Distributed Tic-Tac-Toe

Team 07

Members

- Henrik Erpenbach (herpenba, henrik.erpenbach@smail.th-koeln.de)
- Jonas Fender (jfender, jonas.fender@smail.th-koeln.de)
- Johannes Hirth (jhirth, johannes.hirth@smail.th-koeln.de)
- Bastian Schetter (bschette, bastian.schetter@smail.th-koeln.de)

version from 2021-05-19, 15:37:39 +0200

Table of Contents

1. Outline

1.1. Vision

Our project will be a web application. The application will give users the opportunity to play a game of Tic-Tac-Toe against online opponents. For this goal, a user will be able to search for a match, either against a random opponent or against a friend by exchanging something like a match key. After a match is found, the players should be able to play Tic-Tac-Toe against each other. When the match ends the end result of the game should be written into a database for each player.

Technically, the matches need a bi-directional connection to each player, so that the server can speak to a player without the player sending a prior request. For database requests (e.g. player information) an additional REST-interface may be required.

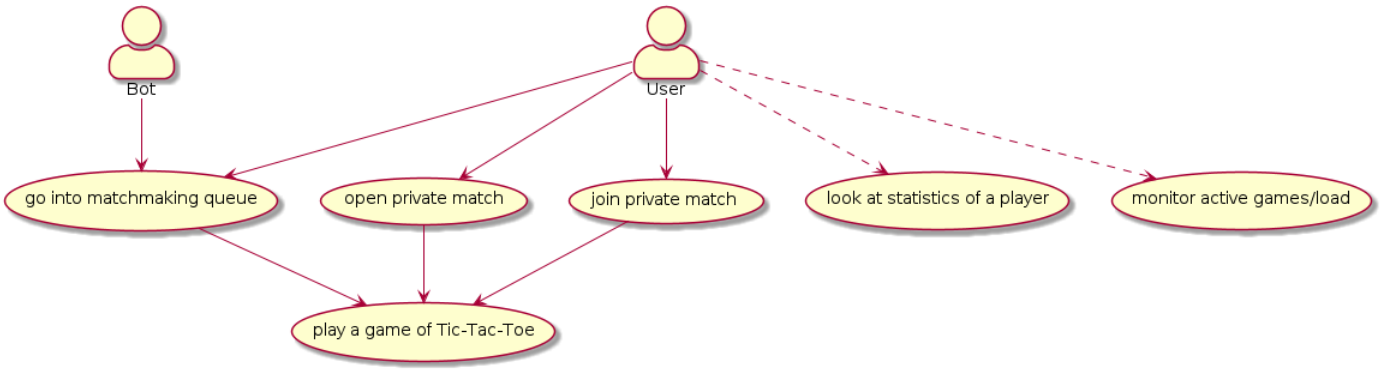
In the spirit of this lecture, the match instances will be separated into different parallel containers, depending on the number of matches or the CPU load of a single container. A matchmaker should be responsible for routing the players to the specific instances.

As an addition to this load-splitting, there should be an overview page in the frontend where the different containers, current games and current load are displayed.

To artificially increase the load on our system we will implement some sort of bots that can be spawned into the matchmaking system, get put into a match against each other by the matchmaker and then play Tic-Tac-Toe.

