FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Statistics of competitions organised by Czech tennis association
Student:	Vojtěch Drška
Supervisor:	Ing. Ondřej Guth, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Design and implement software for retrieving data and statistics on tournaments, players and other entities under Czech Tennis Association. Create a client–server application. Write the server side using Spring Framework, cover the code with unit and integration tests, and deploy it on a cloud platform in a scalable way. In addition, write scrappers which keep the data up to date. Create a mobile client for viewing the data and statistics gathered by the server.

Electronically approved by Ing. Michal Valenta, Ph.D. on 29 January 2022 in Prague.

Bachelor's thesis

STATISTICS OF COMPETITIONS ORGANISED BY CZECH TENNIS ASSOCIATION

Vojtěch Drška

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Ondřej Guth, Ph.D.
May 10, 2022

Citation of this thesis: Drška Vojtěch. *Statistics of competitions organised by Czech tennis association.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank Ing. Ondřej Guth, Ph.D for supervising this thesis, his time, his suggestions and his personal approach which made the cooperation a pleasant experience. I would also like to thank my parents for supporting me throughout my studies.

Declaration

In Prague on May 10, 2022 ....

Abstrakt

Bakalářská práce se věnuje návrhu a implementaci mobilní aplikace, která slouží k zobrazování statistik soutěží pořádaných Českým Tenisovým Svazem. Dále se tato práce zabývá návrhem a implementací softwaru sloužícího k získávání a zpracovávání dat z webových stránek českého tenisového svazu

Klíčová slova tenis, mobilní aplikace, statistiky, soutěže, sport, REST API, Java, iOS, Python, Swift

Abstract

The subject of this bachelor thesis is design and implementation of a mobile application, used to display the statistics of tennis competitions organised by Czech Tennis Association. Another objective of this thesis is design and development of software used for extracting and parsing data from the website of Czech Tennis Association

Keywords tennis, mobile application, statistics, competition, sport, REST API, Java, iOS, Python, Swift

List of abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AWS	Amazon Web Services
CSS	Cascading Style Sheets
DTO	Data Transfer Object
HATEOAS	Hypermedia As The Engine Of Application State
HTML	Hyper Text Markup Language
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
MVVM	Model-View-ViewModel
ORM	Object Relational Mapping
regex	Regular Expression
RDBMS	Relational Database Management System
REST	Representational State Transfer
UI	User Interface
URI	Uniform Resource Identifier
VPC	Virtual Private Cloud
XPath	XML Path Language

Introduction

In my eyes, tennis is wonderful sport which is why I have been playing it for more than 15 years now. One of the main reasons that makes me love it to the extent I do are the countless competitions organized by Czech Tennis Association.

Given the fact that tennis is a quite widely played sport in Czech Republic with more than 90 000 registered players and roughly 1 500 clubs it comes as no surprise that the amount of matches played each year is quite large. Even though the results of these matches are publicly accessible through the website of Czech Tennis Association[1], there are close to no statistics regarding the individual players except for the rankings which are sadly created only twice a year. Not having any statistics or current rankings makes it very hard for the players to evaluate their performance and it also makes it quite cumbersome to calculate how many points the player has already gained in current season. As there are not only tournaments but also team matches in which teams of players owned by clubs compete against each other, the absence of statistics and up to date rankings makes it unnecessarily complicated for the clubs to pick the players that would best suit their teams.

The objective of this thesis is to analyze existing solutions, mainly in the form of website of Czech Tennis Association[1], define the functional and non-functional requirements, develop software used for obtaining the competition related data from the website of Czech Tennis Association, design and implement the REST API providing the statistics created from said data and create a mobile application used to display statistics queried from the REST API. In addition to the statistics, the application will also provide its users with weekly rankings of players based on the latest available data. In order to achieve the best possible availability of the REST API it will be deployed and run in cloud.

The thesis is split in to five chapters. The chapter number 1 is focused on the analysis of existing solutions, providing some background to how the tennis competitions organised by Czech Tennis Association work and mainly on the requirements analysis of our application. In the chapter number 2 the architecture of the application is designed alongside with the database schema. Also, the choices of technologies to be used are discussed in this chapter. The chapter no.3 is devoted to the implementation of all parts of our application and the problems that were solved during the implementation are discussed there. Chapter number 4 focuses on how different parts of the application were tested and finally the chapter number 5 concludes how we were able to accomplish the requirements specified in chapter number 1 and describes some of the work that is planned to be done on the application in the future.

Chapter 1
Analysis

1.1 Tennis competitions

In order to fully understand the problems which are addressed in this thesis as well as the structure and constraints which will be present in the database, it is necessary to explain how the tennis competitions organised by Czech Tennis Association work as well as some of the regulations that follow them. There are two main types of competitions, tournaments and team matches.

1.1.1 Tournaments

The participants of tournaments are individual players. The season during which tournaments are played takes place from the start of November till the end of September. The number of participants is limited on each tournament and is determined beforehand by the club that is the organizer of the tournament. There are two parts of each tournament, singles and doubles. Participant may choose if they wish to participate in only singles or doubles or if they prefer to participate in both. The tournaments are always played in the form of a knockout draw, which means that once the participant loses they get eliminated from the respective part of tournament. There are also regulations preventing players from participating in multiple tournaments of the same age category at the same time. There is, however, no regulation that limits the players from participating in two tournaments of different age categories at the same time.

1.1.2 Team matches

The participants of team matches are two teams. Vast majority of the team matches takes place from the end of April till the end of July. In the beginning, teams get divided to groups of around eight participants. These groups are determined by the results that the teams achieved during the last year's team match season. Each team must have a club which is its owner. Only players native to the club or players drafted by the club before the start of team match season may be part of the team. Team consists of six players, four males and two females. During one team match, each player plays one singles match and one doubles match which equals to a total of nine matches during one team match. The team whose players won the most matches is declared the winner of the team match.

1.1.3 Points

When a player participates in a tournament, they are awarded points for each match they win. The amount of points is determined by a total of 3 factors the first one being that the closer the player gets to finals, the more points they are awarded.

Second factor is the category of the tournament. Tournaments are divided into multiple categories. There are categories A, B, C, D which, in the given order, are ranked from the highest to the lowest level. Therefore the tournaments of category A are the ones where the player is awarded the most points, but they feature some of the best players in Czech Republic. On the other hand tournaments of category D will award players with very little points, however they are open only to less professional or hobby players and players above certain rank are restricted from those tournaments. There are also tournaments of categories MCR and P. The tournaments of category P are regional tournaments and they take place twice a year. The tournaments of category MCR take place also twice a year and their participants are determined by the results of regional tournaments.

The last factor which determines the amount of points the player is awarded on a tournament is the sum of rankings of the top 8 participants. This sum determines the subcategory of the tournaments as per the rules[2].

