Group 15

Charlotte Eder, Group Leader, charlotte.eder@uzh.ch, 17-826-090  
Minh Phuong Vu, minjphuong.vu@uzh.ch, 18-730-788  
Kai Mitiyamulle Arachchige, kai.mitiyamullearachchige@uzh.ch, 18-705-434  
Jordan Cedeño, jordan.cedenocedeno@uzh.ch, 14-713-440  
Raphael Haemmerli, raphael.haemmerli@uzh.ch, 17-714-593

Just One

SoPra FS2020 Milestone 4

## Summary of our implementation of Just One

In general, we tried to stay as close as possible to the original version of Just One. First, a new player must register an account with a username and a password. Then, he or she logs in with this account. After that, the player has access to a lobby overview, where he or she can see all the games that have already been created.

On this ***overview page*** the player can either join a ***lobby***, create a new lobby or have a look at the leader board. The leader board ranks all players that have ever played JustOne according to the total amount of points they gathered in these playthroughs. The total amount of played games is also shown in order for people to know whether the top players are actually skilled or are have just played the game a thousand times. When the player clicks on the button for creating a lobby, he or she can choose a name for the game, enter the maximum number of players, the number of ***bots*** (***angels*** = helpful players, ***devils*** = sabotaging players) and set an optional password for joining the game. After the player clicks on create game, the game is shown to the players in the lobby. These players can now join the lobby by clicking on “choose game” and then “join lobby”. They have to enter a password, if one was set during creation of the game.

When the player that created the game (the host) clicks on “start game” all the players that have joined the lobby get redirected to the actual game page (in-game page -> ***active game***). Here they can play a game of Just One. This ***game*** consists of ***turns***, which have ***four phases***. These are as follows: “choose a mystery word”, “give clues”, “give a guess” and “word reveal”.

If a player is the ***active player***, he or she does not see the words on the card in the middle of the table. Instead, during “choose a mystery word”, he or she must choose a word by clicking on its location on the card. Then the active player waits until all ***passive players*** and ***bots*** have submitted their clues. In the third phase “give a guess”, the valid clues are shown to the active player and based on these clues the player must guess the original mystery word in the input box. If his or her guess was right all players get a fixed amount of base points. Depending on how fast they were to submit their clue or guess they might get extra points. If some ***passive players*** gave invalid clues, they will be deducted some points instead of getting extra points. If the active player guesses incorrectly an extra card is discarded from the card-stack and no one gets base points. ***Passive players*** that gave invalid clues are penalized by getting minus points. The ***active player*** never gets penalized for an incorrect guess, since he is only as good as the clues provided to him, which in an extreme case might be none (, because all are invalid).

All the other players are the ***passive players*** during that turn. They can see the card and after the phase “choose a mystery word”, all the words are crossed out except the selected one. During “give clues”, they write a helpful clue into their input box and press the button to the right in order to submit their clue. During the “give a guess” phase they wait for the ***active player*** to submit their guess. Next follows “word reveal”, when the ***passive players*** get shown the submitted guess by the ***active player*** and if it matches the mystery word.

When the card stack is empty, the game finishes and an end screen is shown. After that, the players get redirected to the lobby overview.

Some Remarks about the ***bots***: The bots are only capable of giving clues, but cannot guess the mystery word, meaning they only mimic ***passive players*** but never the ***active player***. Regarding their clues: The datamuse API used for producing the clues of the bots delivers relatively sophisticated clues. Sometimes a player might require a lot of domain specific knowledge to get the connection between the clues given by a bot and the mystery word. This however adds from our point of view an interesting level of difficulty to the game. An example would be “TARDIS” for the mystery word “doctor”, where TARDIS is the telephone-cabin like machine from the television series “doctor who”. While this clue might seem obvious to some people, it might also be quite a difficult clue to decipher for people not very familiar with “doctor who”.  
Also important to note is that the ***bots*** do not get points during the game. This is an intentional decision because we wanted the competitive aspect of our game to focus on the competition between human players. ***Bots*** are there to make the game more interesting, diverse and also to make it possible to play by yourself.

