

2.3 UML Use-Case Diagram



Chapter 3

Architectural Design

The application employs a Layered architectural style to effectively organize modules and consolidate related classes. This architectural approach aligns with the application's design philosophy, as it allows for the segmentation of various functionalities. This segmentation enhances readability and facilitates debugging processes by providing clear delineation of the components.

3.1 Conceptual Architecture

In terms of application type, I've opted for a Web application, adhering to the Client-Server pattern. This choice aligns with the primary objective of serving new players to the game. Additionally, all information will be stored within a SQL database.

Regarding the architectural layers, the application is structured into five distinct layers, each with its designated role:

1. **Presentation Layer** – This layer is responsible for defining the components necessary for the user interface (UI). It ensures the presentation of data and interaction elements to the user.

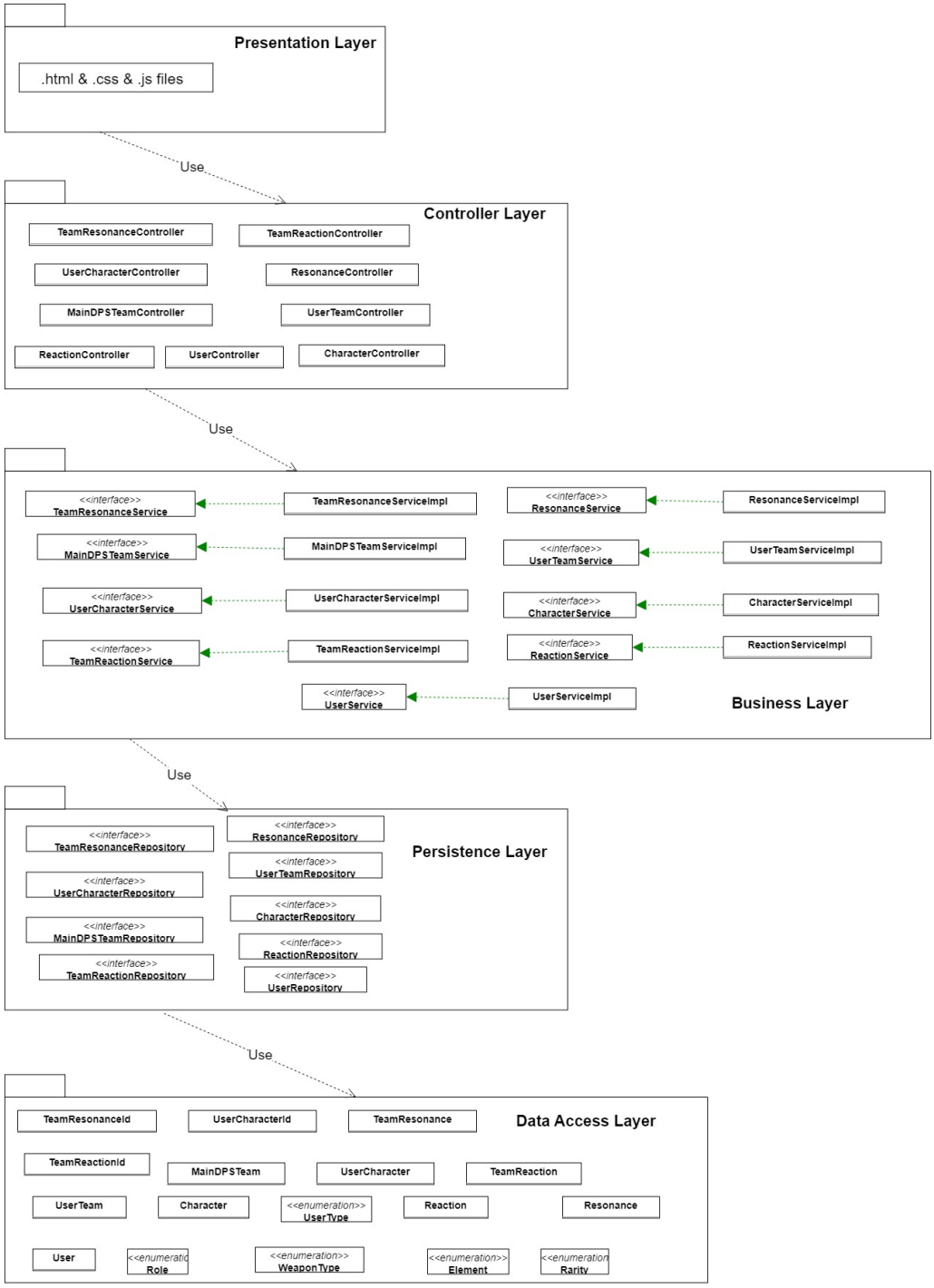
2. **Controller Layer** – Serving as the bridge between the UI and the core functionalities of the application, the Controller Layer manages user inputs and delegates appropriate actions to the underlying components.