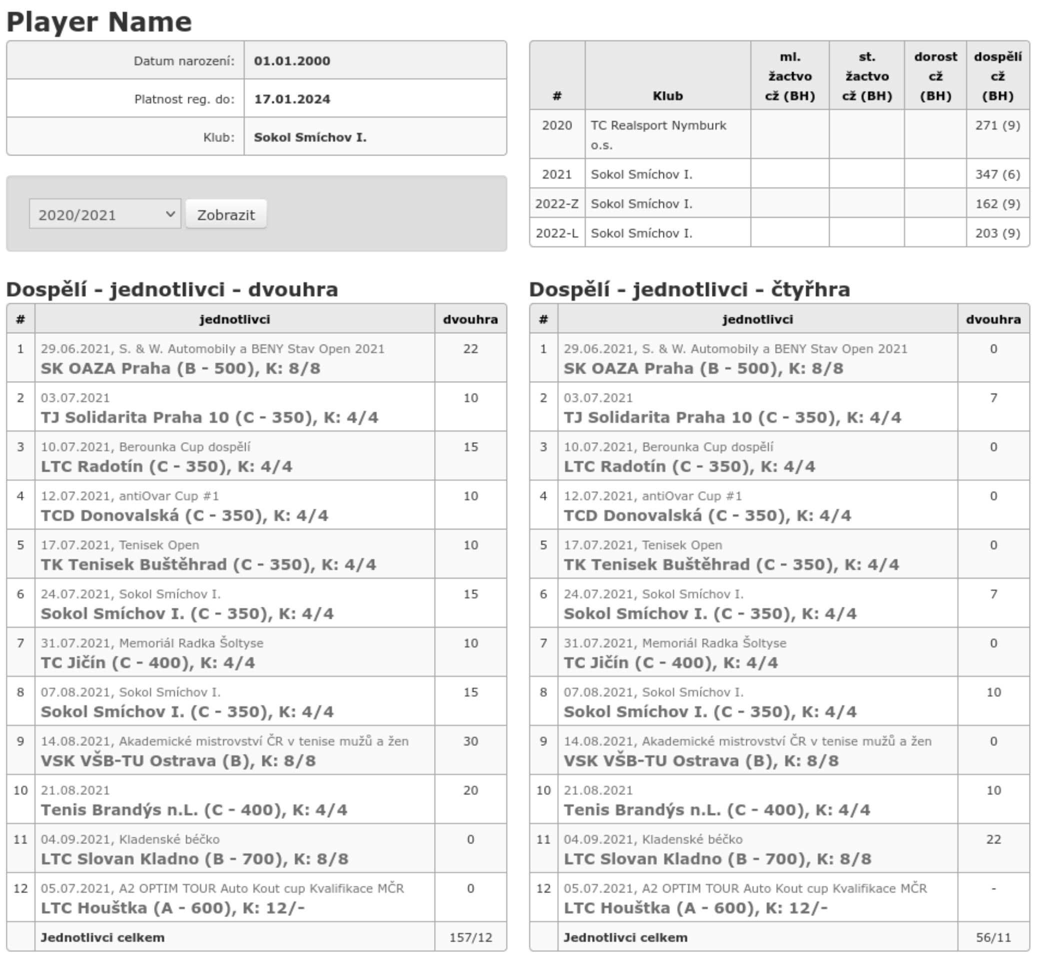
There is also a special situation in which a player may be awarded points twice in a single part of a tournament. This may happen at some of the biggest tournaments as they do not have main draw only, but they also have qualification. Therefore when a player wins the qualification and then wins at least one match in the main draw, they are awarded points once for the qualification and once for the progress they were able to make in the main draw.

Players are also awarded points for the matches that they win during a team match. The number of points awarded is determined simply by the ranking of the opponent alone.

1.1.4 Rankings

Twice a year (only once a year till the 2022) new official rankings are made based on the amount of points the player has earned throughout the season. In order to make the rankings as relevant as possible, there are some rules that determine which points will be counted in the rankings and which not. Only the best eight events in singles and best eight events in doubles are counted into the rankings. By an event we mean a tournament or a sum of four best team matches. This rule is incredibly important as it prevents some players from participating in and immense amount of low category tournaments per year and earning large sums of points from the quantity and not quality of their matches. This very much helps to make the official rankings more relevant. Also, no more than four events from winter or summer season may be counted in the rankings as the competition tends to be weaker during the winter[2].

Currently the official rankings are only made twice a year. As the season progresses the official rankings start to lack relevance as they do not reflect the latest performances of players. A sample situation would be that if a player missed the whole past season he will remain unranked throughout the current season despite their winnings. This will make taking part in competitions of higher category impossible for them even though they may be skilled enough to compete as the players are accepted based on their rank. Unfortunately we cannot change these rules as they are made by Czech Tennis Association but the least that we can do is to create weekly rankings to give the players rough estimate of their rankings throughout the season. The weekly rankings will also help to determine the current skill of players as it will reflect upon their latest performances.



Player Name

Datum narození:	**01.01.2000**
Platnost reg. do:	**17.01.2024**
Klub:	**Sokol Smíchov I.**

#	Klub	ml. žactvo cž (BH)	st. žactvo cž (BH)	dorost cž (BH)	dospělí cž (BH)
2020	TC Realsport Nymburk o.s.				271 (9)
2021	Sokol Smíchov I.				347 (6)
2022-Z	Sokol Smíchov I.				162 (9)
2022-L	Sokol Smíchov I.				203 (9)

Dospělí - jednotlivci - dvouhra

#	jednotlivci	dvouhra
1	29.06.2021, S. & W. Automobily a BENY Stav Open 2021 **SK OAZA Praha (B - 500), K: 8/8**	22
2	03.07.2021 **TJ Solidarita Praha 10 (C - 350), K: 4/4**	10
3	10.07.2021, Berounka Cup dospělí **LTC Radotín (C - 350), K: 4/4**	15
4	12.07.2021, antiOvar Cup #1 **TCD Donovalská (C - 350), K: 4/4**	10
5	17.07.2021, Tenisek Open **TK Tenisek Buštěhrad (C - 350), K: 4/4**	10
6	24.07.2021, Sokol Smíchov I. **Sokol Smíchov I. (C - 350), K: 4/4**	15
7	31.07.2021, Memoriál Radka Šoltyse **TC Jičín (C - 400), K: 4/4**	10
8	07.08.2021, Sokol Smíchov I. **Sokol Smíchov I. (C - 350), K: 4/4**	15
9	14.08.2021, Akademické mistrovství ČR v tenise mužů a žen **VSK VŠB-TU Ostrava (B), K: 8/8**	30
10	21.08.2021 **Tenis Brandýs n.L. (C - 400), K: 4/4**	20
11	04.09.2021, Kladenské béčko **LTC Slovan Kladno (B - 700), K: 8/8**	0
12	05.07.2021, A2 OPTIM TOUR Auto Kout cup Kvalifikace MČR **LTC Houštka (A - 600), K: 12/-**	0
	Jednotlivci celkem	157/12

Dospělí - jednotlivci - čtyřhra

#	jednotlivci	dvouhra
1	29.06.2021, S. & W. Automobily a BENY Stav Open 2021 **SK OAZA Praha (B - 500), K: 8/8**	0
2	03.07.2021 **TJ Solidarita Praha 10 (C - 350), K: 4/4**	7
3	10.07.2021, Berounka Cup dospělí **LTC Radotín (C - 350), K: 4/4**	0
4	12.07.2021, antiOvar Cup #1 **TCD Donovalská (C - 350), K: 4/4**	0
5	17.07.2021, Tenisek Open **TK Tenisek Buštěhrad (C - 350), K: 4/4**	0
6	24.07.2021, Sokol Smíchov I. **Sokol Smíchov I. (C - 350), K: 4/4**	7
7	31.07.2021, Memoriál Radka Šoltyse **TC Jičín (C - 400), K: 4/4**	0
8	07.08.2021, Sokol Smíchov I. **Sokol Smíchov I. (C - 350), K: 4/4**	10
9	14.08.2021, Akademické mistrovství ČR v tenise mužů a žen **VSK VŠB-TU Ostrava (B), K: 8/8**	0
10	21.08.2021 **Tenis Brandýs n.L. (C - 400), K: 4/4**	10
11	04.09.2021, Kladenské béčko **LTC Slovan Kladno (B - 700), K: 8/8**	22
12	05.07.2021, A2 OPTIM TOUR Auto Kout cup Kvalifikace MČR **LTC Houštka (A - 600), K: 12/-**	-
	Jednotlivci celkem	56/11

■ **Figure 1.1** Profile of player

1.2 Analysis of existing solutions

The only website which features the data of Czech tennis competitions is the official website of Czech Tennis Association[1]. Its biggest strength is undoubtedly the fact that it is the official and only place where players can sign up for tournaments organized under the Czech Tennis Association giving them no other choice than to use it. Also, given the previously mentioned reasons, it is certain that the data on this website are complete as all the competitions and their results must be listed on it.

