1222·2022 800 ANNI
UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Web Applications A.Y. 2023-2024
Homework 1 – Server-side Design and Development

Master Degree in Computer Engineering
Master Degree in Cybersecurity
Master Degree in ICT for Internet and Multimedia

Deadline: 29 April, 2024

Group Acronym	LIF	
Last Name	**First Name**	**Badge Number**
Busato	Nicola	2119291
Cini	Jacopo	2125757
Gusella	Michele	2122861
Miele	Riccardo	2116946
Momesso	Jacopo	2123874
Pozzo	Nicola	2125382

1 Objectives

The aim of this project is to develop a web application that will help the game-master during the game Lupus in Fabula (the basic rules of which are explained below). This app will help the game-master to keep track of what happens during the game, e.g. what role each player has, what happened during the night (which player is dead) or which player is voted during the day to be eliminated.

The traditional method of moderating the game involves manually recording actions on a phone or paper. This application aims to streamline the process, providing an organized platform for tracking player roles, nighttime actions, and daytime voting outcomes.

1.1 Rules of the game

In the remote village of Fabula, some people become werewolves at night. They attack an innocent person to satisfy their instincts. During the day, the survivors discuss what to do. At the end of the discussion, they lynch one of them. Who will survive the massacre?

- **Aim of the game**:
  There are two factions in the game: the Werewolves and the Villagers. The aim of the Werewolf faction is to eliminate all villagers. Conversely, the aim of the Villagers faction is to lynch all the Werewolves.
- **Preparation**:
  Before the game starts, one player is chosen to be the master/moderator. He doesn't belong to any faction and he will only manage the game tracking everything that will happen. The other players will play the role given to them by their card and look at their own card secretly.
- **Game**:
  The game is divided into two phases: night and day. At night each role with an effect that resolves during this phase will be called by the master to perform the respective action described in the role card. Once the master has called all roles with a night effect, night will end and the day phase will begin. During the day, people will discuss and vote to lynch someone, hopefully a wolf or a member of the wolf faction.
  - **Night**:
    The moderator declares the beginning of the night ("it is night, everyone close your eyes"). All players then close their eyes, trying not to make any kind of noise for all the duration of this phase. The master then begins to call each role with effect:
    Example of wolves calling:
    - Master: "Wolves open their eyes and choose who to go and maul."
    - Wolves silently agree and point to their prey
    - Master: "Decide who to send among you to maul your chosen prey"
    - Wolves indicate who among them will go
    - Master: "The wolves close their eyes."

    Example of good role calling (e.g: Seer):
    - Master: "Seer open eyes and choose whom to investigate"
    - Seer points to the player he wants to investigate
    - Master nods to say whether the player has a bad or good role
    - Master: "Seer closes eyes."

    Note: This phase should be played even if the called role is already dead in order to not give clues to others. Once the master has called all roles with night effect, the night is over.
  - **Day**:
    The master now declares the start of the day ("it's day, everybody open your eyes"). The master then gives a recap of what happened during the past night, listing who died (if anyone died), who was

anointed, etc. without giving explicit information about what happened (e.g. recap: Master: "A died, B died, C was anointed"). Once the master has finished the recap, the still-living players can then start discussing among themselves to decide who to lynch. You may lie freely but under no circumstances may you show your card to others.

- **Voting**:
  After a maximum of three minutes of discussion, the master stops the discussion and asks each player, starting with the one to the left of the one who died first and proceeding clockwise, who he thinks should be lynched. All players, including ghosts (i.e., those who died), in their turn vote indicating who they want to lynch. At the end of the voting, the two players with the most votes will be clued (in case of a tie, the player closest to the first dead will be chosen). The two suspected players can now take a short speech defending themselves against the charges. After the two speeches are over, the remaining alive unindicted players will vote again for the new player to be lynched among the two suspects (in this case the voting will be counterclockwise starting with the alive unindicted player who was last to vote before). At this point the day is over and the master can start a new night and so on until one faction wins.
  Note: all dead players (i.e. ghosts) for the rest of the entire game must not speak or show their role to other players.

- **Victory Condition**:
  The master declares the game over with a villagers victory if the villagers lynch all the werewolves. Werewolves, on the other hand, are declared winners if at any point in time they are equal in number to the still-living villagers (e.g., 2 werewolves and 2 villagers, or 1 and 1): in that case the werewolves unceremoniously maul the remaining villagers! In case there are victory stealer roles in the game, they will win if their victory condition is fulfilled, and in that case the master will declare them winners any time in the game when they have completed their victory condition.

This game requires at least five participants, one of whom will be the game-master, and the other will get roles assigned randomly, for instance by drawing a card. The main roles are wolves and farmers, but other roles are used - like medium, knight, werehamster etc - in order to make the game more entertaining, with specific rules applying to each character.

The table of all the possible roles is shown in the next page [1.1].

Name of the role	Faction	Description
Berserker	Wolf	One night can kill two players but then dies (can by pass the protection of the knight).
Carpenter	Villager	If it's voted out he can refuse to wood for the bonfire (once per game).
Dorky	Wolf	It's a wolf but does not know the others wolves. Every night can ask if a player is a wolf and if it is, it can join the group.
Explorer	Wolf	Once per game can kill a player ignoring other effects, after that it's a simple wolf.
Farmer	Villager	Villager with no special effect.
Giuda	Wolf	It's seen as a farmer by seer but wins with wolves. It knows only one wolf.
Hamster	Victory Stealer	Can die at night only if it's probed or protected. Wins if it survives with villagers.
Hobbit	Villager	Can't die from wolves if there are more than one.
Illusionist	Villager	During night can block the effect of a player.
Jester	Victory Stealer	Wins alone if it's burned.
Kamikaze	Villager	If it's attacked during night, kills the wolf that has attacked (wolf bombarder can defuse the bomb).
Knight	Villager	Can protect one player (not twice in a row).
Medium	Villager	Can ask if the player burned is a wolf or not.
Plague spreader	Villager	During night can point a player, during the following day the player cannot answer yes or no (if it does, it will die and the two players closer will suffer the same effect.
Puppy	Wolf	It's seen as a villager and cannot kill during night. If it's the last remaining wolf, it becomes a wolf.
Sam	Villager	If it's voted out can kill another player.
Seer	Villager	At night can ask if a player is a wolf or not.
Sheriff	Villager	Can kill a person during night, if it's not a wolf it will die.
Wolf	Wolf	Wolf with no special effect.

Table 2: Table of all possible roles

2 Main Functionalities

2.1 How a game session works

To play the game, all the players need to be registered (selecting a username + password and giving the email) to the Lupus Web App. As said in the previous chapter, there are two type of players:

- **MASTER**: The master directs the game, managing the flow of the game and writing down all the events that have happened.
- **OTHERS**: The other players have a specific role (with special effects) and will play the game, trying to win based on the role that they have.

The app permit to the master to efficiently manage the game and to record all the events that have happened while permit to other players to see their role; both can see the current status of the game (the number of the round, the phase and the players that have died and, off course, who is still alive); the masters also can see which role has every player.
A game session in the app works as follow:

1. The master creates the game adding all the players using their username. Then chooses the roles for the game and how many instances of each role there will be (N.B. some roles can have only one instance). After that the game can start.
2. The app will assign to the players (not to the master) a role at random from the ones selected by the master.
3. The game starts and the app will guide the master through the game suggesting what it's needed to do and permitting the master to write down what happened during each phase (for example, at night phase, the app will "ask" the game-master which is the target of the wolves).
4. When the game finishes, the master saves the logs and close the session. The app will create the logs of the game.