3. **Business Layer** – At the heart of the application, the Business Layer encompasses the logic, validation, error handling, and functional operations. It orchestrates the core functionality of the application and ensures its coherence and integrity.

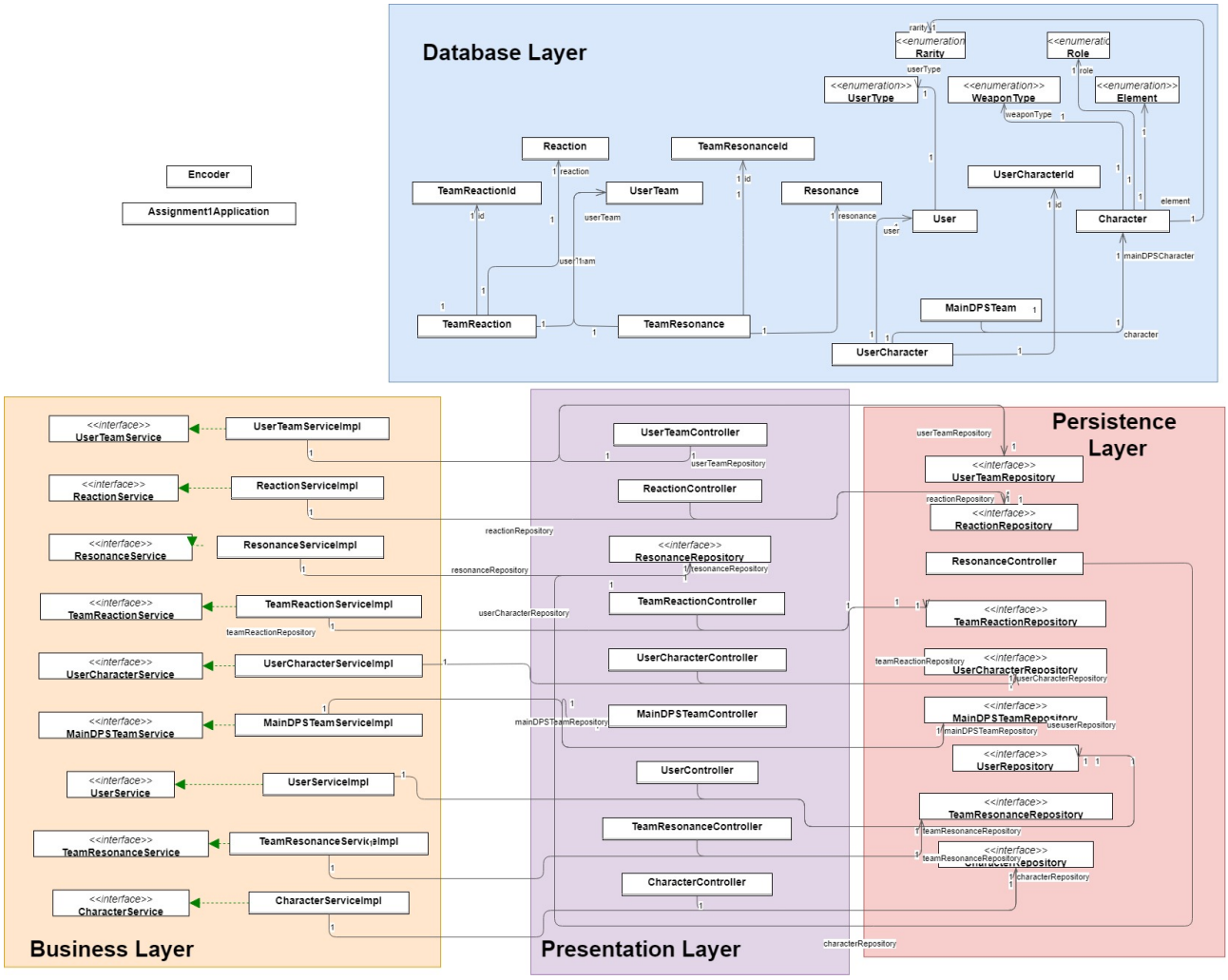
4. **Persistence Layer** – Responsible for interfacing with the database, the Persistence Layer facilitates data access, retrieval, and modification. It executes database queries and transactions, forwarding results to the Business Layer for processing.