### New Remarks

Some remarks about our implementation: We have decided not to implement two aspects of JustOne for our game:

* Skipping a card in phase 4: We have decided to not make this option available in order to raise the stakes for the players. If their clues are really bad, the active player doesn’t just have the option to skip the card and avoid losing two cards. We hope this to be an incentive to give good clues.
* Skipping to phase 4 (from phase 2) if all clues are ‘invalid’: This feature represents an edge case, and we think that implementing that edge case would just confuse people if it ever occurred. Plus, we think it is funny, that the active player has to guess the word, with nothing to help him, we want him to be confronted by the failure of the passive players.
* Passive players can ask the active player to choose a different number: We think this would make it too easy for people to bend the game to their will and force the mystery word chosen to be one that can be easily guessed.
* A correctly guessed card being removed if the guess of the last round is wrong: We do not think that this feature makes any sense. All rounds are equal in their difficulty and we do not see the benefit of punishing all players if the guess of the last round is wrong. We want the punishment to be equal for all rounds.

## What could be implemented in the future?

The ReadMe of both the client and the server offer some ideas for new feature implementations. A few things that could be done are adding the possibility of bots being an active player, the implementation of a spectator mode or the option for players to provide their customized rules for a game (especially the last idea would increase the value of repeated playthroughs greatly).

Another aspect of our game that has the potential to be fleshed out is the sound environment. At the moment we only have three sounds that are played during the in-game (countdown, wrong guess, correct guess). Future developers could implement more feedback sounds, like a sound every time a clue is submitted. Background music that plays throughout the game is also something that would make the whole experience feel a bit more exciting, especially when waiting for other players to perform an action.

Thirdly, it would also be nice that the phases and the timer, which are now implemented on the server, could be even further integrated into the client. This would make the game even more synchronous for all players and it would open the possibility to also implement a client for other platforms like mobile systems.

## Challenges and Strengths during Planning, Developing and collaborating remotely

### Planning

During the first sprint, the backend team realised that the conceptual model of the class diagram from milestone 2 was lacking a lot of important classes. Also, it was not built on the idea of a client invoking methods but more on a main method that calls methods like it had been the case during the “Software Construction” course. Thirdly, the way entities must be designed in order to save them to a database was not considered either. This posed significant problems when it came to the planning of how our web application has to implemented since it was not clear which components have to be built and how they should work together.

This challenge was quickly overcome though. The backend team created a new class diagram that represented the system as it should be implemented. After that, the developing process corresponded roughly to how it was planned. This problem caused a short delay and could have been prevented with proper planning from the beginning. However, it taught us the value of clearly planning the design of high-level components and the systems overall structure before any code is written.

Development

Server:

The most difficult part during developing was to gain some deeper knowledge on using an external API with Java, JPA and JUnit. Especially the implementation of the first new service class and first repository was tough, but after some initial hurdles a lot of the further classes could be implemented similarly to existing ones, once we had acquired the necessary know-how.

Client:

Here difficulties arose mainly from testing. Especially in the beginning it was difficult to test certain functions since they were not implemented on the server yet. Also, some functionalities could only be tested after other functionalities had been implemented. For example, the creation of a game had to work properly in order to test some functionalities on that game.

What went smoothly was mainly the design part since a lot of styling could be copied from Figma and reused for a lot of pages.

What helped all the group members was the introduction of a rest specification sheet (Appendix A). This file was a key factor of the successful communication between the frontend and backend team. It shows an overview of all requests that can be made, in which situation they can be used and whether they already work or not. With this simple table, non-functioning, newly needed or changed requests could be communicated clearly.

Splitting up of the whole team into a backend and frontend team was further helpful since it allowed us to specialize our individual knowledge to a higher degree than if we were working on both parts simultaneously with surface level knowledge instead. This also made it way clearer who to ask specific questions to, when certain types of problems arose since each person was a “specialist” of a specific domain. For example, Jordan could be asked for JPA-related questions or Kai for React-questions.

### Collaborating remotely

After some initial lethargy, the team decided that they would need a way of working together remotely which allowed them to be nearly as productive as if they could meet at university. They decided to do all communication over “Discord”[[1]](#footnote-1). On this platform, a server with several voice channels (meetings, frontend and backend) and text channels was created with the only purpose of working on Sopra.

We also introduced a daily scrum meeting at 13.00, where first each group member would quickly (max. two minutes) explain on which tasks they worked on during the last day. Then, questions would be discussed that concerned the whole sopra group. Next, it was made sure that everyone knew what he or she would have to do until the next scrum meeting. After that, the frontend and backend team would usually split up and discuss more in-depth questions in a frontend and backend voice-channel on the server.