All the players, including the master, can see the logs of the game from their personal profile. The profile permit also to visualize the logs of all the matches played and also the personal statistics (how many roles have played and how many wins for example).

2.2 Functionalities and services

The main functionalities of the Lupus Web App are:

- Management of a game for the master
- Visualization of the current status of the game for all the players (and game-master)
- Random assignment of the roles to the players
- Visualization of personal statistics (can also see the statistics of other players)
- Visualization of the logs of a game (and also of all game played)
- Visualization of the rules of the game (not needed to be logged)
- Visualization of the roles and their corresponding effects
- Management of friends list (adding or deleting friends)

3 Data Logic Layer

3.1 Entity-Relationship Schema

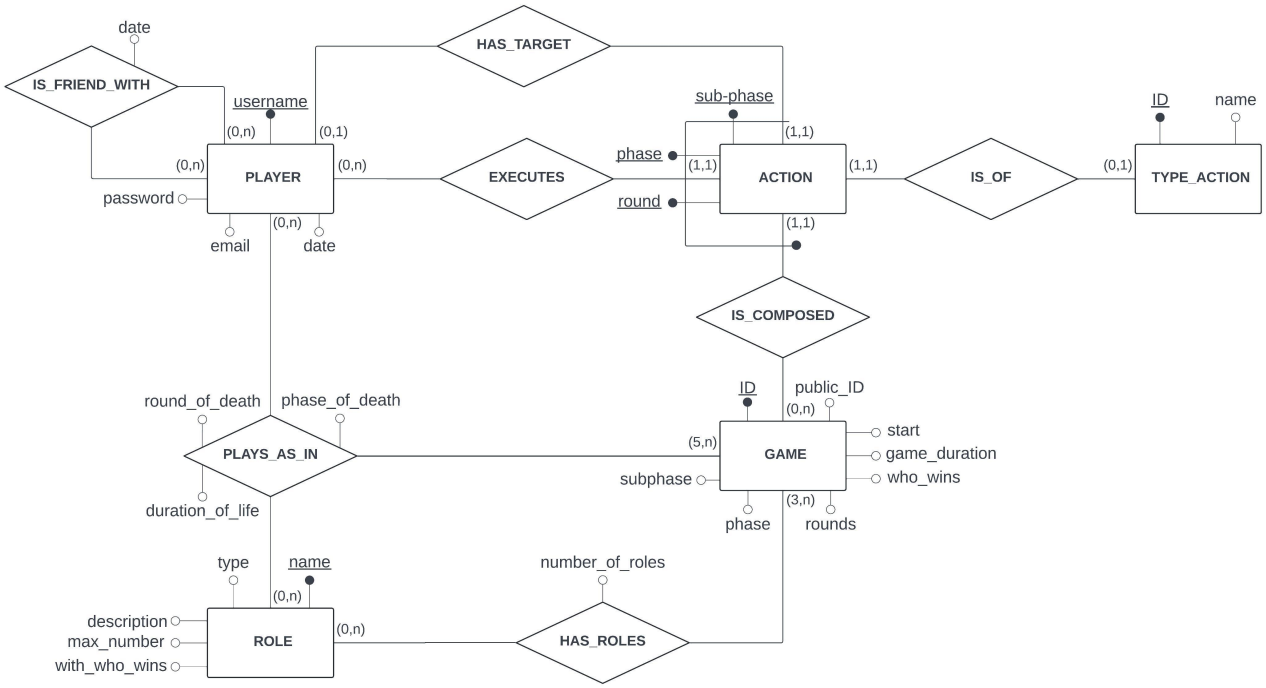


Figure 1: ER scheme of LUPUS app

The ER schema contains several entities that are:

- **PLAYER**: the player entity represents the person that has subscribed to LUPUS app in order to play the game. The player creates an account providing the email and choosing a username and password. The primary key is the username (type: VARCHAR(20)) while the other attributes are email (type: VARCHAR(100)), password (type: VARCHAR(100)) and date (type: DATE). The password is hashed through md5 and none attribute can be null. The email has to be unique.
  Each player can have a list of friends that it's recorded in the relation called IS_FRIEND_WITH (containing also the attribute date (type: DATE) that indicates the date in which the player has added another player into its friends list). A player can have its friend list empty and a player does not necessarily have to be on a friends list.
  A player plays a specific role in a game, so is in a relation with the entities GAME and ROLE called PLAYS_AS_IN in which are stored the information of the duration of the life during the game. These information are contained into the attributes round_of_death (type: SMALLINT), phase_of_death (type: SMALLINT, 0 for night and 1 for day) and duration_of_life (type: TIME).
  A player also can execute actions during the game so it participates in the relation EXECUTES and can be the target of an action so the player has cardinality of (0,1) with the relation HAS_TARGET.
  All the other cardinality of the player are (0,n).

- **ROLE**: the role entity indicates the role that a player can have during the game. The role is identified by a primary key name (type: VARCHAR(20)) and has other attributes like type (type: SMALLINT), with_who_wins (type: SMALLINT), max_number (type: SMALLINT) and description (type: CHARACTER VARYING).

Name contains the name of the role, type indicates if the role is good, bad, neutral, victory stealer or master; with_who_wins indicates if the role wins the game with the faction of wolves, with farmers, if it wins for itself or it's a master; max_number contains the maximum possible number of players with the same role in a game; description contains the description of the role.
Role participates in the relation PLAY_AS_IN and in HAS_ROLES both with cardinality (0,n).
HAS_ROLES contains the attribute number_of_roles (type: SMALLINT) that indicates how much instances of a specific role are in a game. If there are no tuples of a specific role this means that the role was not present in that game.

- **GAME**: game is the entity that represents the match. It is identified by the primary key ID (type: SERIAL) and contains also the attributes public_ID (type: CHARACTER VARYING), start (type: TIMESTAMP), game_duration (type: TIME), who_wins (type: SMALLINT), rounds (type: SMALLINT), phase (type: SMALLINT) and subphase (type: SMALLINT). public_ID is unique and indicates public identifier of a game, it is composition of three roles taken at random from the ones in the game (not the master). start and game_duration contain the information about the start of the match and the duration; who_wins indicates which faction has won (farmers, wolves, hamster or jester or if the game isn't over yet); rounds represents the number of rounds played during the match; phase and subphase contain the phase and the subphase of the current game or the last phase and subphase of the game if it's finished.
  A game is composed of several actions so participates with cardinality (0,n) into the relation IS_COMPOSED.
  A game also can be played only if there are at least 5 players (cardinality (5,n) with relation PLAYS_AS_IN) and at least 3 roles (cardinality (3,n) with relation HAS_ROLES). The roles needed are wolf, farmer and master.
- **ACTION**: action is the entity that stores all the actions that happened during a game. It is identified by a composite key formed by three identifiers: one referred to the game (through the relation IS_COMPOSED) and two to the player (through the relations EXECUTES and HAS_TARGET) and by other three attributes: round, phase and subphase (all the types: SMALLINT).
  The entity action participates also to another relation IS_OF with cardinality (1,1). The relation HAS_TARGET indicates the player that is the target of the action (can also be itself).
- **TYPE_ACTION**: every action can be of a specific type like protecting, killing and seeing (the seer can ask the master if a certain person is a wolf). The type of action is identified by an ID (type: SERIAL) and there is an attribute name (type: VARCHAR(20)) containing the name of the specific type of action. A specific type of action (not the most important but maybe types of certain role) does not necessarily have to happen in a game so the cardinality with IS_OF is (0,1).