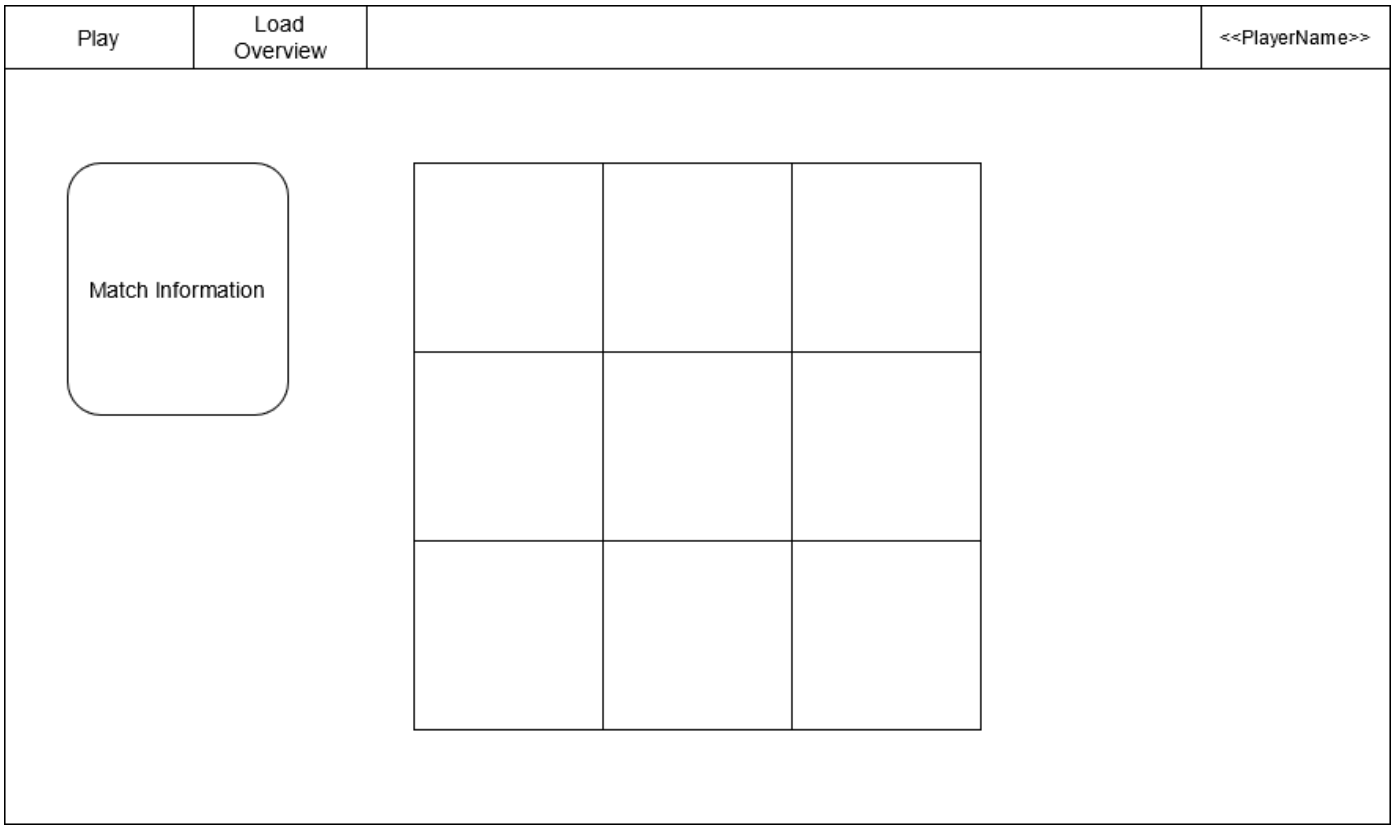
1.2. Use Cases

Below is a use case diagram outlining the different use cases described above.