## Reflexion on Milestone 4

### New features

During milestone 4 we implemented the last major feature for our application, a chat in the lobby of a game. Due to the knowledge we gained in the third milestone, this was done fairly quickly and hasn’t caused any problems. At times during milestone three we were unsure whether we would have the time to implements this feature since it looked like our workload was big enough on its own and the implementation of a chat isn’t mandated by the course. However, since most major features were fully implemented in the third milestone, we decided to do it and are very happy with the result.

A minor feature that was implemented is the opportunity for a player to leave an ongoing game by clicking on a button. In that case the user would be replaced by a bot and the game would continue as planned, the same action is triggered if a user decides to leave a game by closing the browser. This feature might not be as “sexy” as a chat, but we think it is a great improvement to our system in terms of robustness. Moreover, this feature is based on two changes on the server. Firstly, the Logic Server, e. g. the in-game component was refactored so that it uses now the state pattern with a state for each game phase. This change makes the in-game more robust to requests which are called during the wrong game phase. Also, it makes it a lot easier to extend the game which we already noticed when implementing the timer on the server. This timer has three functions: it notices when a game phase should already have ended and finishes it automatically (depending on the phase differently -> state pattern). It notifies the frontend about the time the timer has started and how long a game phase takes and it notices when a player has been absent for more than 30 seconds and replaces this player at the end of a game round with a bot.

### Enhancing usability

In the feedback we received for milestone 3 (by the SOPRA team and group 14) it was noted that our user interface was not very intuitive. For the frontend team, this was one of the main focus points in the fourth milestone. In order to improve this aspect, the following steps were taken:

* Player HUD of a user is fixed in the middle of the screen, below the table
* The mystery word is now chosen by clicking on a word on the card instead of entering a number between one and five
* Input fields and buttons of other players were disabled
* Input can only be given where it is needed (e. g. passive players can only enter input in the “give clue” phase)
* At the start of a new phase, an animation is played across the screen that tells a user what to do
* The button is marked with a click icon
* The player order has been adjusted such that the active player role changes clockwise.
* Clues can be sent by pressing enter
* Pulsating animation shows where input has to be given
* Sounds are played to indicate whether a mystery word was guessed correctly

The tutorial provided by us has also been vastly improved in order to help players understand the game. We think that the interaction that is offered during the tutorial also helps the players a great deal in understanding the game flow.

### Timer synchronization

Before starting milestone 4, the timer was handled exclusively in the frontend. If the page would now be refreshed by a user, the timer would also restart again. This would allow a user to cheat in our game so it was decided to implement a timer in the backend too, so it could be synchronized with the frontend. This feature may not be very visible but just as the option to leave game, it increased the robustness of our application greatly.

### Refactoring and fixing bugs

Both the server and client have undergone a refactoring process. In the client, the main goal was to lower the duplicated amount of code (2. Mai: 19.7% -> 21. Mai: 1.7%) and extract some parts of the “in game” file and make separate components out of them (at the beginning of the development phase, this possibility did not cross our minds because of our non-existing experience programming in React, Javascript, HTML and CSS). SonarCloud has been a great help during the refactoring process when it came to duplicated code.

The main goal of the refactoring in the server was the implementation of the state pattern. The state pattern is responsible for the timer synchronisation, made the system overall much more open for extension and the intense use of flags has also decreased tremendously. Several edge cases that have not been accounted for before, are now also taken into consideration thanks to the state pattern.

Additionally, several minor bugs have been fixed in milestone 4 that have not been noticed before. Here are a few examples:

* After completing a game and players were redirected to the lobby overview, they couldn’t visit any other page for 30 seconds. They would always be redirected to the lobby overview
* It was possible to join an ongoing game through the lobby overview -> handle by removing each game that has been started from the lobby overview
* Upon refreshing the “in game” page in phase 4 (“word reveal”) a new card was drawn

Fixing bugs was probably the most tedious and nerve-racking task but its importance cannot be denied.

## Testing

### Representivity of chosen Tests

For us, these tests are representative of all our tests since especially the unit tests are focused on catching edge case behaviour and checking that a method works in the way it was intended to. We wrote the tests right after having implemented a new method so their main goal was to show to the developer that he or she caught all the edge cases and would not push a method that was not working properly.