3.2 Other Information

3.2.1 Restructuration

We have decided to restructure the ER scheme deleting the entitiy TYPE_ACTION (with relation IS_OF) and the relation HAS_ROLES. The modifications made are:

- HAS_ROLES has been deleted because we have decided to retrieve the attribute number_of_roles using a query.
- TYPE_ACTION has been removed and substituted with an attribute type_action (type: VARCHAR(20)). The field of the attribute is managed with an enumeration.

We have decided to improve the efficiency of the app reducing the size of the database. In order to achieve this goal these modifications were necessary.
The restructured ER scheme can be seen in the image in the following page [2].

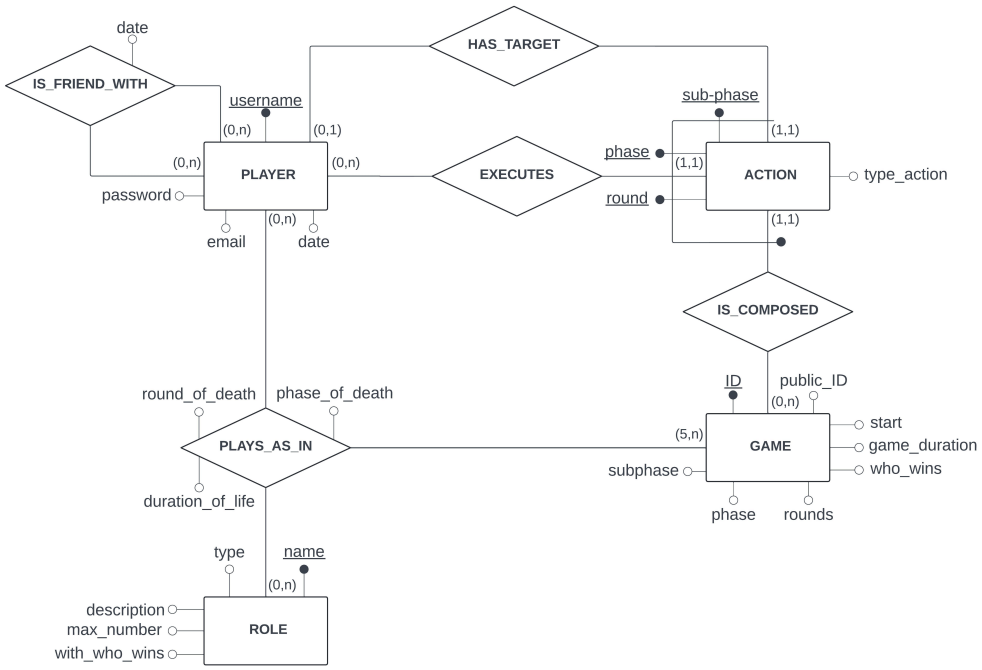


Figure 2: Restructured ER scheme of LUPUS app

3.2.2 Logical Scheme

To better visualize the representation of the ER scheme we have created the logic scheme of it. This has permitted to have a better view of the tables that are present in the real database.
To create the logic scheme we have done some modifications:

- The table ACTION has two "new" attributes one referring to PLAYER (deletion of relation EXECUTES) and one to GAME (deletion of relation IS_COMPOSED). So the primary key is now composed of six attributes.
- The relation HAS_TARGET has been added into the table ACTION with a reference to the table PLAYER.

The logic scheme of the Lupus App can be seen in the image below [3].

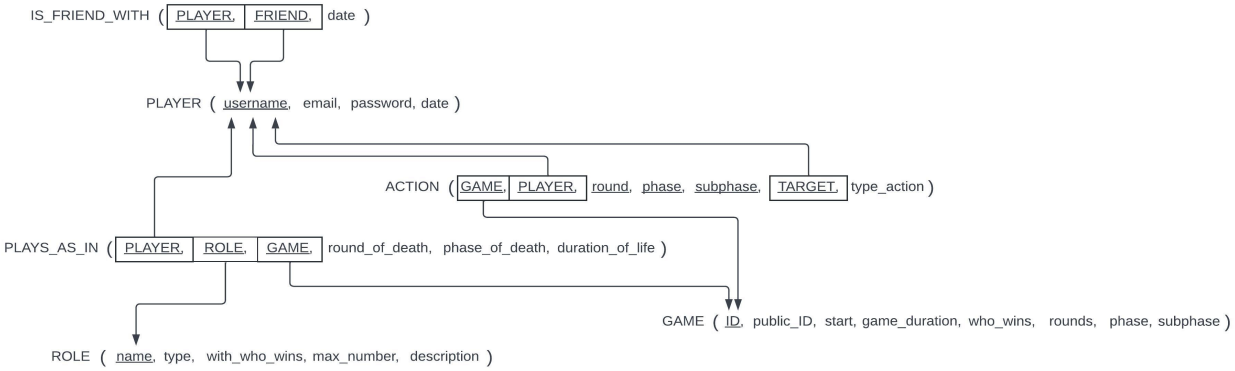


Figure 3: Logic scheme of LUPUS app

3.2.3 Other informations

The only table in which the insert is not possible is the Role table. In this table there are all the possible roles and, for this reason, insert is not possible.
In a future other roles can be implemented, so it can be possible the insert only for developers (maybe the new roles to add will be suggested by players subscribed to the app).
The update operation can be done to all the tables except role and action.
The delete is only possible in is_friend_with tables. This is because a player can decide to delete a friend from its friends list.

4 Presentation Logic Layer

The web site have two main sections: Game section and Player section.
The pages developed for the Lupus in Fabula web app are as follows:

- **Login/Signup Page:** Single page use for both login and sign up, allowing each user to have his own account, where the games he plays will be saved.
- **Home page:** Allows to access the personal area, to enter in an existing game or create a new one and view their logs and statistics.
- **Rules page:** On this page, users can view the game is rules and the various roles.
- **Game section:** in this section the player can create a game and take the role of game-master or if the user is already inside a game in progress it allows him to view his role.
  - **Game settings page:** on this page, if the player is not already in a game, it will allow him to create a new one, otherwise he can enter the current game.
  - **InGame Master view page:** page dedicated only to the game-master, it is similar to the Player page, but from this page the game-master can not only see the status of the game, but also which player has which role. Furthermore, he can mark the actions that the players perform.
  - **InGame Player view page:** this page is dedicated to players who are participating in a game, they can see their role and current game status (e.g. who is alive and who is dead).
  - **Log page:** shows the actions that happened during the game.
- **Player section** In this section the player can view his statistics, e.g. how long he has been playing, how many games he has won, etc., and can see the history of the games he has played.
  - **User page:** From here the user can see his or her own information, e.g. email, change password and manage friends.
  - **Statistics and Logs page:** the user can see the statistics of the matches he has played, e.g. the number of matches played, won and lost, roles most played etc, and shows the logs of the games that he played.

4.1 Login/Signup page

We have a single page that allows the user to login or sign up into the system. For the sign up we require an unique username and email. To do the login it's possible to use either, a username or the email, combined with the password. A data check is carried out, e.g. the username's length must be between 3 and 29 characters, the email must be of a correct format and the password must comply with security standards, i.e. be 8 characters, with upper, lower case, numbers and special characters, (see figure [4]).