5. **Database Layer/Model Layer** – This layer defines the classes or entities that represent the data stored in the database. It essentially maps the application's domain model to the database schema, ensuring seamless interaction between the application and the underlying data storage.

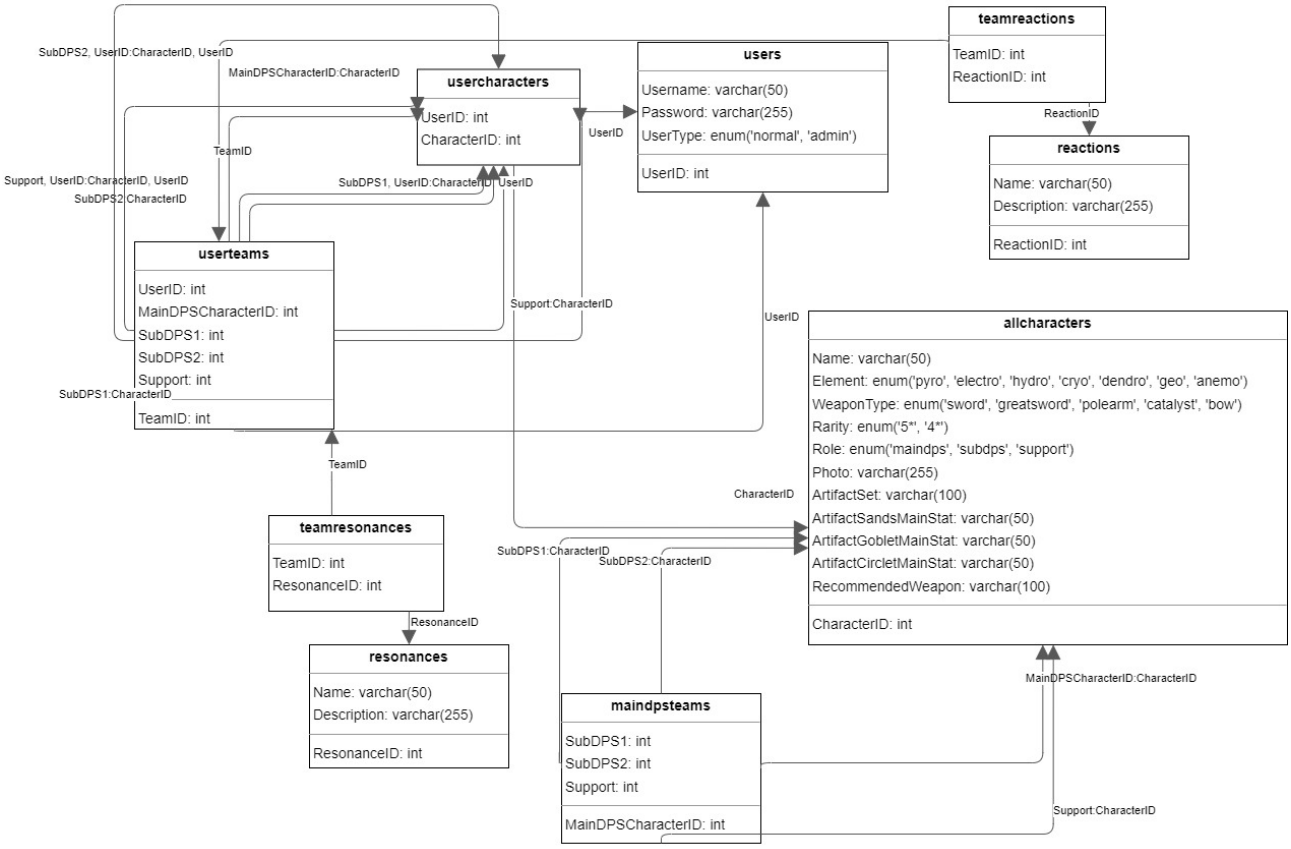
3.2 Package Diagram



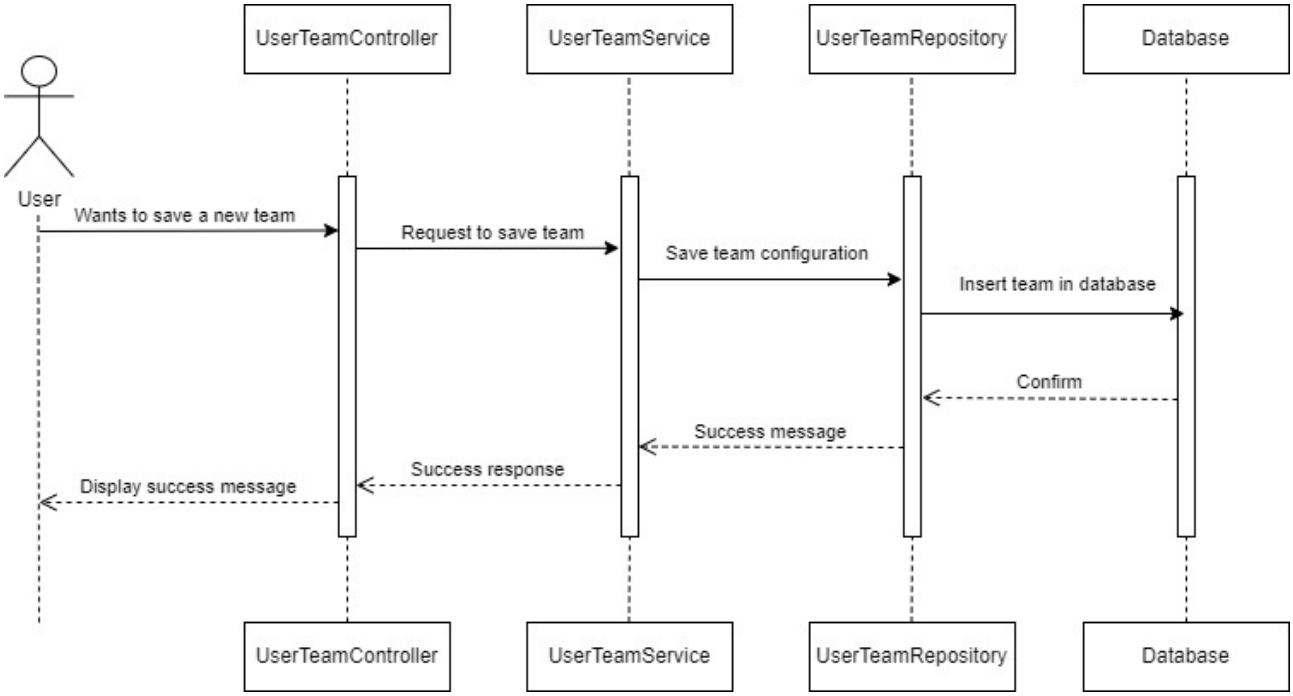
3.3 Class Diagram



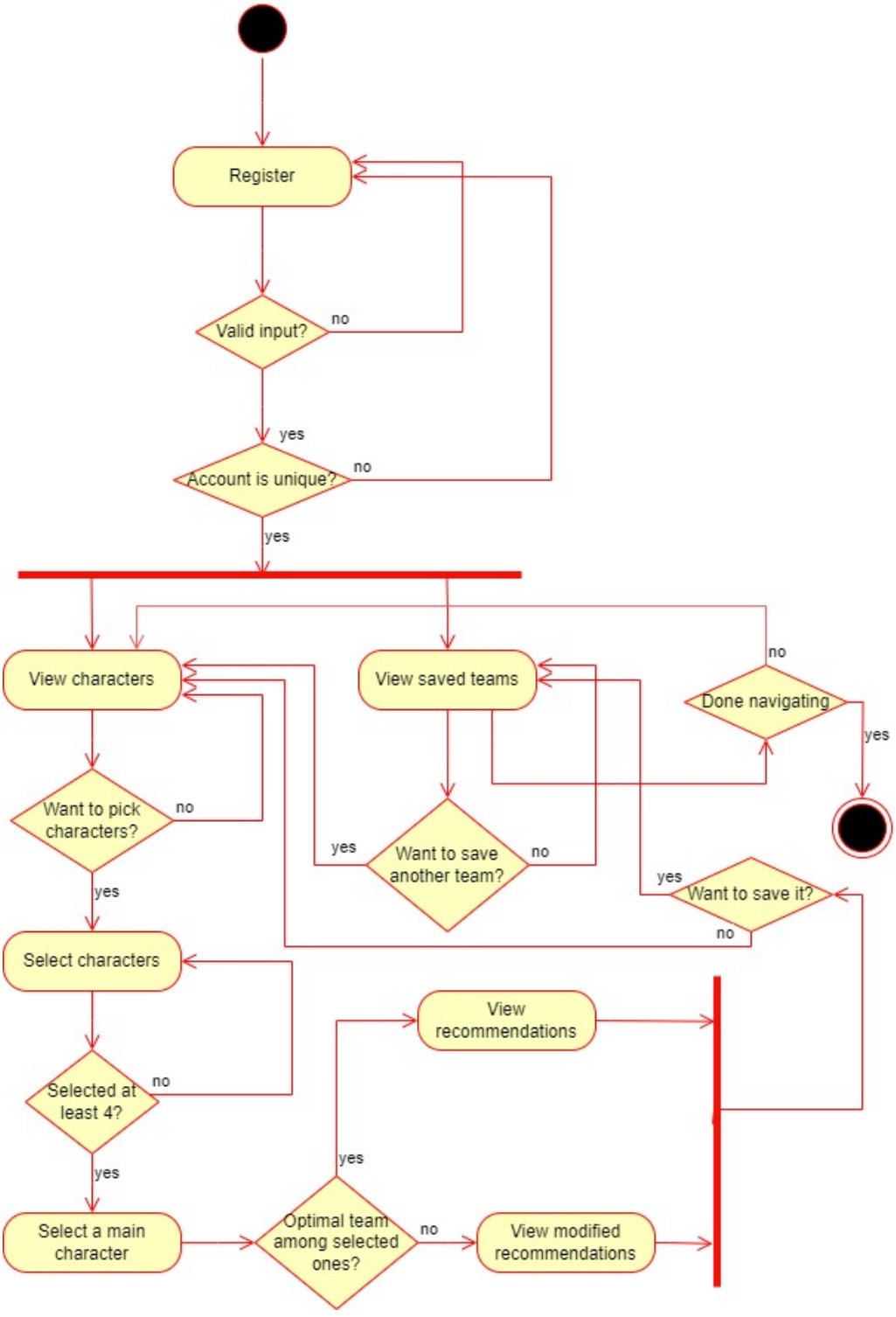
3.4 Database (E-R/Data Model) Diagram



3.5 Sequence Diagram



3.6 Activity Diagram



Chapter 4

Supplementary Specifications

4.1 Non-Functional Requirements

1. **Performance:** The application should have fast response times for user interactions, with minimal latency in loading pages and executing requests.