Even though this website is overall great and serves its purpose quite well, it really only displays the data from its database and does not work with them any further in the slightest. This, however, cannot be taken as a negative as the main purpose of the website is for the players to view the calendar of tournaments and team matches and sign up for tournaments. In the figure 1.1 it can be seen that all the tournaments that the player participated in are displayed along with the points that they were awarded. If the player wanted to know how much of the points is actually calculated towards the rankings, that information is nowhere to be found. Another thing that can be noticed is that apart from the yearly rankings there is no such thing as current or weekly rankings. The two previously mentioned issues will definitely be addressed in our application

Apart from the official website of Czech Tennis Association[1] there are no websites or applications of any sort that feature data related to competitions organised by Czech Tennis Association.

1.3 Requirements analysis

This section will describe the requirements our application must conform to. All of these requirements describe functionalities/features that the end users expect from the application. Our application must meet all of the requirements as it would fail to fulfill its purpose otherwise.

1.3.1 Functional requirements

Functional requirements describe what our application must be able to do. These requirements describe the features that will be available to the end user of application. In case the application does not fulfill some of the requirements it fails as it no longer offers the promised features to the end user.[3][4]

1.3.1.1 F1: Weekly rankings

The application must provide its users with weekly rankings for all existing categories. It should also feature multiple types of weekly rankings namely: singles, doubles, combined. As the names suggest, singles rankings will count only the points player gained on singles events, doubles only the points gained on doubles events and combined will sum up the points gained from both singles and doubles events. The details on how the points are summed up can be found in subsection 1.1.3.

1.3.1.2 F2: Detail of player

The application must contain a screen displaying some of the basic statistics of the selected player. This screen should also feature links to other screens with more detailed statistics, for example links to *Match screen* and *Points screen.*

1.3.1.3 F3: Profile of player

As the expected end user is a player, the application must provide an option of selecting himself/herself in the list of players. This information will than be used to display their basic statistics and a quick link to player's details on the main screen.

1.3.1.4 F4: Player's counted points

Each player must be able to display the amount of points that is counted towards both the official and weekly rankings. As the player may be participating in multiple categories at same time they must be given the option of choosing the category in which they are interested in. The counted points must also be available for all the previous seasons. The details on how the points are summed up can be found in subsection 1.1.3.

1.3.1.5 F5: Achievements of a player

The application must be able to provide the players with number of times they were able to place on each of the top three places (winner, finalist, semifinalist) in the past events.

1.3.1.6 F6: Official rankings

The application must contain not only the weekly rankings but also the official rankings. The reasoning behind this requirement is that even though the official rankings are available on the official website of Czech Tennis Association[1] they still are a vital part of player's statistics and therefore should be present in the app.

1.3.1.7 F7: Match statistics

The application must provide the players with the information of how many singles and doubles matches they have won and lost throughout their career. The time frame of whole career is inspired by the player's statistics on the page of ATP association[5] as well as it has been requested by multiple players. The application also must contain details of all these matches such as dates of the matches, scores and the events at which the matches were played. Even though this data is available on the official website of Czech Tennis Association[1] it plays a very important part in the player's statistics and therefore should displayed in the application.

1.3.1.8 F8: Head to head comparison of players

The application must provide the option of comparing players head to head. This comparison should show the number of singles/doubles matches that the players played against each other. It also should display the dates of the matches, scores and the events at which the matches were played.

1.3.1.9 F9: Player search

The application must have the option of searching through the players by their name.

1.3.1.10 F10: Counted events

Each player must be able to view the events from which their points counted towards the rankings originated. This feature should not only be able to display the events for the current season, but also for all the previous seasons. As player can compete in multiple categories at the same time, they must be given the option of choosing the category they are interested in.

1.3.1.11 F11: Tiebreaks

The application must be able to display the amount of tiebreaks won and lost for each player.

1.3.1.12 F12: Super-tiebreaks

The application must be able to display the amount of super-tiebreaks won and lost for each player.

1.3.1.13 F13: Data scrapping

The application should update its data at least on a weekly basis. This process should be fully automated and not require any manual intervention.

1.3.2 Non-functional requirements

Non-functional requirements define the way application must be doing things. They can be also though of as quality constraints that the application must satisfy.

1.3.2.1 N1: Device support

The application must support a significant part of mobile devices used in the current market.

1.3.2.2 N2: Availability

The application must be able to display data at all times.

1.3.2.3 N3: User experience

The application must be user friendly. The user interface should be designed in a way that makes it intuitive and easy to use.

1.3.2.4 N4: Responsiveness

The application should remain responsive at all times.

1.3.2.5 N5: Security

The application must be designed and implemented with security measures in mind. As the data used by the application may contain personal information the application must not be vulnerable to any kind of attacks which could cause the data to be stolen or leaked.

1.3.2.6 N6: Cost efficiency

The application, mainly the cloud related part, should be implemented as cost effectively as possible. With that said, the application still and most importantly should be implemented in as safe as possible manner and be fully functional, however, it should also not use more resources than necessary.

1.3.3 Use cases

Use cases describe different scenarios in which actors interact with application. Each of these scenarios must consist of the following parts:

- Actor, the entity which will be interacting with the application.
- Goal, the successful outcome of the use case.
- System, description of the steps needed to obtain the given goal.

[6] The relations between individual use cases and functional requirements can be seen in the table 1.1

1.3.3.1 UC1: Player search

Actor: Any user
Goal: Search player by name.
System: In this use case the user is at the *Search player* screen. The presumption is that user arrived at this screen through a link from a different screen which will make use of the player selected here. The search bar will be present on the screen into which the user will enter the desired name of player. After that the search results will appear in the form of a list. The user will then select the player of their choice from the list of search results. After that the user will be taken back to the screen from which the *Search* screen was acessed.

1.3.3.2 UC2: Profile selection

Actor: Any user
Goal: Set the default player profile on the main page.
System: After opening the application there will be a tab present on the main screen prompting the user to select their default player profile if they have not done so already. After taping the tab the user will be taken to a *Search player* screen where they will perform actions according to the UC1 1.3.3.1. After returning to the main screen the tab which previously prompted the player to select the default profile will now contain the basic statistics of the selected player. Its behaviour will also change as it will take the user to the default player's details page on tap and it will let the user change the default player on long press.

1.3.3.3 UC3: Head to head

Actor: Any user
Goal: Show the head to head comparison between two players.
System: After opening the application the user will be greeted with the main screen. On the main screen, there will be a tab with title *Head2Head* present. After taping the tab user will be taken to a *Head2Head* screen. This screen will contain two fields prompting the user to select players. After taping one of the fields the user will be taken to a *Search player* screen where they will perform actions according to the UC1 1.3.3.1. Then they will repeat the whole process of searching and selecting player for the second player. After selecting the second player, the statistics of the matches and the matches played by the selected players will appear.

1.3.3.4 UC4: Displaying details of a player

Actor: Any user
Goal: Show the details of a player
System: After opening the application the user will be greeted with the main screen. On the main screen, there will be a search bar present. After taping the search bar the user will be taken to a *Search player* screen where they will perform actions according to the UC1 1.3.3.1. Then the user will be taken to the Player details screen of the player.

1.3.3.5 UC5: Displaying counted points of a player

Actor: Any user
Goal: Show the details of desired player
System: In this scenario we expect the user to already be at the *Player details* screen. On this screen the user will select the tab with title *Points* after which the user will be taken to Points screen. Here the user can select the season and category in which they are interested the most after which the application will show the counted points and associated events in the given season and category.

1.3.3.6 UC6: Official rankings

Actor: Any user
Goal: Show the official rankings
System: After opening the application the user will be greeted with the main screen. On the main screen there will be a tab with title *Official rankings* present. After tapping the tab the user will be taken to the Official rankings screen. There the user will be able to pick the category, season and gender of the rankings which will then be displayed by the application.