4.2 Home page (Interface Mockup)

From here, the user can access the main sections of the site, i.e. he can create a game (but only if he is not already in a game that he has started), enter a game in which he is a playing player, access his statistics and see his game history, or he can enter his private area, (see figure [5]).

4.3 Rules page (Interface Mockup)

On this page it is possible to see all the rules of the game, so what Lupus in Fabula is and how to play it, and it also shows the roles in the game, with their characteristics and how they win, divided into various categories, (see figure [6]).

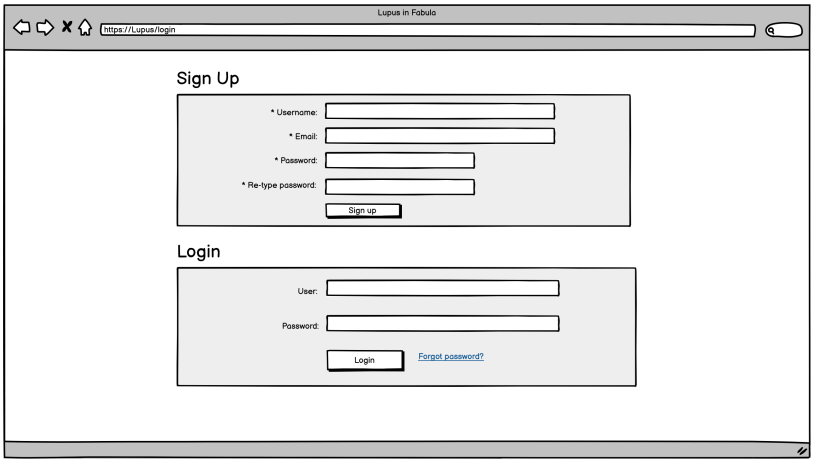


Figure 4: Login/Signup page

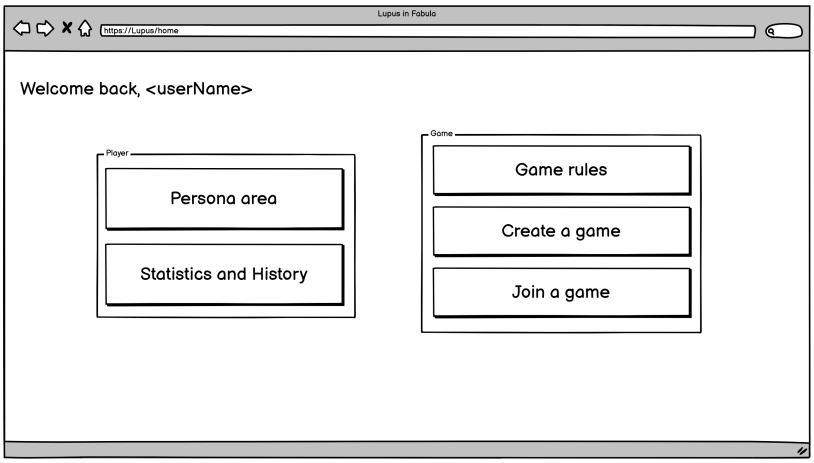


Figure 5: Home page

4.4 Game section

4.4.1 Game settings page (Interface Mockup)

From here the game-master has the possibility to create a new game, choose the game settings, such as how many wolves and how many farmers, and which other roles will be active. The Game-master can also choose the players who will play in that game, he can easily add his friends or search for them manually via username, (see figure [7]).

4.4.2 InGame Master view page (Interface Mockup)

From this page the Game-master is able to see the situation of the game, entering what happened during that phase, (visible in the top section of the mockup). During the night phase he is asked to select which player was eaten by wolves, which player was protected by the Knight, etc., while during the day phase he is asked which person each player voted to kill. At the end of each phase, (night or day), the application will respond what happened i.e. who died or who received some malus for example from the Plague Spreader.
The Game-master can see the current state of the game, so he knows the role of each player, whether he is alive or dead, (left section of the mockup). In addition he can see what happened in the previous phases via the logs

Figure 6: Rules page

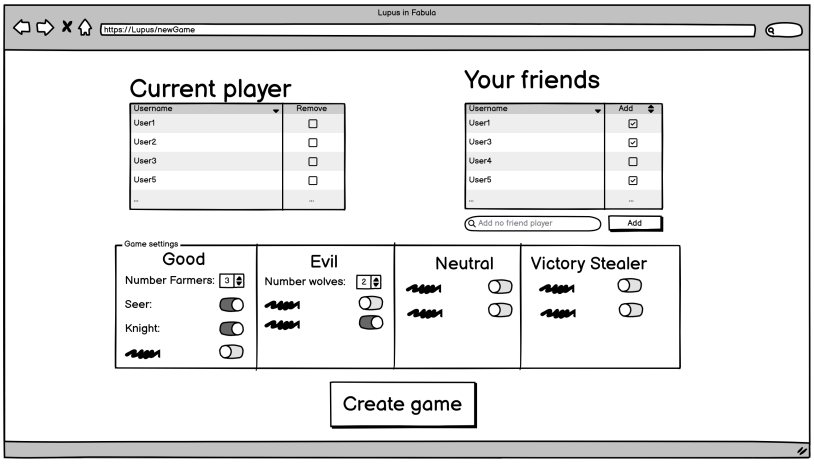


Figure 7: Game settings page

(bottom right part of the mockup), so he can go through each phase to view what happened (see figure [8]).

4.4.3 InGame Player view page (Interface Mockup)

It is similar to the Game-master page, but with the difference that the player can only see his private game information, such as what role he has, seeing his card with the role description, and he can see the public information, i.e. who is still alive, who has died and who has been plagued by the Plague Spreader.
It can also see the match logs, but only the public information, so who died and in what round and phase and who voted who (see figure [9]).

4.4.4 Log page (Interface Mockup)

From here it is possible to see the logs of a game, i.e. what happened, who killed/protected/voted for whom, at what stage who died, who won etc. This page shows all the information, even what happened during the night, but only when the game is over, if not it shows only the public information (see figure [10]).