2. **Security:** The system should implement strong authentication and authorization mechanisms to protect sensitive user data and prevent unauthorized access.
3. **Usability:** The user interface should be intuitive and easy to navigate, with clear instructions and feedback provided to users.
4. **Maintainability:** The codebase should be well-organized and documented, making it easy for developers to understand and maintain the application in the future.
5. **Data Integrity:** The application should ensure the accuracy and consistency of data stored in the database, with appropriate validation and error handling mechanisms in place.

4.2 Design Constraints

Create an application utilizing the JAVA programming language and integrate two frameworks: Spring and Hibernate for database connectivity. MySQL will be utilized as the backend database. For the frontend, React will be employed due to its beginner-friendly nature. The communication between the backend and frontend will be facilitated through API calls.

Chapter 5

Testing

5.1 User Controller API Endpoints

5.1.1 GET /api/users/

Description
Displays the index page.
Request Parameters
None.
Request Example
No request parameters needed.
Response Codes

- **200 OK**: Returns the index HTML page.

5.1.2 GET /api/users/register.html

Description
Displays the registration page.
Request Parameters
None.
Request Example
No request parameters needed.
Response Codes

- **200 OK**: Returns the register HTML page.

5.1.3 POST /api/users/register

Description
Registers a new user.
Request Parameters

- **user (User)**: The user object containing registration details.