We usually try to cover as much ground as possible when writing tests, so that you can run all the tests after implementing any new feature and test if it breaks pre-existing features. This approach has proven vital for development as to avoid breaking the build, which would not only hinder the backend developers greatly, but even more so the frontend developers that depend on a working build to be able to advance with their tasks and test their own code.

### Overall Test Coverage

Our overall test coverage right now sits at slightly over 86% test coverage. Our goal is to maintain that level of test coverage in the following sprints or even increase it if time allows.

### Beta Testing

In order to improve the usability of our application, we decided to let family and friends play our game and document what issues they would encounter. The important part about this kind of testing was to not only write down the problems the testers encountered but also to analyse this data, write down the problems that occurred multiple times and then think about a solution for these problems. After that, the solution would have be to be tested and reanalysed again.

In our case, we collected the usability issues in a Google Docs. After that, we discussed for each point how to fix this issue and who would be responsible for fixing it.

## Lessons learned

Looking back and reflecting on our project as a whole, we have learned several important lessons when it comes to software engineering, project planning and group coordination.

### Daily SCRUM meeting

The introduction of daily SCRUM meeting was one of the most important things we did. Even if they were short sometimes, they allowed us to get a quick overview of where we stand, discuss important points and ask questions. This was especially important due to us splitting into front- and backend groups.

Our approach also allowed us to quickly adapt to changing requirements. At first for example it was planned that the timer would be fully implemented in the front end, however after some time it became clear to use that for the purpose of synchronization the backend should also implement a timer. This was not planned in the beginning so we were unsure how this would go. However, the due to us meeting frequently we were very flexible when it came to the implementation, e. g. when it was unclear whether the front end should adjust the time formatting for the display or if the backend should already do that and give us a value that can be immediately displayed.

### Dividing the tasks among the group

Since all of us had no former experience with a lot of the tools used in the SOPRA course, we decided to split our group into a front- and backend team. This allowed us to fully concentrate on one aspect of the application. Although this has helped us a great deal it should be noted that the coordination between front and backend team cannot be neglected if this approach is taken. To integrate both teams in the development as a whole regular SCRUM meetings and up-to-date REST specifications that are clear to everyone are essential.

### Higher level design in frontend

Designing small components that focus on one aspect and can be reused is a way of programming in React that we embraced too late. The render method of our “in game” page for example contains several components that could be extracted. We did that for a few components like a message being displayed in the beginning of each phase or the timer, however we should have done that in the beginning in order to avoid technical debt, ensure evolvability and provide understandable code. This certainly had to do with the fact that we were new to React, JavaScript, HTML and CSS. We will certainly take this into consideration for future projects we will be working on.

### Diagrams backend

As already described in report 3, we encountered a lot of problems because we did not flash out the class diagram of the server enough thoroughly. Therefore, we will put more effort next time into learning about the technologies used and draw a more explicit diagram. Also, when projects get bigger, we might should think about the database design as well. Lastly, we also learned that it is important to apply the design principles also to the tests. When for example a lot of tests use an active game with three players, do not write this setup for every single test but put this creation method into an external class and use it as a helper for all the other tests. Otherwise, when there is for example a change in the way a game is set up, a lot of tests must be changed manually.

### Programming -> learn it by doing

All of us knew upfront that this web application will cost time, effort and a great degree of initiative. However, even though, or maybe exactly because this project has been completed at great expense, each and every one of our group learned more practically related things about software engineering than in other course so far. This goes to show that the courage to set high goals and openness to embrace new tools and frameworks can do a lot to improve one’s programming skills.

## Links:

**GitHub Group 15 folde**r: https://github.com/sopra-fs-20-group-15

**GitHub Client:** https://github.com/sopra-fs-20-group-15/client

**GitHub Serve**r: https://github.com/sopra-fs-20-group-15/server

**SonarQube:** https://sonarcloud.io/organizations/sopra-fs-20-group-15/projects

**Heroku client:** https://sopra-fs20-group-15-client.herokuapp.com/

**Heroku server:** https://sopra-fs20-group-15-server.herokuapp.com/

**Jira:** https://sealjira.ifi.uzh.ch/secure/RapidBoard.jspa?rapidView=46&view=planning.nodetail&issueLimit=100

# Appendix A

# Login, User Registration:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Mapping | Method | Parameter | Parameter Type | Status Code | Returned Value | Descripiton | Status |
| /register | POST | PlayerName  <String, Password> | Body | 201 | PlayerId<Long> | Register a player | Done |
| /login | PUT | PlayerName  <String, Password> | Body | 200 | - | Login a player | Done |
| /logout | PUT | id<long>, token<String> | Query | 200 | - | Logouts player | Done |