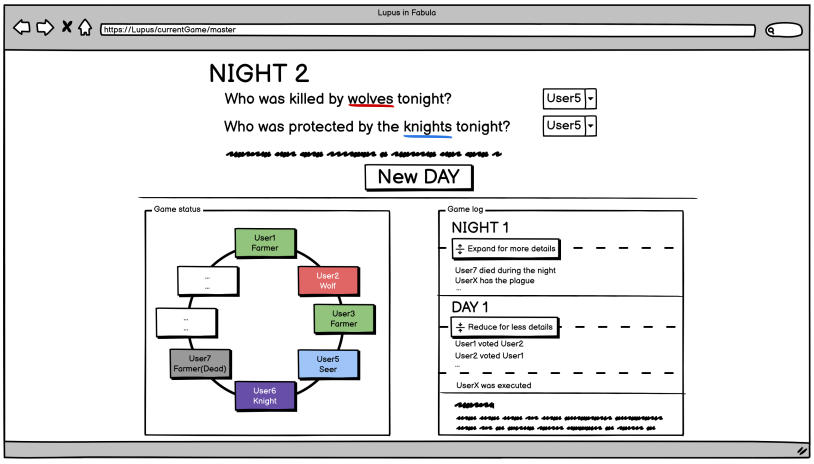


Figure 8: InGame Master view page

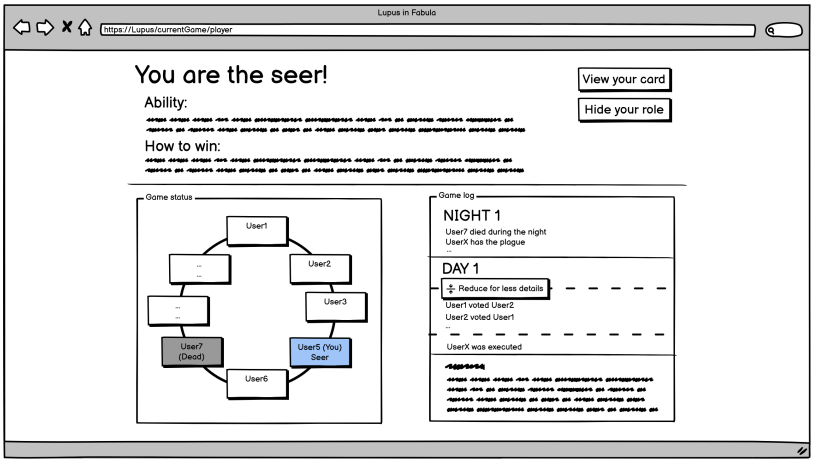


Figure 9: InGame Player view page

4.5 Player section

4.5.1 User page (Interface Mockup)

From this page, the user can manage his profile. He has the possibility of viewing his friends with some associated statistics, e.g. number of games played together and how long they have been friends. The user has the possibility of removing friends. Moreover, he can change his credentials like the email or the password (see figure [11]).

4.5.2 Statistics and Logs page (Interface Mockup)

From here the user can see his statistics, i.e. the time played, the number of games played, won and lost, the roles he played and how many times. He can also see the history of the games he has played, with some associated information and has the possibility of viewing the entire match log (see figure [12]).

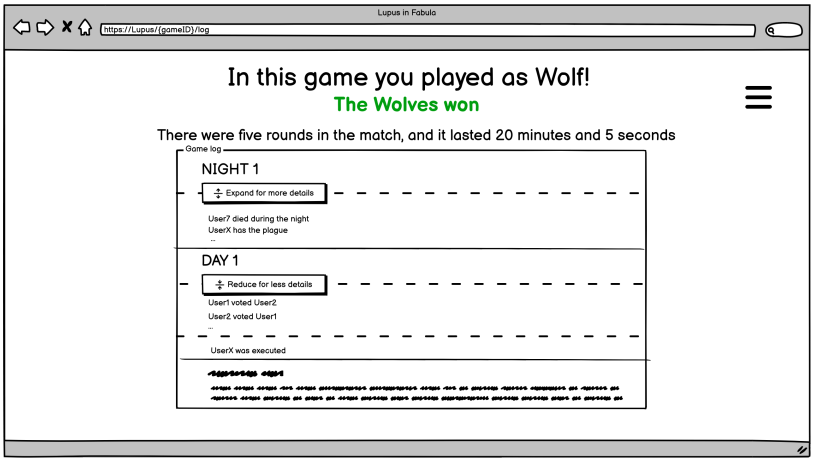


Figure 10: Log page

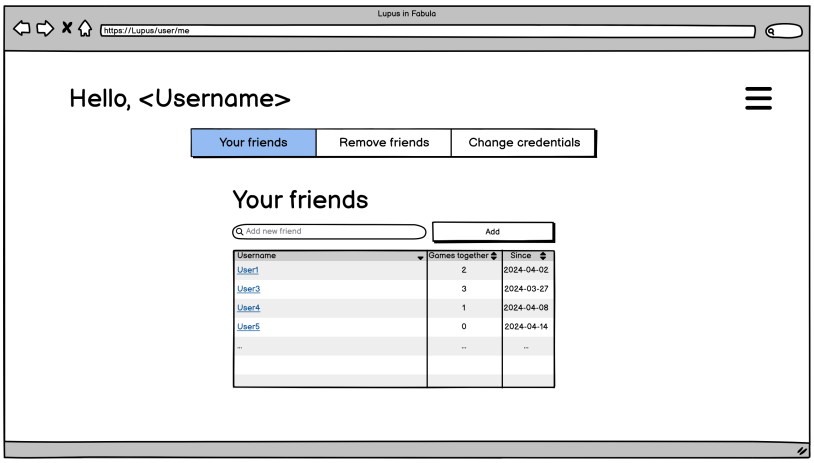


Figure 11: User page

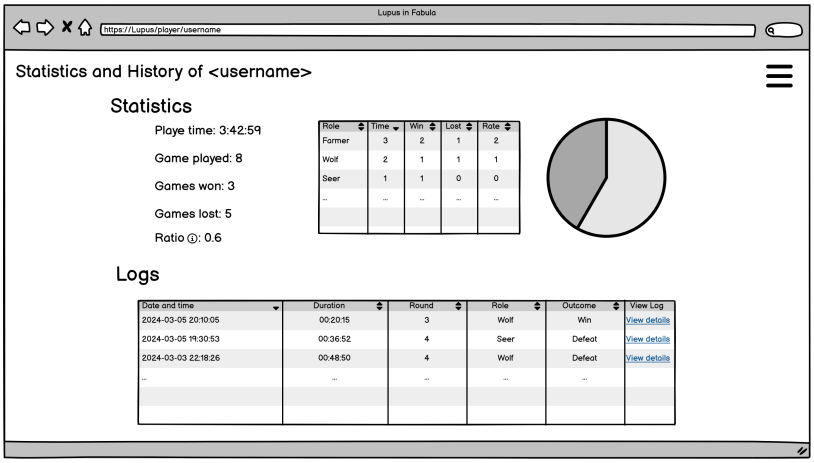


Figure 12: Statistics page

5 Business Logic Layer

5.1 Class Diagram

This section describes the various classes used to manage the two REST services ListFriendsRR and GamePlayerRR.

5.1.1 ListFriendsRR

Figure 13: List friend diagram

The class ListFriendsRR extends the AbstractRR class, which represents the basic functions for every REST resource; ListFriendsRR implements the doServe() function, which is used to handle the request to view all the friends of a particular user. This REST resource communicates with the database and retrieves the list of all friends using the ListFriendsDAO class, which extends AbstractDAO; ListFriendsDAO implements the doAccess() function, which is used to access the database and execute the query to retrieve the list of all friends of the user. This REST resource utilizes error codes implemented through the ErrorCode class, which implements functions like toJSON(), useful for converting errors into a JSON file, and valueOf(string s), useful for returning the value of a specific field passed as input. This REST resource utilizes LogContext to manage logs within the server, implementing various functions to set the values associated with logs (user, IP, game, action). This REST resource also utilizes "friend" to manage various instances of friends saved in the database, implementing toJSON() functions that create a JSON file representing an object, and fromJSON() functions that create a list of friends from a JSON file. These functions are inherited from the AbstractResource class, which represents the abstract resource. Additionally, this REST resource utilizes "Message" to manage all the various messages returned by the various REST resources, implementing toJSON() functions that create a JSON file representing an object. These functions are also inherited from the AbstractResource class. Finally, access to this REST resource is managed by the UserFilter filter, which implements filtering logic for user login and implements the doFilter() function to execute this check.