Request Example

```
POST /api/users/register
Content-Type: application/json

{
    "username": "example_user",
    "password": "example_password"
}
```

Response Codes

- **200 OK**: User registered successfully.
- **500 INTERNAL SERVER ERROR**: Failed to register user.

5.1.4 GET /api/users/login.html

Description

Displays the login page.

Request Parameters

None.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the login HTML page.

5.1.5 GET /api/users/not_logged_in.html

Description

Displays the not logged in page.

Request Parameters

None.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the not logged in HTML page.

5.1.6 POST /api/users/login

Description

Logs in a user.

Request Parameters

- **user (User)**: The user object containing login details.

Request Example

```
POST /api/users/login
Content-Type: application/json

{
    "username": "example_user",
    "password": "example_password"
}
```

Response Codes

- **200 OK**: User logged in successfully, returns the dashboard URL.
- **401 UNAUTHORIZED**: Invalid username or password.

5.1.7 GET /api/users/normal_dashboard.html

Description

Displays the normal dashboard if the user is logged in and is a normal user, otherwise redirects to the not logged in page.

Request Parameters

None.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the normal dashboard HTML page if the user is logged in and is a normal user.
- **302 FOUND**: Redirects to the not logged in page if the user is not logged in or is not a normal user.

5.1.8 GET /api/users/admin_dashboard.html

Description

Displays the admin dashboard if the user is logged in and is an admin user, otherwise redirects to the not logged in page.

Request Parameters

None.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the admin dashboard HTML page if the user is logged in and is an admin user.
- **302 FOUND**: Redirects to the not logged in page if the user is not logged in or is not an admin user.

5.1.9 GET /api/users/user

Description

Retrieves a user by their ID.

Request Parameters

- **userId (Long)**: The ID of the user to retrieve.

Request Example

```
GET /api/users/user?userId=123
```

Response Codes

- **200 OK**: Returns the user if found.
- **500 INTERNAL SERVER ERROR**: Error response if user retrieval fails.

5.1.10 GET /api/users/get_user_by_id/{userId}

Description

Retrieves a user by their ID for messaging purposes.

Request Parameters

- **userId (Long)**: The ID of the user to retrieve.

Request Example

```
GET /api/users/get_user_by_id/123
```

Response Codes

- **200 OK**: Returns the user if found.
- **500 INTERNAL SERVER ERROR**: Error response if user retrieval fails.

5.1.11 GET /api/users/admin_user_management.html

Description

Displays the admin user management page if the user is logged in and is an admin user, otherwise redirects to the not logged in page.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the admin user management HTML page if the user is logged in and is an admin user.
- **302 FOUND**: Redirects to the not logged in page if the user is not logged in or is not an admin user.

5.1.12 GET /api/users/create_user.html

Description

Displays the create user page if the user is logged in and is an admin user, otherwise redirects to the not logged in page.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the create user HTML page if the user is logged in and is an admin user.
- **302 FOUND**: Redirects to the not logged in page if the user is not logged in or is not an admin user.

5.1.13 POST /api/users/create_user

Description

Creates a new user.

Request Parameters

- **user (User)**: The user object containing registration details.

Request Example

```
POST /api/users/create_user

{
    "username": "example_user",
    "password": "example_password"
}
```

Response Codes

- **200 OK**: Indicates successful user creation.
- **500 INTERNAL SERVER ERROR**: Error response if user creation fails.

5.1.14 GET /api/users/update_user.html

Description

Displays the update user page if the user is logged in and is an admin user, otherwise redirects to the not logged in page.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the update user HTML page if the user is logged in and is an admin user.
- **302 FOUND**: Redirects to the not logged in page if the user is not logged in or is not an admin user.

5.1.15 PUT /api/users/update_user/{userId}

Description

Updates an existing user.

Request Parameters

- **userId (Long)**: The ID of the user to update.
- **user (User)**: The updated user object.

Request Example

```
PUT /api/users/update_user/123

{
    "username": "updated_user",
    "password": "updated_password"
}
```

Response Codes

- **200 OK**: Indicates successful user update.
- **500 INTERNAL SERVER ERROR**: Error response if user update fails.

5.1.16 GET /api/users/delete_user.html

Description

Displays the delete user page if the user is logged in and is an admin user, otherwise redirects to the not logged in page.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the delete user HTML page if the user is logged in and is an admin user.
- **302 FOUND**: Redirects to the not logged in page if the user is not logged in or is not an admin user.