1.3.3.7 UC7: Weekly rankings

Actor: Any user
Goal: Show the official rankings
System: After opening the application the user will be greeted with the main screen. On the main screen there will be a tab with title *Weekly rankings* present. After tapping the tab the user will be taken to the Weekly rankings screen. There the user will be able to pick the category, gender and type of the rankings which will then be displayed by the application.

1.3.3.8 UC8: Matches

Actor: Any user
Goal: Show the match related data
System: In this scenario we expect the user to already be at the *Player details* screen. On this screen the total number of singles/doubles matches that the player won/lost will be displayed in one of the tabs. After selecting the respective tab, the user will be taken to *Match* screen which will display all the events and matches played in the selected season and category.

1.3.3.9 UC9: Scrapping tournament related data

Actor: Time
Goal: Scrap new tournament related data
System: The cloud platform will contain multiple timed events which will trigger scrapping functions in given times. The functions scrapping the tournament related data will be ran on a weekly basis throughout the whole year except for October as this month is free of all competitions. These functions should operate autonomously and without any manual intervention thus the time is the only actor.

1.3.3.10 UC10: Scrapping team match related data

Actor: Time
Goal: Scrap new team match related data
System: The cloud platform will contain multiple timed events which will trigger scrapping functions in given times. The functions scrapping the team match related data will be ran on a weekly basis from April to June inclusive. These functions should operate autonomously and without any manual intervention thus the time is the only actor.

■ **Table 1.1** Relations between use cases and functional requirements

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13
UC1									X				
UC2			X						X				
UC3							X	X	X				
UC4	X	X		X	X	X	X		X		X	X	
UC5		X		X			X		X	X			
UC6						X							
UC7	X												
UC8		X					X	X					
UC9													X
UC10													X

1.3.4 Domain model

Domain model displays the entities and associations between them that are important for the application. The domain model of our application can be seen in the figure 1.2. Currently we are aware of the entities that exist in our domain and we are also aware of the associations that they have to each other. The model, however, does not display attributes of the entities as we at this point in time do not know what the exact attributes are. This due to the fact that majority of the data is scrapped from the official website of Czech Tennis Association[1] and we do not know yet which information about the entities we will be able to scrap.

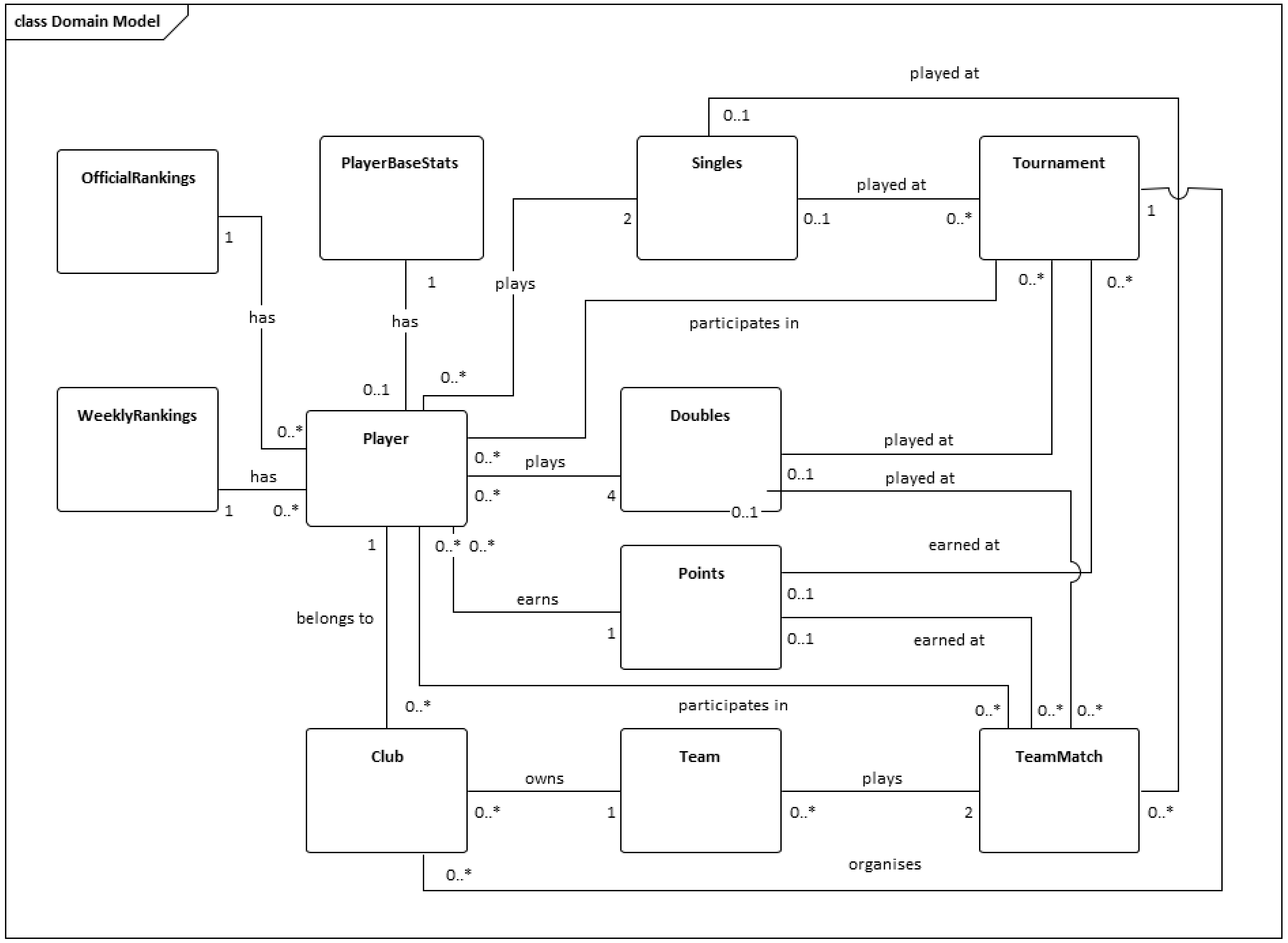


Figure 1.2 Domain model

Chapter 2
Design

2.1 High-level Architecture

The three-tier architecture will be used to design the application. This architecture divides the application into three tiers:

- Presentation
- Application
- Data

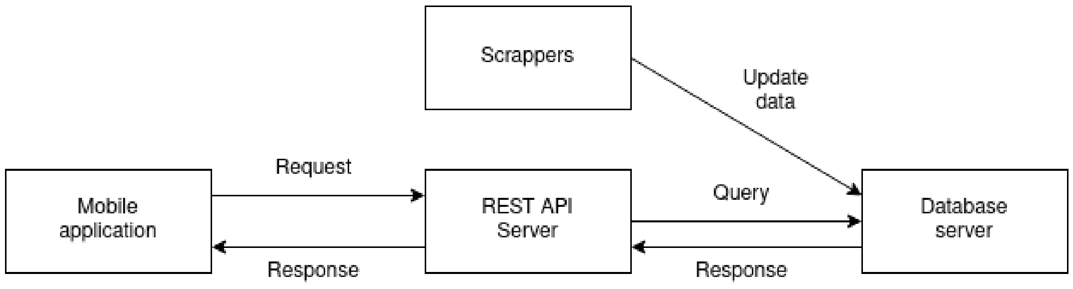
The presentation tier contains the interfaces with which the end user interacts. Its main role is to request data from the application tier based on the given inputs and display them in an easy to use/read form. The application tier contains all of the business logic. Its purpose is to both handle the requests from the presentation tier and manipulate the data in the data tier. The last tier is the data tier whose main purpose is to store and manage the data processed by the application tier. In the three-tier architecture the presentation and data tiers should never communicate with each other directly but rather use the application tier as a middle man. One of the main advantages of this architecture is the fact that resources such as database can be shared amongst many users. Another advantage that should be mentioned is that once a trend in requested data emerges, we can employ solutions such as caching to make the application faster and more efficient.[7][8]

The way three-tier architecture is going to be applied on our application can be seen in the figure 2.1. The mobile application will act as the presentation layer. The application layer will be made up of two parts, REST API and scrappers. It also can be seen that the REST API server will only consume data from the database as all the data will origin from the official website of Czech Tennis Association[1] and be inserted by the scrappers. The idea behind this decision is that the scrappers will run only couple times a week for brief periods of time and thus the server would be idling for the majority of the time which would violate the non-functional requirement 1.3.2.6.