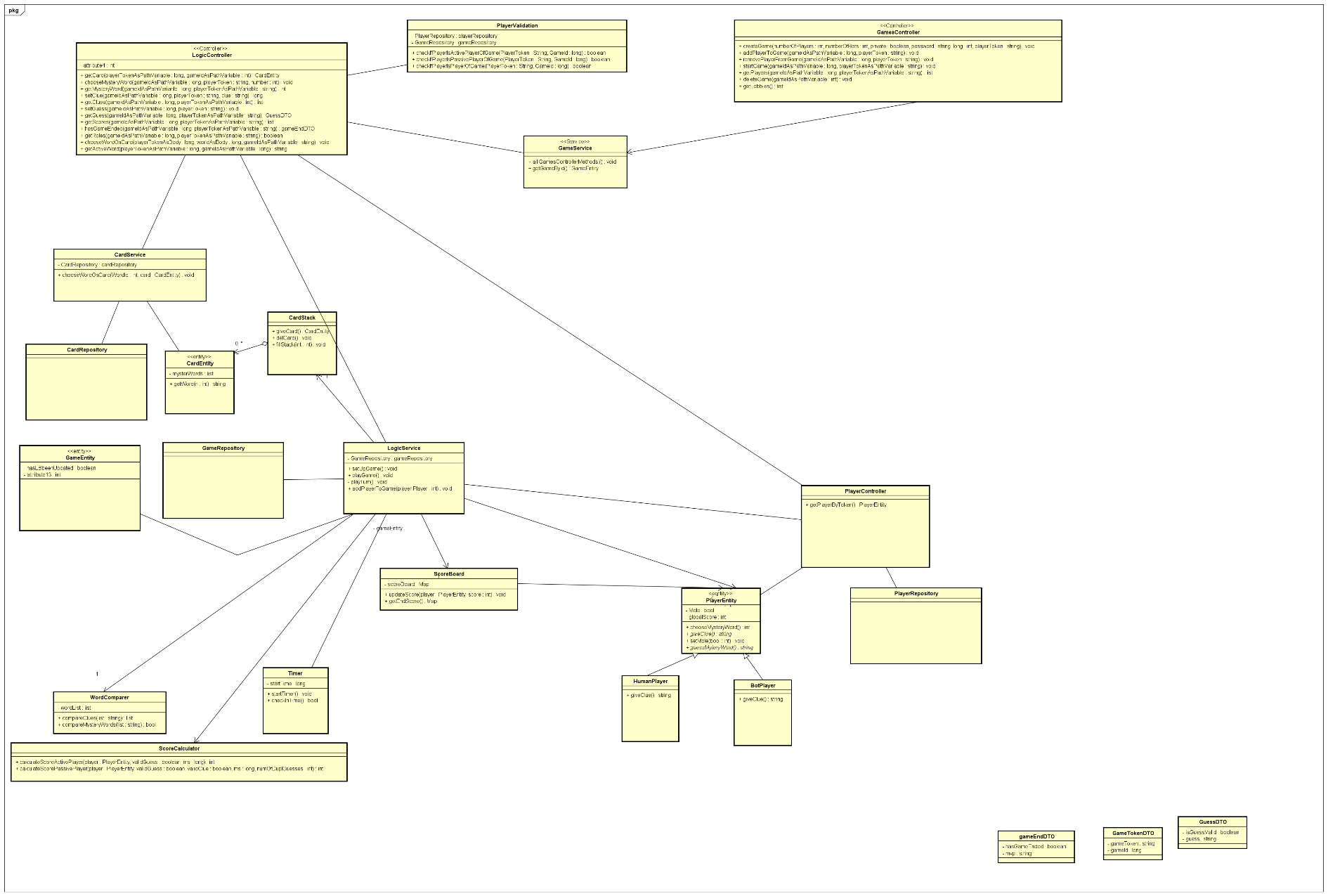
# Lobby

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Description | Mapping | Method | Parameter | Parameter Type | Status Code | Returned Value | Status |
| Create GameSetUpEntity | /games | POST | NumberOfPlayers<Long> NumberOfDevils<Long>  GameType<String(public or private)>  PlayerToken<String>  Password<String> (only if private game) | Body | 201 |  | Done |
| Join Player | /games/{gameId}/players | PUT | playerToken<String>  password<String> (if private) | Query | 200 |  | Done |
| Leave Player | /games/{gameId}/lobbies/players | PUT | playerId<long> | Body | 200 |  | Done |
| Delete GameSetUpEntity | /gameSetUps/{gameSetUpId} | DELETE | Body: PlayerToken<String> |  | 200 |  | Done |
| Overview all Lobbies | /games/lobbies | GET | - |  | 200 | List<LobbyOverviewGETDTO>  LobbyOverviewGetDTO:  String gameName  GameType gameType  Long numOfDesiredPlayers  Long numOfAngels;  Long numOfDevils;  Long numOfActualPlayers | Done |
| Overview specific Lobby | /games/lobbies/  {gameSetUpId}/{playerToken} | GET |  |  | 200 | LobbyGetDTO:  Long activeGameId  Long gameSetUpId  String gameName  String hostname  List<String> playerNames  Long numOfDesiredPlayers  Long numOfActualPlayers  Long numOfAngels  Long numOfDevils | Done |

# Game-Related: Play a whole turn

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Descripiton | Who? | Mapping | Method | Parameter | Parameter Type | Status Code | Returned Value | Phase | Status |
| Create Active Game | Host | /games/{gameSetUpId} | POST | playerToken<String> | Body | 201 | Long<id> | 0 | Done |
| Get Game Info -> Roles | All | /activeGames/{gameId} | GET |  | Query | 200 | Id<Long>, activePlayerName<String>, playerNames <List<Strings>>  passivePlayerNames<List<Str>> | 0 | Done |