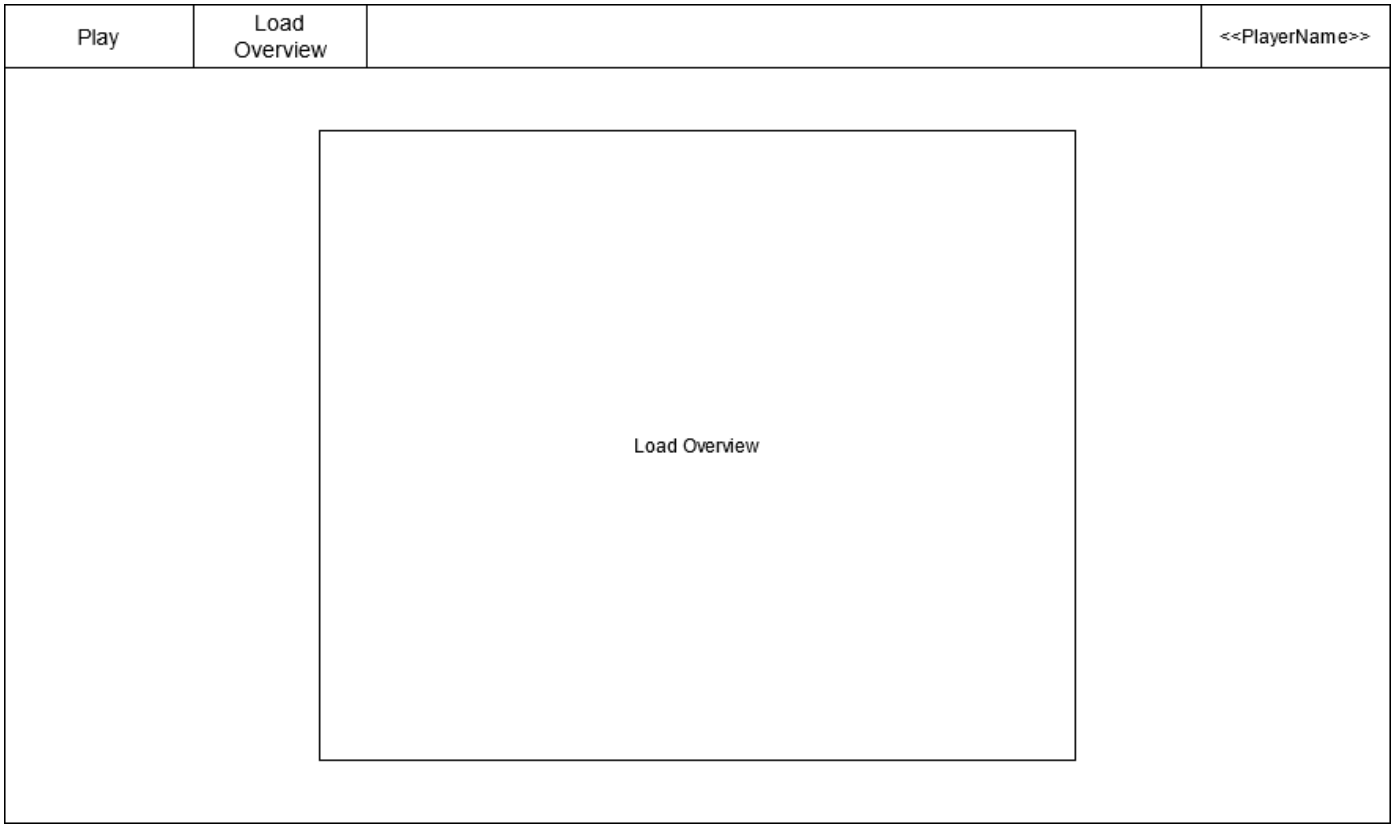


1.3. Interfaces

This is a mock-up of the index page with a welcome message. On opening the site, the user will see this view.



This is a mock-up of the playing field, reachable by the navigation bar. On opening this view, a dialog will pop up. The user will be able to queue themselves for match in the dialog.



This is a mock-up of the system monitor. This view will show the user the current load on the system. The exact data that will be shown is still to be determined.

1.4. Technology

1.4.1. Frontend

TypeScript	Angular	WebSocket
TypeScript is a programming language that build upon the programming language JavaScript. In contrast to JavaScript, TypeScript has a type-safety feature.	Angular is a web-application framework that builds on the singlepage concept. Angular is based on TypeScript. Instead of directly manipulating the DOM, Angular uses a template engine.	To implement a game of Tic-Tac-Toe, a bi-directional, full-duplex connection between client and server is required, so that the server can send data to the client without the client sending a prior request.

1.4.2. Backend

GCP	Docker	Kubernetes	Java	MySQL
Google Cloud Platform will be used as our cloud service provider.	The logic for a game of Tic-Tac-Toe should be run within a Docker container. If a new container should be spawned for each new game or if multiple games can run in one container is to be determined.	For load balancing reasons Kubernetes will be used to create and manage Docker containers depending on current demand.	Java is an object oriented programming language and will be used for our backend development. A WebSocket framework, a REST framework and a database framework may be needed.	MySQL will act as the database management system to store the results of previous games and the associated players.

1.5. Focus Area and Research Question

1.5.1. Description

Our focus is on enabling the system to handle large concurrent loads. The user experience should not deteriorate when many other users are playing at the same time. For this the load will be distributed on many different nodes. Guided by two research questions we want to find the best way to do this.

1.5.2. Research Question(s)

1. Is there a measurable difference in performance and resource usage if multiple matches are played within one container instead of starting a new container for every match? If so, is there an optimal number of matches that can be served from one container?
2. Is one centralised matchmaker enough to handle the load or can this be a bottle neck? If it can, what would be a possible solution?

1.5.3. Approach

We want to challenge our system by developing a method to deploy many bots as players at the same time. Our matchmaking system will then have to handle all new requests, match two opponents and create a new game instance for them.

---

2. Architecture

The goal of this chapter is to describe your system’s architecture as a "grey box". For example, it contains details about the system’s internals down to the component level but not farther.

The structure of this document is inspired by and partially copied from the courses "Software-Engineering" and "Software-Praktikum", which are taught in "Technische Informatik (Bachelor)" as well as by the arc42 template (https://arc42.org/) authored by Dr. Gernot Starke and Dr. Peter Hruschka.