2.2 Database

2.2.1 Architecture

When choosing the type of database that we want to use we have the options of using either SQL or NoSQL database. We decided to use SQL database due to the following reasons, the first



■ **Figure 2.1** High level architecture

one being the cost. Whilst running and SQL database in AWS can cost as low as 140 dollars per year, running a similar NoSQL instance would cost roughly 300 dollars per year[9]. Next reason for using SQL database is that we want to have our database ACID compliant which can be done by simply choosing the right RDBMS. The last reason as to why we want to use SQL database is that our data is very structured and we can easily obtain the information we need from it using SQL queries. Also, we plan to use stored procedures2.2.3 as part of our application logic. Admittedly this would be also realizable when using NoSQL database such as MongoDB, however, this would be quite expensive time-wise as we have no previous experience doing so.

2.2.1.1 Relational database management system

It was decided to use PotsgreSQL as the RDBMS due to multiple reasons. The first reason is that PostgreSQL is ACID compliant by default[10]. This is important to us because the consistency of data is crucial in our case as the statistics and overall the data displayed by our application would be of no value if the data were not consistent. Another reason for using PostgreSQL is that it is one of the most popular RDBMSs currently[11] which ensures great support from cloud providers as well as many resources when running into issues. The last reason is the cost of running the database in AWS as the type of RDBMS can change the price quite drastically. When selecting identical instances, availability zones and reserving the instance for one year, PostgreSQL came out at around 140 dollars per year which was significantly cheaper than other options such as MySQL at 165 dollars or Oracle at 230 dollars[9]. The only database was cost-wise on par with PostgreSQL was MariaDB, however we still decided to use PostgreSQL mainly due to the fact that it is much more popular[11].

2.2.2 Database model

The model of the database is partially given by the structure of data that we will be scrapping from the official website of Czech Tennis Association[1]. Nevertheless, the model will also contain many entities and relations of our own as we will be trying to derive information from the data in order to create interesting statistics. The structure of the database can be found in the figure 2.2. Even though most of the tables are quite self-explanatory some of them could use a few words.

Age_category_enum table will contain the different age categories in which the players can compete. There are multiple entries in this table which are never going to be changed.

Gender_enum table will contain the gender categories under which the tournaments and rankings are divided. The entries in this table are never going to be changed.

Match_type_enum table will contain the match types, namely singles and doubles. The entries in this table are never going to be changed.

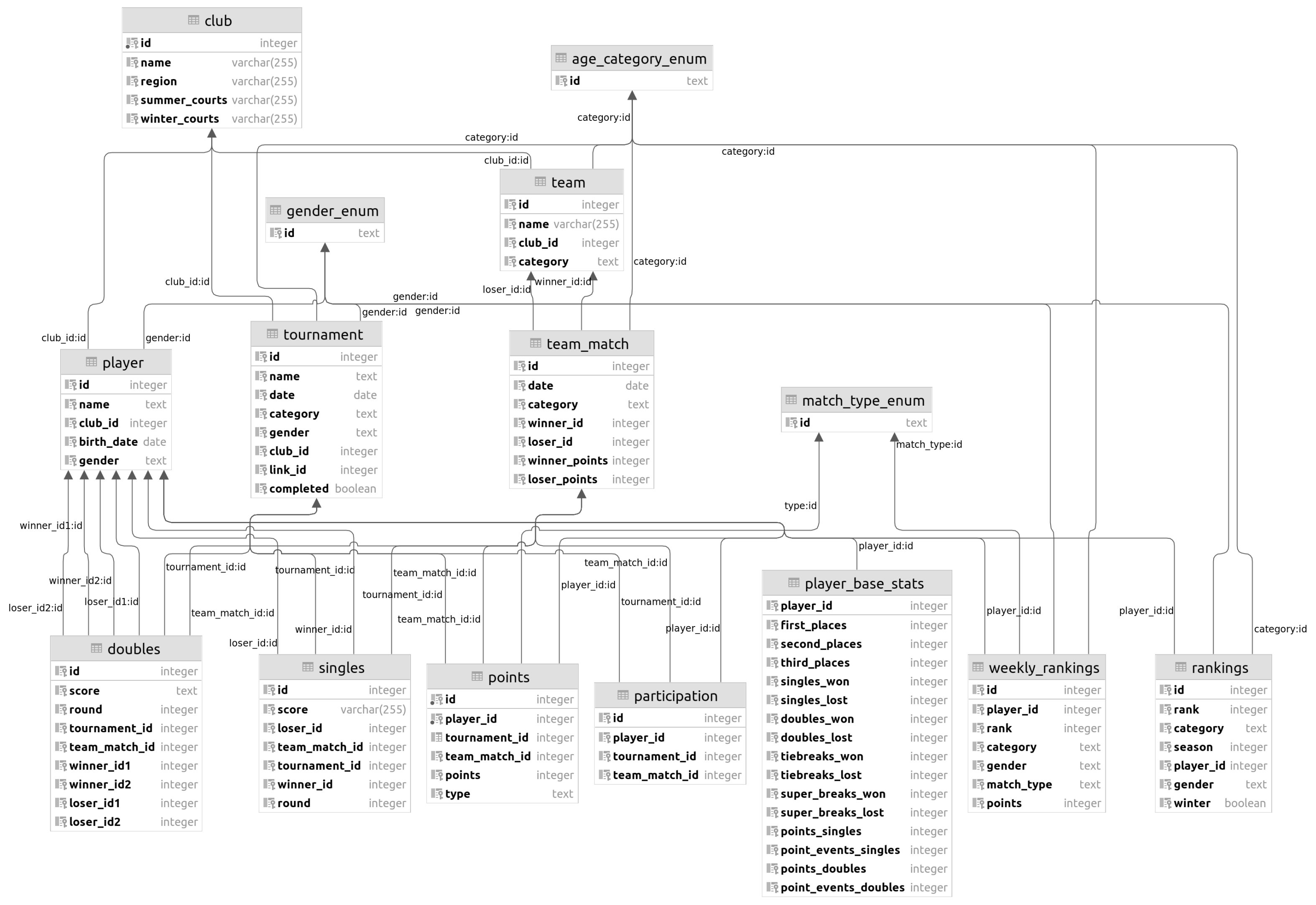
Participation table is the decomposition of many-to-many relationship between the player and tournament/team match tables. It will hold the information of all player's participation in tournaments and team matches.

Player_base_stats table represents the basic statistics of a player. These statistics will be calculated on demand by a stored procedure in the database and inserted into this table.

Points table contains the amount of points gained by a player on an event. It should be noted that if player has not gained any points on an event, then there will be no record in the table. This decision was made in order to improve the performance of the database as the amount of entries in the table would be doubled without it and it would not bring in any new information or benefit.

It should be noted that *age_category_enum*, *gender_enum* and *match_type_enum* tables could have been created as enumerated type as PostgreSQL enables us to do so. The reason for creating these enumerables as tables is that JPA, which we will be using as the API will be implemented using Spring Boot 2.5, does not allow us to bind entities to PostgreSQL enumerated types.