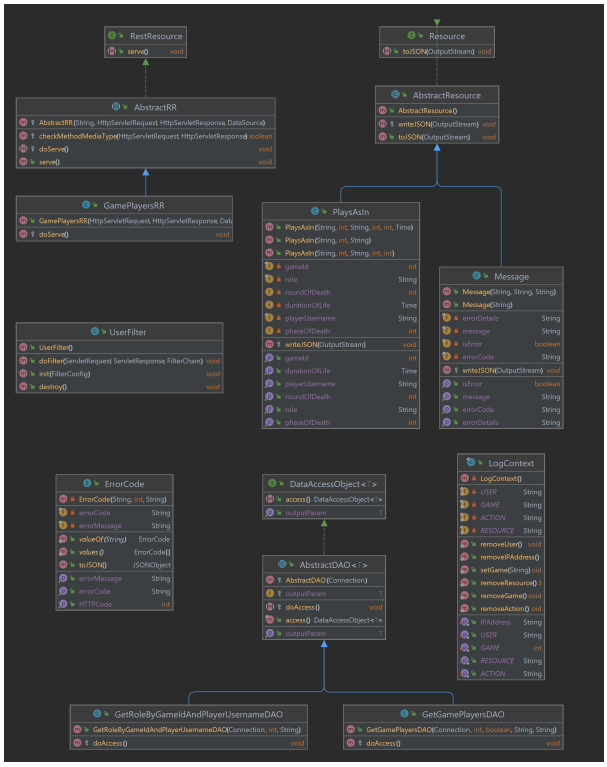


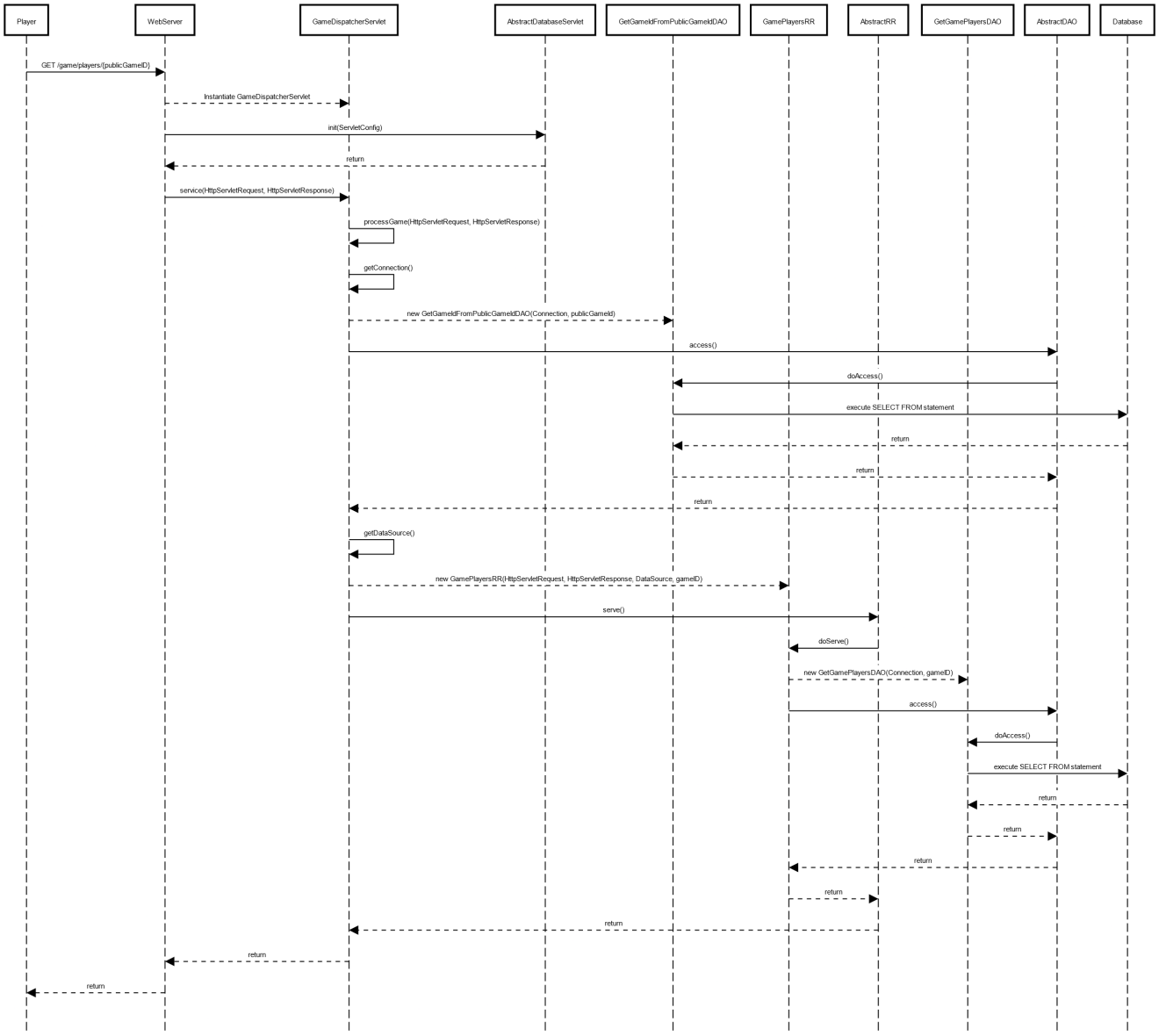
Figure 14: List friend diagram

5.1.2 GamePlayerRR

The class GamePlayerRR extends the AbstractRR class, which represents the basic functions for every REST resource; GamePlayerRR implements the doServe() function, which is used to handle the request to get all player in game with a specific role. This REST resource communicates with the database and retrieves the list of all role associated with a user in a game using the GetRoleByGameIdAndPlayerUsernameDAO class, which extends AbstractDAO; GetRoleByGameIdAndPlayerUsernameDAO implements the doAccess() function, which is used to access the database and execute the query to retrieve the list of role This REST resource communicates with the database and retrieves the list of all player in a game using the GetGamePlayersDAO class, which extends AbstractDAO; GetGamePlayersDAO implements the doAccess() function, which is used to access the database and execute the query to retrieve the list of all player in a game. This REST resource also utilizes "PlayAsIn" to manage various instances of PlayAsIn saved in the database, implementing toJSON() functions that create a JSON file representing an object. These functions are inherited from the AbstractResource class, which represents the abstract resource. Additionally, this REST resource utilizes "Message" to manage all the various messages returned by the various REST resources, implementing toJSON() functions that create a JSON file representing an object. These functions are also inherited from the AbstractResource class. This REST resource utilizes error codes implemented through the ErrorCode class, which implements functions like toJSON(), useful for converting errors into a JSON file, and valueOf(string s), useful for returning the value of a specific field passed as input. This REST resource utilizes LogContext to manage logs within the server, implementing various functions to set the values associated with logs (user, IP, game, action). Finally, access to this REST resource is managed by the UserFilter filter, which implements filtering logic for user login and implements the doFilter() function to execute this check.

5.2 Sequence Diagram

Figure 15: Sequence diagram to retrieve a list of players and roles



Here we report the sequence diagram to retrieve the list of players and roles that are playing a match identified by a public game ID. The player issues a GET request to the web server at the following URI: /game/players/{publicGameId}. The web server instantiates the GameDispatcherServlet which calls its processGame() method, passing the HttpServletRequest and HttpServletResponse.
Firstly, the control is passed to the GetGameIdFromPublicGameIdDAO which receives as arguments the connection (defined in the AbstractDatabaseServlet extended by the GameDispatcherServlet) and the publicGameID taken from the URI. The GetGameIdFromPublicGameIdDAO extends the AbstractDAO and contacts the Database Server which executes the SQL statement to retrieve the private game ID given the public game ID. If some errors occur, the control is returned to the SessionServlet and a new Message object is created.
Then, a rest resource called GamePlayersRR is instantiated. This resource takes in input HttpServletRequest,

HttpServletResponse, the data source and the private game ID. This resource proceeds to call GamePlayersDAO which is the DAO that actually returns the list of players and their respective roles. It works similarly to GetGameIdFromPublicGameIdDAO.
Finally, the list formatted in JSON is returned to the user.