2.1. Architectural Goals

Table 1. Quality attributes

Goal	Rating	Rationale
speed of development	4	the product needs to be in a workable state within a few months
inexpensiveness	1	money is not an issue because we only use free GCP options or coupons
size of scope	2	as development time is restricted the scope of our application must be rather small
response time	2	Tic Tac Toe is turn based so the quickest response time is not essential
throughput	3	our focus lies on being able to manage a large number of active games, but the games itself do not require a lot of bandwidth
capacity	5	our focus lies on being able to manage a large number of active games
scalability	5	our focus lies on being able to manage a large number of active games
availability	2	it's only a game (during peak hours it needs to be available, but downtime during the night is neglectable)
reliability	3	turns should end up in the actual match they were played in but if an input is lost it would not mean the end of the world
resilience	4	D3T should be able to handle extraordinary situations, meaning for example a burst of new players but also weird inputs like players joining and canceling games very quickly
consistency	5	consistency is key because you always want your state of the game to be exactly the one your opponent sees
operability	4	we want our application to be usable by a large audience
simplicity	5	we want our application to be usable by a large audience
modifiability	3	as we are already a group of multiple developers, modifiability is somewhat important to be able to work with other people's code. But as the project lifetime is limited, in a few months nobody will need to modify our code anymore.
security	1	we don't store any sensitive data and if anyone hacks at Tic Tac Toe he/she is a bad human being
usability	4	a game should be fun. An unusable game is not fun.

2.2. Quality Attribute Scenarios

When a Tic Tac Toe tournament takes place and hundreds of players join the matchmaking queue at the same time the system needs to be able to rapidly scale up the game servers so the waiting time is not longer than with just a few active players.

A minimalistic view of the game and necessary functions should be presented to the users. We will not add any advertisment banners or other distracting things to our frontend view.

For any game, consistency is very important. Every move a player makes should be displayed to the opponent exactly. There should never be a state where the visualizations of the match for each player differ from each other. The decision that one player has won the game should always be correct according to the rules.

We will not give special attention to making our game unhackable, but intend that our backend game logic implementation will be able to differentiate between legal and illegal user inputs and act accordingly. So if a player changes the JavaScript frontend code within his browser, he will not be able to get an advantage in the game.

2.3. Constraints

Table 2. Constraints

Constraint	**Category**	**Explanation and rationale**
Google Cloud Platform	technical	We only have coupons for GCP so we will be using it. GCP's strengths and limits must be considered right from the start.
WebSocket	technical	Communication between clients and server(s) will happen via WebSockets. This has to be taken into consideration when planning the component layout. The number of WebSockets that a server can handle at the same time is limited.
remote work	organizational	Due to a current pandemic all work has to be done remotely.

2.4. Stakeholders

Table 3. Stakeholders

Stakeholder group	**Class**	**Responsibilities, interests and concerns**
developers	promoters	Develop the game and the backend architecture. Quick development with many freedoms. A concern is that none of the developers has a lot of knowledge about distributed cloud application development.
players	defenders	Play games. Want to play games and have fun. If the game does not work out like they imagined they might just switch to another game.
support members	defenders	They offer support for players if any problem occured. Ideally no problem occurs and the support members are happy. Concerns might be that due to them not being able to directly fix problems they might receive hate from players.
CEO	latents	Manage the whole company, Tic Tac Toe is just a small part of that. Maximizing profits.

Stakeholder group	Class	Responsibilities, interests and concerns
		If the game Tic Tac Toe somehow generates bad press due to for example many bugs this might lead to the whole company being looked down upon.

2.5. Scope and Context

Table 4. Use Cases

Actor/system	use case	explanation
Player	join matchmaking queue	join the matchmaking queue to get matched against a random opponent
Player	open private match	to play a game against a friend
Player	join private match	to join a game that a friend hosts
Developers	use our API	our game can not only be played via our GUI but also using an API. Other developers might use this to create for example a Discord bot to play Tic Tac Toe on our system
Game websites	embed our game	websites like spieleaffe.de can embed our game into their site