Another thing that should be mentioned is that the database model was created while implementing the scrappers 3.1 as we would otherwise not know the attributes of the entities as mentioned in 1.3.4.



■ **Figure 2.2** Database model

2.2.3 Stored procedures

It was decided to implement parts of the application logic as stored procedures as we expect it to be much simpler and save a lot of bandwidth in some cases. Another reason as to why we want to use procedures is that some actions, such as creation of weekly rankings or updating the participation, will most likely have to be performed from both the API and scrappers. If we were to not use the stored procedures we would have to implement the process of creating the rankings and participation in both the API and scrappers which would certainly be a less elegant solution than using stored procedures.

2.3 Mobile application

2.3.1 Architecture

For the development of mobile application the MVVM architectural pattern will be used. The components which makeup the MVVM architecture and their relations can be seen in the figure 2.3.

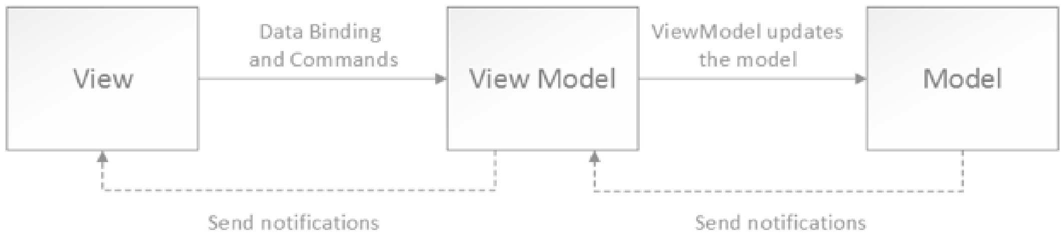


Figure 2.3 MVVM Architecture

2.3.1.1 View

The View layer is used to define UI of an application. All the information regarding the design and appearance of screens and other UI related elements are contained in this layer. Apart from displaying the UI it also handles all the user interaction.

2.3.1.2 Model

This layer describes the structure of data that are being used in the application. These structures are usually in the form of structs or classes. It also tends to be the layer in which business logic is handled.

2.3.1.3 ViewModel

The ViewModel layer is used to provide the data from Model layer in an easily displayable form to the View layer, sometimes alongside with new functionalities. It could be said that the ViewModel layer is a middle man between the Model and View layers. This layer is also responsible for notifying the View layer when any change to the displayed data happens so the respective view gets refreshed.

2.3.2 Platform

Firstly, it is important to say that the current plan is to get the application as a whole into production. This means that two applications will have to be developed, one for iOS and one for Android. Unfortunately, creating two mobile applications for two different platforms would make the topic of this thesis too broad to be completed within a reasonable time-frame which is why we decided to include creation of only one of the applications in this thesis. Therefore, the first decision that has to be made is whether the application should be running on Android or iOS devices. One thing that should also be noted is that there is an option of using Flutter that would enable us to make just one application for both platforms which is undoubtedly its main advantage. On the other hand, creating two mobile applications means that they could be developed in their native programming languages which would enhance both the user experience and robustness of the applications. As both options seem to be equally great, the decision came down to our current possibilities and personal preference. The Faculty Of Information Technology at CTU currently offers the students a course on iOS development in Swift which has quite impeccable reputation. In addition to that, Swift has always been a language which we wanted to learn and with that it was decided that two standalone applications are going to be developed and the one which will be part of this thesis will be running on iOS platform[12].

2.3.3 Programming language

As mentioned in 2.3.2 the language that is going to be used for the development is Swift. There is also an option of using Objective-C however after a short research it became clear that Swift is the language to use as it is faster, safer and much easier to learn and use[13].

2.3.4 UI Framework

When choosing the UI framework we are presented with two options, UIKit and SwiftUI. UIKit is the older one of the frameworks and therefore possesses all the functionalities and is very well documented and time proven. On the other hand, SwiftUI is a rather new framework as it was released in 2019. The main advantage that SwiftUI has over UIKit is that it is much easier to learn and use. As SwiftUI is considered the successor of UIKit it also is just a matter of time till UIKit stops to receive support and will become deprecated. One of the drawbacks of SwiftUI is that some features that UIKit has are not yet supported in it. This, however, is not much of a problem as our application will most probably not require many advanced functionalities and even if it would, there is an option of developing part of code in UIKit as the frameworks are inter-operable. Considering the information above it was decided to use SwiftUI in our application as the main UI framework[14].

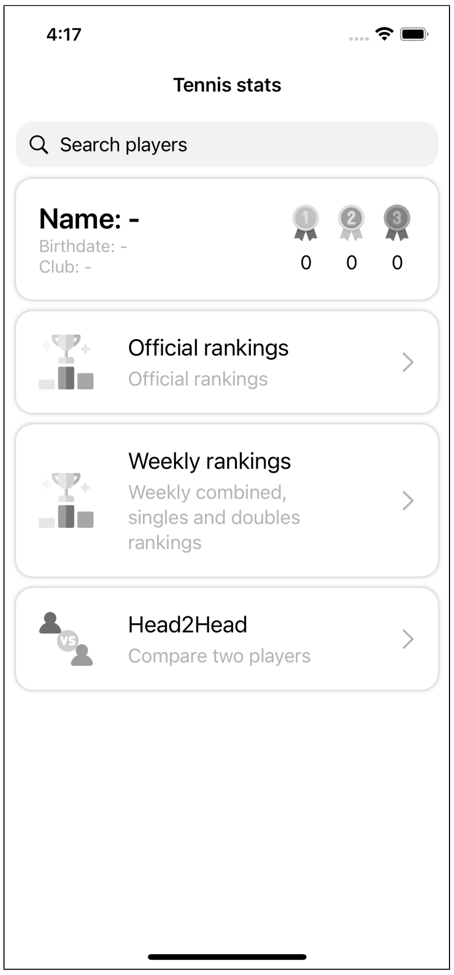
2.3.5 UI Design

2.3.5.1 Main screen

In the figure 2.4 we can see the *Main Screen* with no default user selected. In the figure 2.5 the *Main Screen* with selected default player can be seen. This screen is used to set and alter the default user and show some of their statistics along with a link to their *Player details* screen.

2.3.5.2 Search screen

In the figures 2.6 and 2.7 we can see the *Search Screen* without and with search results respectively. This screen is used in order to search for players by name and is needed for 1.3.3.1 use case and all use cases which require it.