5.3 REST API Summary

Filters:
L (logged in) $->$ the user must be logged in
M (master) $->$ the player must be Master in game={gameID}
When {gameID} is mentioned it refers to an automatically generated string (e.g. wolf-seer-farmer) which uniquely identifies a match within the web application. {username} refers to the username of a player.

URI	Method	Description	Filter
/signup	POST	Allow to sign up a new user with his credentials	
/login	POST	Allows a user to login using his credentials	
/logout	POST	Allows a logged in user to logout	L
/user/{username}	GET	Returns a brief summary about a player. If he's logged in, it shows more information	L
/user/{username}/logs	GET	Logs of matches played by such player	L
/user/{username}/statistic	GET	Returns player statistics	L
/user/me	GET	Returns information about logged in user	L
/user/me	PUT	Updates logged in user information (e.g. password)	L
/user/me	DELETE	Deletes logged in user's account	L
/user/me/friend	GET	Returns friends' list of logged in user	L
/user/me/friend	POST	Adds a user as a friend of logged in user	L
/user/me/friend	DELETE	Deletes a user inside friends' list of logged in user	L
/rules	GET	Returns rules of Lupus in Fabula	
/game/settings	GET	Returns game settings (e.g. minimum number of wolves)	L
/game/settings	PUT	Receives game settings and then inserts them into the database. Random roles are assigned to players as well	L
/game/status/{gameID}	GET	Returns general information about a match	L
/game/players/{gameID}	GET	Returns list of players who are playing/who played a match identified by gameID. For each player, it returns his role if and only if the logged in player has the right to see it	L
/game/players/{gameID}/master	GET	Returns list of players who are playing/who played a match identified by gameID	M

/game/log/{gameID}	GET	Returns logs of a match. If the match is still in play, it shows only public information. When the match is finished, it shows all the information (e.g who killed who)	L
/game/log/{gameID}/master	GET	Returns logs of a match, showing all kinds of information even if the match isn't finished	M
/game/actions/{gameID}	GET	Returns a list of actions that a player can execute	M
/game/actions/{gameID}	POST	After players execute actions during a phase (day or night), such actions are inserted into the database updating the logs	M

Table 3: Describe in this table your REST API

5.4 REST Error Codes

5.4.1 Login or edit credentials

Error Code	HTTP Status Code	Description
EUSR1	400 - BAD_REQUEST	Error throw when one or more input fields are empty.
EUSR2	400 - BAD_REQUEST	Error throw when the username is in an invalid format.
EUSR3	400 - BAD_REQUEST	Error throw when the email is in an invalid format.
EUSR4	400 - BAD_REQUEST	Error throw when the password is in an invalid format.
EUSR5	400 - BAD_REQUEST	Error throw when the passwords do not match.
EUSR6	409 - CONFLICT	Error throw when the username has already used
EUSR7	409 - CONFLICT	Error throw when the email has already used
EUSR8	400 - BAD_REQUEST	Error throw when the submitted credentials are wrong
EUSR9	404 - NOT_FOUND	Error throw when the user isn't found.
EUSR10	409 - CONFLICT	Error throw when update has failed.

Table 4: Login or edit credentials error code

5.4.2 Game

Error Code	HTTP Status Code	Description
EGME1	404 - NOT_FOUND	Error throw when one or more players does not exist.
EGME2	409 - CONFLICT	Error throw when one or more players are already in a game.
EGME3	409 - CONFLICT	Error throw when the gamemaster is already in a game.
EGME4	404 - NOT_FOUND	Error throw when one or more roles does not exist.
EGME5	400 - BAD_REQUEST	Error throw when the number of players entered does not correspond to the number of roles.
EGME6	400 - BAD_REQUEST	Error throw when the number of players isn't enough to play.

EGME7	404 - NOT_FOUND	Error throw when the parameter does not exist.
EGME8	400 - BAD_REQUEST	Error throw when there is an Invalid role with max cardinality.
EGME9	404 - NOT_FOUND	Error throw when the game isn't found
EGME10	409 - CONFLICT	Error throw when the game is over and new actions are sent

Table 5: Game error code

5.4.3 Friend

Error Code	HTTP Status Code	Description
EFRN1	409 - CONFLICT	Error throw when the friend is already in the list.
EFRN2	400 - BAD_REQUEST	Error throw when the friend is not in the list.

Table 6: Friend error code

5.4.4 Game logs

Error Code	HTTP Status Code	Description
EGLN1	404 - NOT_FOUND	Error throw when the logs aren't found.

Table 7: Game logs error code

5.4.5 Invalid data

Error Code	HTTP Status Code	Description
EJSN1	400 - BAD_REQUEST	Error throw when the format for JSON is invalid.

Table 8: invalid data error code

5.4.6 Session

Error Code	HTTP Status Code	Description
ESES1	403 - FORBIDDEN	Error throw when the account isn't logged in.
ESES2	403 - FORBIDDEN	Error throw when the account isn't a gamemaster.
ESES3	404 - NOT_FOUND	Error throw when the game doesn't exist.
ESES4	409 - CONFLICT	Error throw when the player isn't the gamemaster of this game.

Table 9: session error code

5.4.7 Dispatcher

Error Code	HTTP Status Code	Description
EDSP1	404 - NOT_FOUND	Error throw when there is an unknown resource requested.

EDSP2	405 - METHOD_NOT_ALLOWED	Error throw when the method is not allowed.

Table 10: Session error code

5.4.8 Actions

Error Code	HTTP Status Code	Description
EACT1	400 - BAD_REQUEST	Error throw when a null action is requested.
EACT2	400 - BAD_REQUEST	Error throw when an invalid target is requested.
EACT3	400 - BAD_REQUEST	Error throw when a player is not in a game.
EACT4	400 - BAD_REQUEST	Error throw when a role is not corresponding to that player.
EACT5	409 - CONFLICT	Error throw when the player is dead.
EACT6	409 - CONFLICT	Error throw when there are too many wolves actions.
EACT7	400 - BAD_REQUEST	Error throw when an invalid action is requested.
EACT9	400 - BAD_REQUEST	Error throw when the list of vote is not valid.
EACT10	400 - BAD_REQUEST	Error throw when the target of ballot vote is not valid.

Table 11: Actions error code

5.4.9 Errors

Error Code	HTTP Status Code	Description
EDTB1	500 - INTERNAL_SERVER_ERROR	Error throw when there is a internal database error.
EINT1	500 - INTERNAL_SERVER_ERROR	Error throw when there is a internal error.
EINT2	500 - INTERNAL_SERVER_ERROR	Error throw when there is a Null object internal error.

Table 12: Errors error code

5.5 REST API Details

We report three different resource types handled by the application.

Game Logs

This endpoint allows master and regular user to see all logs associated to a specific game.