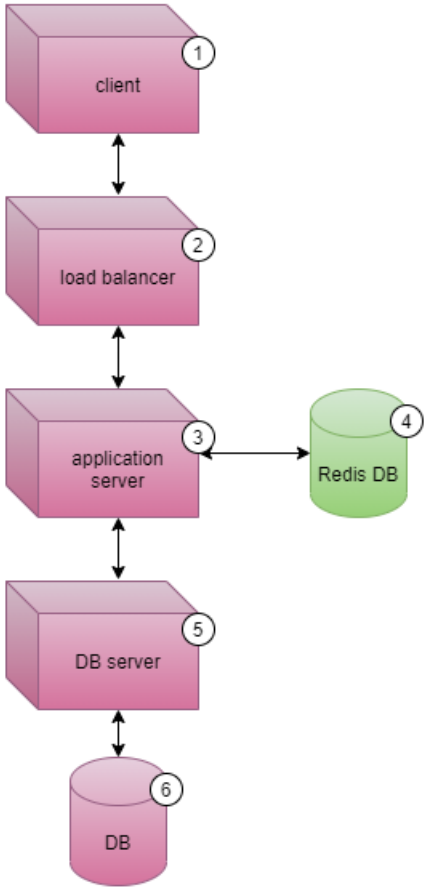
2.6. Key Design Decisions

Table 5. Key Design Decisions

Decision	Category	Explanation and rationale
RESTful web API	technology	A RESTful API will be used for first contact between the matchmaker and any client. Only little information needs to be transfered here and not for a longer period of time, so something like WebSockets was not chosen for this task.
WebSockets	technology	For every game there will be WebSocket connections between the game server and both clients. WebSockets was chosen because we need a continuous, bi-directional flow of data here.
Docker	quality goal	Our goal is to offer a distributed Tic Tac Toe service that can handle high user loads. To fulfill that goal, Docker containers will be used.
Kubernetes	technology	Kubernetes helps us with handling and scaling the Docker containers.
Java	technology	Java is very popular. Implementations of WebSockets are ready to use and usage within Docker containers is fully supported.
No Go	technology	None of us have a lot of experience with the language of Go, so we quickly got rid of the early idea to use Go as our backend programming language.
Split matchmaker from games	architectural	We assume that the main bottleneck will be the actual games. While the matchmaker will only run as one process, the scalability of D3T lies in spawning many containers serving games.
encapsulate turn based logic	organizational	We as a company decided to separate the abstract logic of a turn-based game and the actual implementation of Tic Tac Toe. This is because in the future we might want to

Decision	Category	Explanation and rationale
		develop other turn-based games and can then re-use the existing logic.

2.7. Deployment View



Component(s)	Explanation
client (1)	clients connect to our service via their browser
load balancer (2)	to balance the load if many players want to play at the same time, we use Kubernetes to automatically scale up our game services which run in containers
application server (3)	the logic of both the matchmaker and the actual games runs in Docker containers
Redis database (4)	to ensure consistency of data during a match, each turn is stored in a Redis database
DB server (5)	results of each match get sent to a server to store it in a database
database (6)	the match results get stored in an instance of GCP's "Cloud SQL" using the MySQL-option