5.1.17 DELETE /api/users/delete_user/{userId}

Description

Deletes a user.

Request Parameters

- **userId (Long)**: The ID of the user to delete.

Request Example

```
DELETE /api/users/delete_user/123
```

Response Codes

- **200 OK**: Indicates successful user deletion.
- **500 INTERNAL SERVER ERROR**: Error response if user deletion fails.

5.1.18 GET /api/users/update_account.html

Description

Displays the update account form.

Request Parameters

- **userId (Long)**: The ID of the user whose account is being updated.
- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the update account HTML page if the user is logged in.
- **302 FOUND**: Redirects to the not logged in page if the user is not logged in.

5.1.19 PUT /api/users/update_account/{userId}

Description

Updates the user account.

Request Parameters

- **userId (Long)**: The ID of the user whose account is being updated.
- **user (User)**: The updated user object.

Request Example

```
PUT /api/users/update_account/123

{
    "username": "updated_user",
    "password": "updated_password"
}
```

Response Codes

- **200 OK**: Indicates successful user account update.
- **500 INTERNAL SERVER ERROR**: Error response if user account update fails.

5.2 Saved Team Controller API Endpoints

5.2.1 GET /api/userteams/saved_teams.html

Description

Retrieves the saved teams for the logged-in user and renders them on the saved_teams page.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the saved_teams HTML page with saved teams data.

5.2.2 POST /api/userteams/saveCurrentTeam

Description

Saves the current user team to the database.

Request Parameters

- **selectedCharacterId (String)**: The ID of the selected character.

Request Example

```
POST /api/userteams/saveCurrentTeam

{
    "selectedCharacterId": "123"
}
```

Response Codes

- **302 FOUND**: Redirects to the saved_teams page after successfully saving the current team.

5.3 Character Controller API Endpoints

5.3.1 GET /api/characters/characters.html

Description

Retrieves all characters and renders them on the characters page.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the characters HTML page with all characters data.

5.3.2 GET /api/characters/selectMainDPS.html

Description

Displays a page for selecting the main DPS character.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the selectMainDPS HTML page with the list of main DPS characters.

5.3.3 POST /api/characters/confirmMainDPS

Description

Confirms the selection of the main DPS character.

Request Parameters

- **selectedCharacterId (String)**: The ID of the selected character.

Request Example

```
POST /api/characters/confirmMainDPS

{
    "selectedCharacterId": "123"
}
```

Response Codes

- **302 FOUND**: Redirects to the recommended_team page after successfully confirming the main DPS character selection.

5.3.4 GET /api/characters/recommended$_t eam.html$

Description

Displays a page showing the recommended team composition.

Request Parameters

- **model (Model)**: The model to be populated with data.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Returns the recommended$_t eamHTMLpagewiththerecommendedteamcomposition.$

5.3.5 POST /api/characters/processSelectedCharacters

Description

Processes the selected characters for the team.

Request Parameters

- **selectedCharacterIds (String)**: The IDs of the selected characters.

Request Example

No request parameters needed.

Response Codes

- **200 OK**: Redirects to the selectMainDPS page.

5.3.6 GET /api/characters/show$_t$*eam.html*

Description

Displays a page showing the details of a saved team.

Request Parameters

- **teamId (Long)**: The ID of the team to display.
- **model (Model)**: The model to be populated with data.

Request Example

```
GET /api/characters/show_team.html?teamId=1234
```

Response Codes

- **200 OK**: Returns the show_team HTML page with the details of the saved team.
- **404 Not Found**: If the team with the specified ID is not found.

5.4 Future Improvements

Possible future improvements to the app include:

- Extending the functionality of the app's guide side to view a guide for each character.
- More options for weapons and artifact sets being recommended besides the best one, as it is not always the most accessible one or available for every player.
- Adding information about how, when, and where to obtain certain upgrades for characters.
- Adding the possibility of grouping characters by role or element.
- Including a tier list that gives users an understanding of the power level of a character compared to others.
- Adding descriptions of the character abilities.
- Including tips and tricks for using the recommended team.

Chapter 6

Bibliography

https://www.thymeleaf.org/documentation.html

https://docs.spring.io/spring-boot/documentation.html

https://game8.co/games/Genshin-Impact

https://genshin.gg/

https://genshin-builds.com/en

https://www.teyvatavern.com/