4:17

Tennis stats

Search players

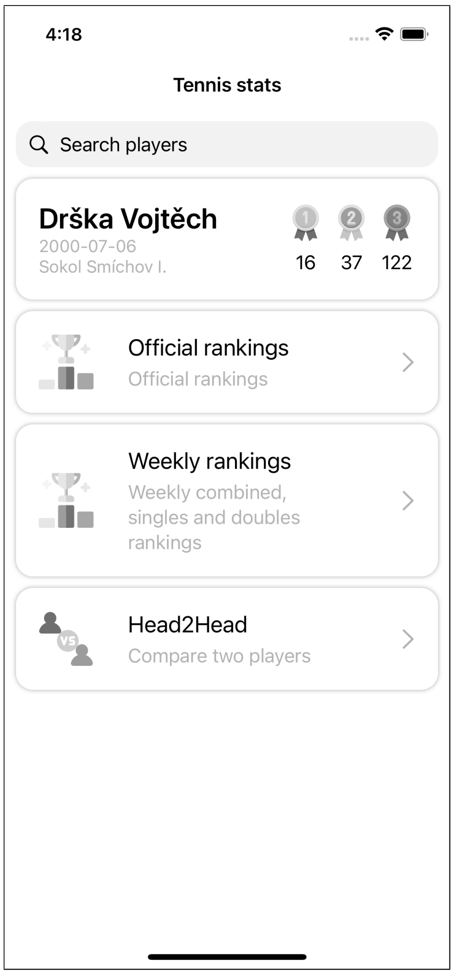
Name: -
Birthdate: -
Club: -
0 0 0

Official rankings
Official rankings

Weekly rankings
Weekly combined, singles and doubles rankings

Head2Head
Compare two players

■ **Figure 2.4** Main Screen – No default player


4:18

Tennis stats

Search players

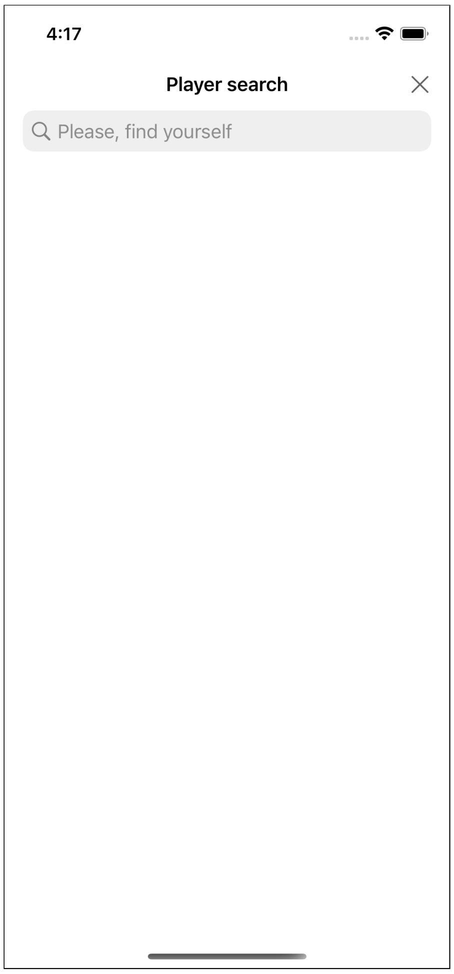
Drška Vojtěch
2000-07-06
Sokol Smíchov I.
16 37 122

Official rankings
Official rankings

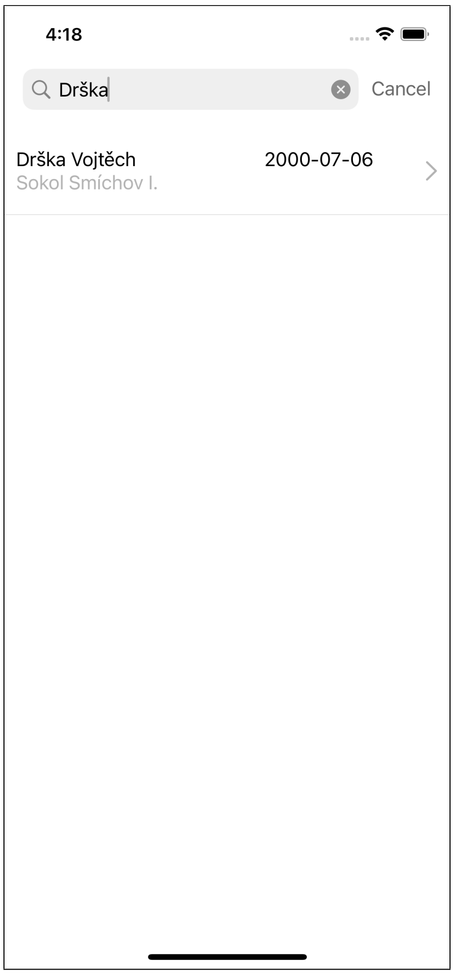
Weekly rankings
Weekly combined, singles and doubles rankings

Head2Head
Compare two players

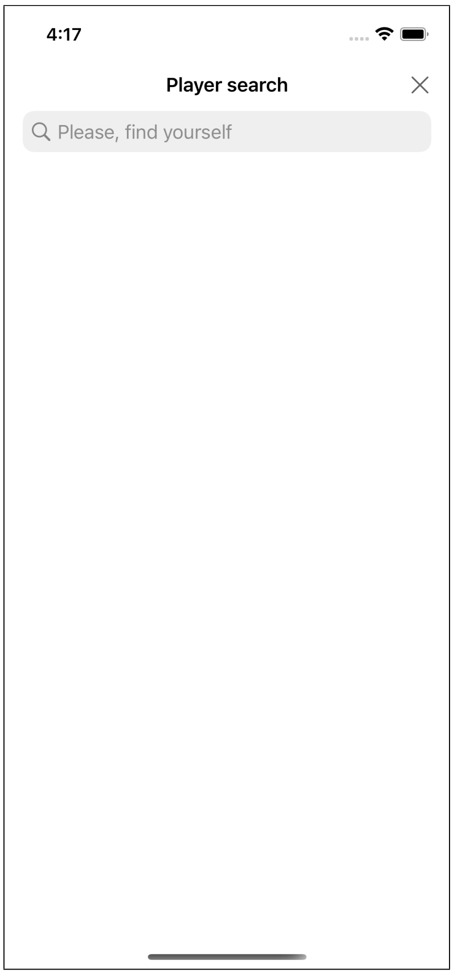
■ **Figure 2.5** Main Screen – Selected default player



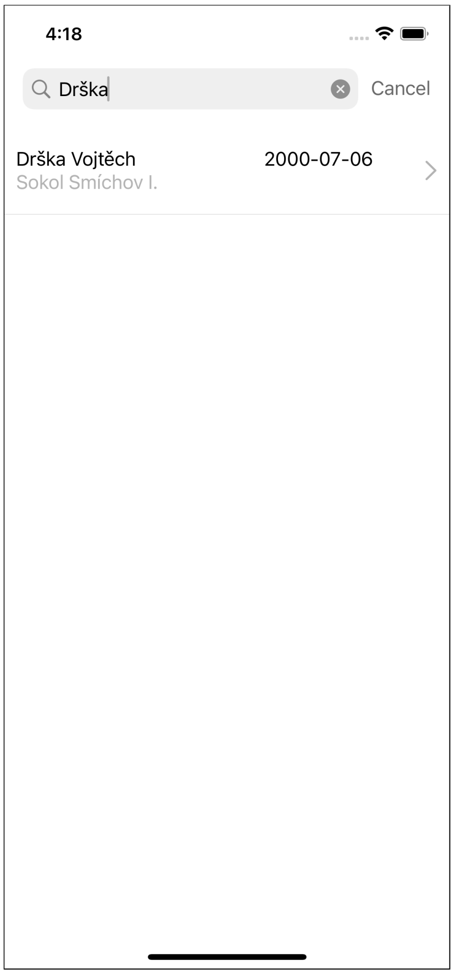
■ **Figure 2.6** Search Screen – Empty



■ **Figure 2.7** Search Screen – With results



■ **Figure 2.8** Player Detail Screen



Drška Vojtěch 2000-07-06
Sokol Smíchov I.

■ **Figure 2.9** Player Detail Screen – Expanded rankings

2.3.5.3 Player Detail Screen

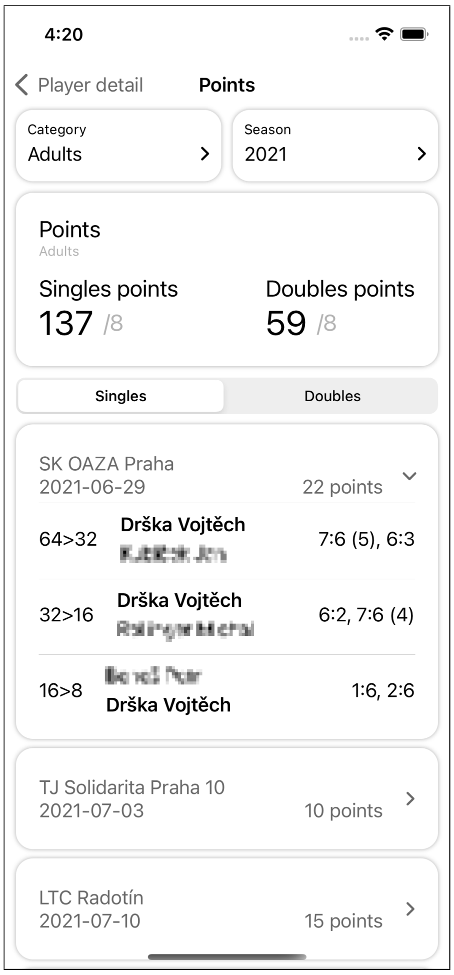
In the figures 2.8 and 2.9 we can see the *Player Detail Screen*. The purpose of this screen is to display the basic statistics of given player and to provide links to screens with more detailed statistics. This screen is required by 1.3.3.4 use case and also by the use cases 1.3.3.5 and 1.3.3.8 as it provides the only means of accessing the *Points Screen* and *Match Screen*.