User

- URL: `lupus/game/logs/{gameId}`
- Method: `GET`
- URL Parameters:
  gameId: String, public ID of game
- Data Parameters: None
- Success Response:
  Code: 200
  Content: Json of list of class Action

- Error Response:
  Code: 404
  Content:
  {
  "message":
  {
  "message": "Invalid game, the game 'dorky-ds-explorer' doesn't exists.",
  "error-code": "EGME9",
  "error-details": "Invalid game."
  }
  }
  Code: 403
  Content:
  {
  "message":
  {
  "message":"Authentication required, not logged in",
  "error-code":"ESES1",
  "error-details":"Account not logged in."
  }
  }

Master

- URL: `lupus/game/logs/{gameId}/master`
- Method: `GET`
- URL Parameters:
  gameId: String, public ID of game
- Data Parameters: None
- Success Response:
  Code: 200
  Content: Json of list of class Action
- Error Response:

  Code: 404
  Content:
  {
  "message":
  {
  "message": "Invalid game, the game {publicGameId} doesn't exists.",
  "error-code": "EGME9",
  "error-details": "Game doesn't exists."
  }
  } **Code:** 403
  Content:
  {
  "message":

{
"message":"Authentication required, not logged in",
"error-code":"ESES1",
"error-details":"Account not logged in."
}
}
Code: 403
Content:
{
"message":
{
"message":"Trying to authenticate the currentPlayer {username} as a gamemaster in the game {publicGameId}",
"error-code":"ESES2",
"error-details":"The account is not a gamemaster."
}
}

Game Settings

This endpoint allows to set up a game by sending the list of players that will play and a list of roles that players will embody.

- URL: `lupus/game/settings`
- Method: `POST`
- URL Parameters: None
- Data Parameters:
  {
  "player": [{"username": value1}, {"username":value2}, ...],
  "roleCardinality": [{"role": "wolf", cardinality: 2}, {"role": "jester", cardinality: 1}, ...]
  }
  Player is an array that contains a set of players that will participate to the game. roleCardinality is an array that contains a set of roles and cardinalities for such roles.
- Success Response:
  Code: 201
  Content:
  {
  "game": {
  "id": 4,
  "public-ID": "explorer-puppy-kamikaze",
  "start": "2024-04-29",
  "game-duration": "",
  "who-win": -1,
  "rounds": 0,
  "phase": 0,
  "subphase": 0
  }

}
The server returns a JSON representation of the newly created game.

- Error Response:
Code: 404
Content:

```
{
    "message":
    {
        "message": "PLAYER userX does not exist",
        "error-code": "EGME1",
        "error-details": "One or more players does not exist."
    }
}
```

When: If a specified player doesn't exist in the database.

Code: 403
Content:

```
{
    "message":
    {
        "message":"Authentication required, not logged in",
        "error-code":"ESES1",
        "error-details":"Account not logged in."
    }
}
```


Code: 400
Content:

```
{
    "message":
    {
        "message": "Invalid JSON format",
        "error-code": "EJSN1",
        "error-details": "Unable to parse JSON: no roleCardinality object found."
    }
}
```

When: If the JSON data parameters are wrongly formatted.

Code: 404
Content:

```
{
    "message":
    {
        "message": "ROLE cat does not exist",
        "error-code": "EGME4",
        "error-details": "One or more roles does not exist."
    }
}
```

When: If one or more roles specified in the request do not exist.

Code: 400
Content:

```
{
    "message":
    {
        "message": "Player number 8 does not match the number of roles 7",
        "error-code": "EGME5",
        "error-details": "Number of players entered does not correspond to the number of roles."
    }
}
```

When: If the number of players isn't equal to the number of roles.

Code: 400
Content:

```
{
    "message":
    {
        "message": "Invalid roles cardinality",
        "error-code": "EGME8",
        "error-details": "Invalid role max cardinality."
    }
}
```

When: If the cardinality of a role exceeds the maximum cardinality specified into the "Role" table.

Code: 409
Content:

```
{
    "message":
    {
        "message": "PLAYER allRole1 is already in a game",
        "error-code": "EGME2",
        "error-details": "One or more players are already in a game."
    }
}
```

When: If a specified player is already playing another game.

Update user's password

This endpoint allows a user to change his/her password.

- URL: `lupus/user/me`
- Method: `PUT`
- URL Parameters: None
- Data Parameters:
  {

```
    "userUpdate": {
        "oldPassword": value1,
        "newPassword": value2,
        "repeatNewPassword": value2
    }
}
```

- Success Response:
  Code: 200
  Content:

```
{
    "message": {
        "message": "User has successfully updated the password"
    }
}
```

- Error Response:
  Code: 403
  Content:

```
{
    "message":
    {
        "message":"Authentication required, not logged in",
        "error-code":"ESES1",
        "error-details":"Account not logged in."
    }
}
```

  Code: 409
  Content:

```
{
    "message":
    {
        "message":"User esempio failed to update the password",
        "error-code":"EUSR10",
        "error-details":"Update failed."
    }
}
```

  When: If the UPDATE query returns an error. For instance, if the old password is not equal to the real old password of the user.

  Code: 400
  Content:

```
{
    "message":
    {
        "message":"New password and repeatNewPassword do not match",
        "error-code":"EUSR5",
```

```
            "error-details":"Passwords do not match."
        }
}
```

6 Group Members Contribution

Busato Nicola Wrote Chapter 4 and designed the mockups. Designed REST API. Set up project and Docker configuration for a Tomcat and a PostgreSQL container. Implemented servlets for user login, signup, and logout features, complete with their corresponding DAOs and classes. Established GameDispatcherServlet and GameMasterFilter. Developed GameActionsGetRR to manage GET requests for /lupus/game/actions/gameID, including the necessary DAOs and classes. Formulated GameSettingsGetRR and GameSettingsPostRR to handle GET and POST requests for /lupus/game/settings, integrating their associated DAOs and classes. Assisted in the development of GameActionsPostRR.

Cini Jacopo Helped to develop GameDispatcherServlet, which then uses two REST APIs: /lupus/game/logs/{gameID} and /lupus/game/logs/{gameID}/master. Developed GameLog rest resource, its respective DAO which is GetActionByIdGameDAO and its resource, the class Action. Contribuited to write section 5 of this document.

Gusella Michele Helped to develop GameDispatcherServlet, which then uses these REST APIs: /game/players/{gameID}, /game/players/{gameID}/master and /game/status/{gameID}. Developed GamePlayers rest resource and GameStatus rest resource, their respective DAO which are GetGamePlayersDAO and GetGameByGameIdDAO, and their resources, the classes PlaysAsIn and Game. Contribuited to write section 5 of this document.

Miele Riccardo With Nicola Busato has brainstormed ideas for the project and sketched out an outline to start developing the project. Has set up the database by creating the ER scheme, the logic scheme and the implementation in SQL. Has written the chapter 3 explaining all the entities and the relations contained in the ER and logic schemes. Has developed RR for statistics and logs (with DAOs and the necessary resources). Has revised and fixed the chapter 2.

Momesso Jacopo Developed the UserDispatcher and UserFilter, the RestResources related to /user/me and /user/{username}, the RulesServlet for handling the rules page and the roles of the game, the GameActionsPostRR, in particular the night phase controls for each action and for each role, and all the relatives DAO for querying the database. Wrote chapter 1 of this document with the goals of the project and the rules of the game Lupus in Fabula, and created the cards of each role that we decided to implement.

Pozzo Nicola Developed the /user/me/friend in UserDispatcher with the related RestResource and DAOs and the day phase in the GameActionPostRR with all the controls for the voting part. Contribuited to write the chapter 2 with the main functionalities of the webapp.