2.8. Software Component View

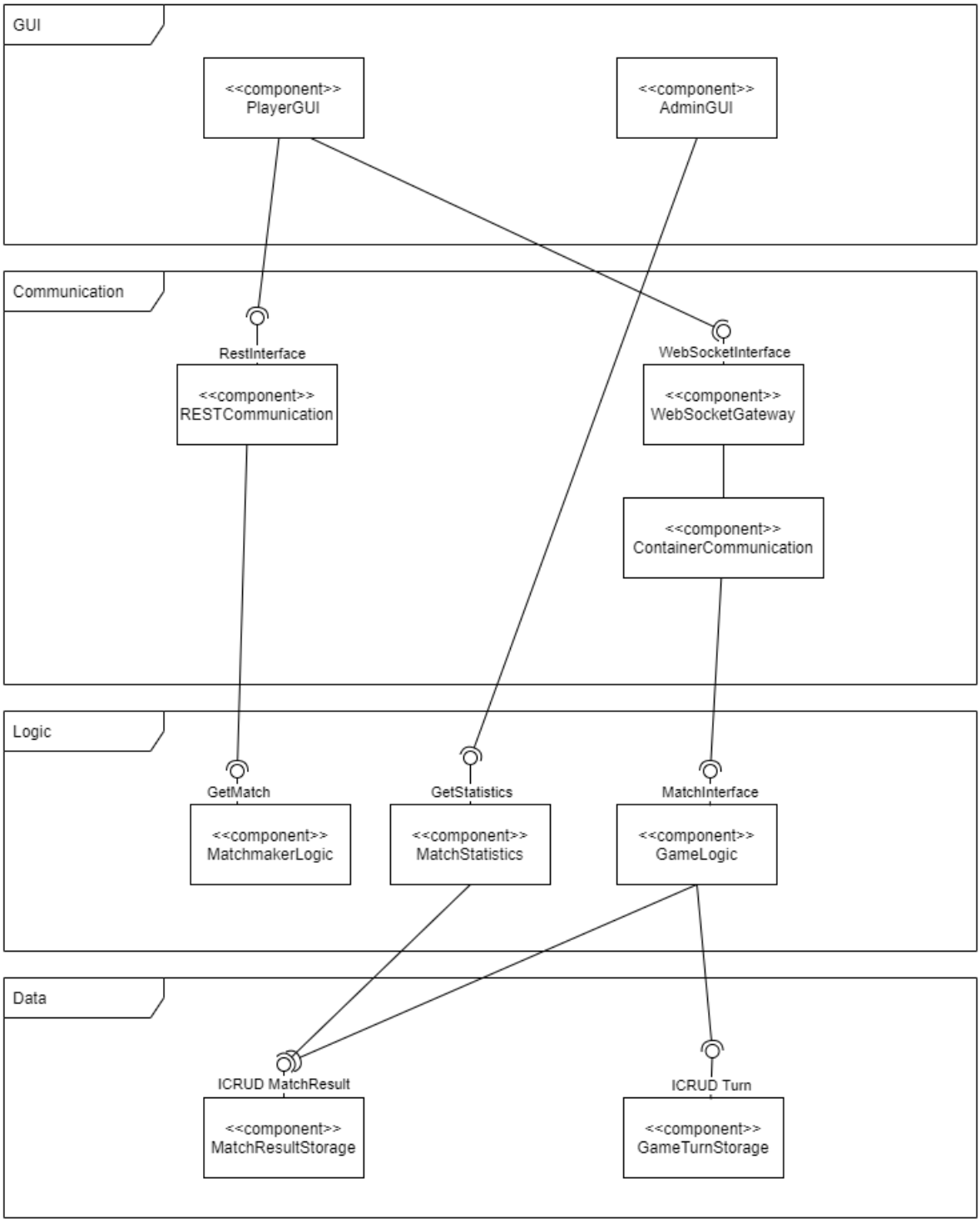
As seen in the following figure, the components of D3T can be split into four layers.

The frontend differentiates between player views and views for an administrator.

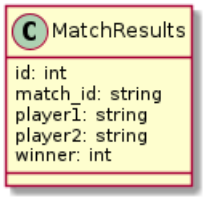
For a player to join and play a match a communication layer is needed, because the actual game gets played in one of many containers. For receiving a match ID the player sends a request to a RESTful API. Once received, a WebSocket connection is put in place to have a bi-directional way of communication to the game logic.

As mentioned in the previous paragraph, the logic is split into a matchmaker and the game logic. Additionally the component MatchStatistics is responsible for providing statistics about match results.

The match results are stored in a database from the game logic after a match has finished. The match statistics logic can then read the results from the same database. During a game, each turn is also stored in a Redis database to ensure consistency.



2.9. Data Schema View



The figure above depicts the data schema of D3T. After each match the match ID, the player names and the winner get stored. This can then be used to create match statistics.

2.10. Runtime View

Three use cases:

1. Find a match against a random opponent
   - player1 sends request to RESTful API
   - request gets sent to matchmaker
   - matchmaker adds player1 to matchmaking queue
   - matchmaker looks if any other player is currently in the matchmaking queue
   - if not, repeat until there is another player in the matchmaking queue
   - if yes, send both players the key to their match
   - player1 receives match key as response view RESTful API
2. Create a private match to play against a friend
   - player2 sends request to RESTful API
   - request gets sent to matchmaker
   - matchmaker creates a new match and sends back match key
   - player2 receives match key as response view RESTful API
   - player2's friend must now enter the match key to join the match
3. Look at statistics about all matches
   - admin1 sends request to view statistics about all matches
   - the match statistics logic selects all matches and calculates for example, how much the win percentages varies depending on if a player has the first or the second move
   - statistics get sent to admin1

2.11. Crosscutting Concepts

This chapter describes important aspect of the architecture which cannot be accommodated in the preceding chapters.

Crosscutting concept	**Category**	**Explanation and rationale**
scalability	operation concepts	The current load on D3T will be measured by CPU and RAM usage. Once one of them surpasses 80%, new containers must be spawned.
source code repository	development	D3T's source code is versioned in a single Git repository hosted on the Gitlab instance of Prof. Woerzberger.
bot implementation	operation concepts	D3T will be playable by bots via an API, not only with the GUI. This is needed to create massive load on the system and answer our Forschungsfrage.