| Initialize Turn | All | /games/{gameId}/initializations | PUT | playerToken<String> | Body | 200 | - | 1 | Done |
| Get Card | PassiveP | /games/{gameId}/cards/{playerToken} | GET | - | Query | 200 | List<Words> | 2 | Done |
| Get Amount of Cards | PassiveP | /games/{gameId}/cards/remainder/{playerToken} | GET | - | Query | 200 | CardsRemainingDTO:  Int cardsOnStack | ANY | Done |
| Choose MysteryWord | ActiveP | /games/{gameId}/  mysteryWord | PUT | wordId<Long> (1-5)  playerToken<String> | Body | 200 | - | 3 | Done |
| Get MysteryWord | PassiveP | /games/{gameId}/  mysteryWord/{playerToken} | GET | - | Query |  | Word<String> | 3 | Done |
| Set Clue | PassiveP | /games/{gameId}/clues | POST | Clue <String>  PlayerToken<String> | Body | 201 |  | 4 | Done |
| Get Clues (valid) | All | /games/{gameId}/clues/{playerToken} | GET | - | Query | 200 | ListofClues<String,String> (DTOList, playername, clue) | 4 | Done |
| Get Clue Players | All | /games/{gameId}/clues/players/{playerToken} | GET | - | Query | 200 | List<PlayerNameDTO>  PlayerNameDTO:  Sring playerName | 4 | Done |
| Set Guess | ActiveP | /games/{gameId}/guesses | POST | Guess <String>   * PlayerToken<String> | Body | 201 | - | 5 | Done |
| Get Guess | All | /games/{gameId}/guesses/{playerToken} | GET | * - | Query | 200 | Guess<String>  IsValid<Boolean> | 5 | Done |
| Game Stats | All | /games/{gameId}/statistics | GET | - | Query | 200 | List<GameStats>,  GameStats: <String playerName, int score, int placement, int correctlyGuessedMysteryWords> | 5 | Done |
| Has Game Ended | All | /games/{gameId}/ends/{playerToken} | GET | - | Query | 200 | HasEnded <Boolean> | 6 | Done |
| Delete Game | All | /games/{gameId} | DELETE | playerToken <String> | Body | 200 | - | 7 | Done |

# Appendix B



**1**

**3**

**4**

**2**



**1.)**



**2.)**



**3.)**



**4.)**

1. <https://discordapp.com/> [↑](#footnote-ref-1)