2.3.5.4 Points Screen

In the figure 2.10 we can see the *Points Screen* of a player. It displays the counted points in given category and season as well as the events the points origin from. It fulfills the use case 1.3.3.5.

2.3.5.5 Match Screen

In the figure 2.11 we can see the *Match Screen* of a player. It displays the matches player by player in the given category and season. It fulfills the use case 1.3.3.8.



4:20

Player detail Points

Category Adults

Season 2021

Points
Adults

Singles points 137 /8

Doubles points 59 /8

Singles Doubles

SK OAZA Praha
2021-06-29 22 points

64>32 Drška Vojtěch
[redacted] 7:6 (5), 6:3

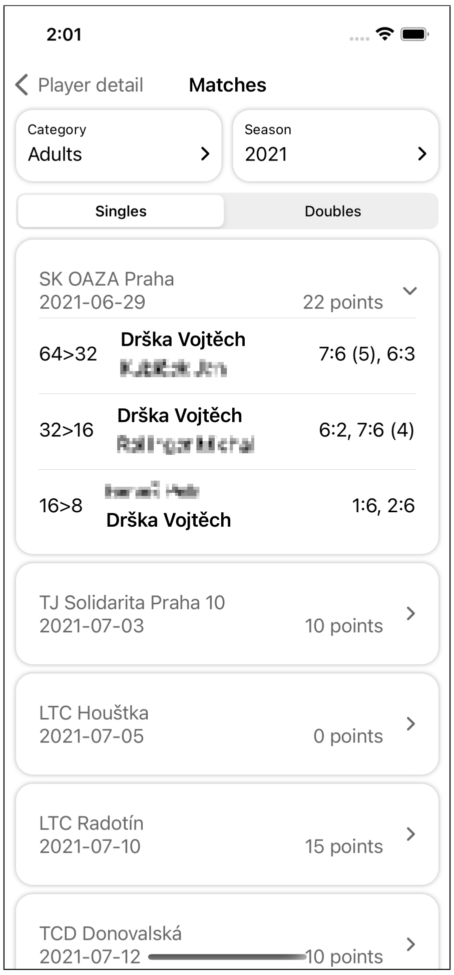
32>16 Drška Vojtěch
[redacted] 6:2, 7:6 (4)

16>8 [redacted]
Drška Vojtěch 1:6, 2:6

TJ Solidarita Praha 10
2021-07-03 10 points

LTC Radotín
2021-07-10 15 points

■ **Figure 2.10** Points Screen



2:01

Player detail Matches

Category Adults

Season 2021

Singles Doubles

SK OAZA Praha
2021-06-29 22 points

64>32 Drška Vojtěch
[redacted] 7:6 (5), 6:3

32>16 Drška Vojtěch
[redacted] 6:2, 7:6 (4)

16>8 [redacted]
Drška Vojtěch 1:6, 2:6

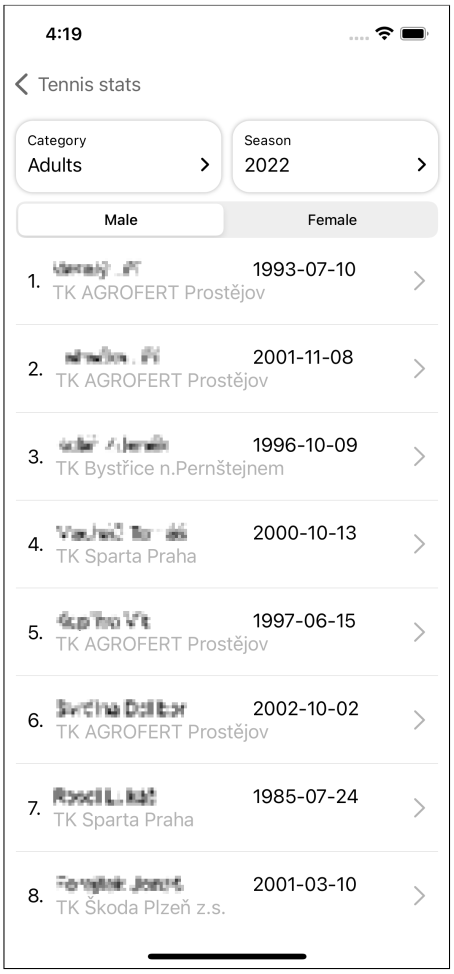
TJ Solidarita Praha 10
2021-07-03 10 points

LTC Houštka
2021-07-05 0 points

LTC Radotín
2021-07-10 15 points

TCD Donovalská
2021-07-12 10 points

■ **Figure 2.11** Match Screen


4:19

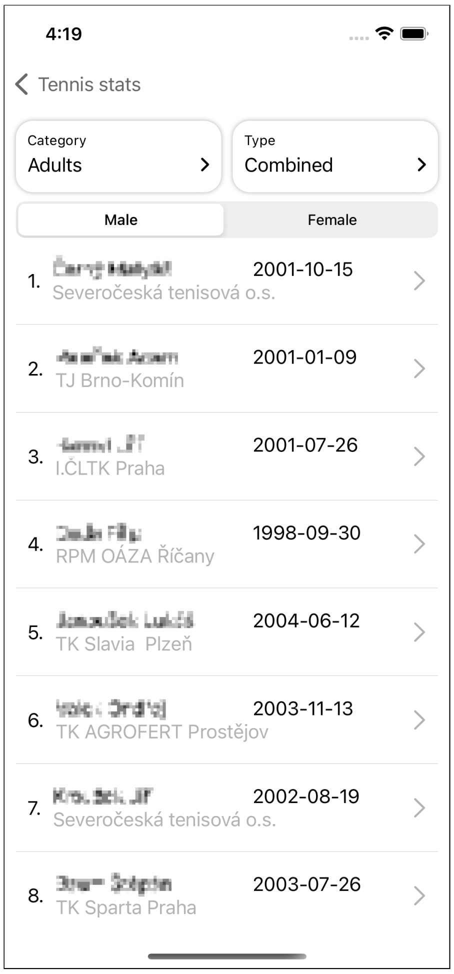
Tennis stats

Category: Adults | Season: 2022

Male | Female

1.	[redacted] TK AGROFERT Prostějov	1993-07-10
2.	[redacted] TK AGROFERT Prostějov	2001-11-08
3.	[redacted] TK Bystřice n.Pernštejnem	1996-10-09
4.	[redacted] TK Sparta Praha	2000-10-13
5.	[redacted] TK AGROFERT Prostějov	1997-06-15
6.	[redacted] TK AGROFERT Prostějov	2002-10-02
7.	[redacted] TK Sparta Praha	1985-07-24
8.	[redacted] TK Škoda Plzeň z.s.	2001-03-10

Figure 2.12 Official Rankings Screen


4:19

Tennis stats

Category: Adults | Type: Combined

Male | Female

1.	[redacted] Severočeská tenisová o.s.	2001-10-15
2.	[redacted] TJ Brno-Komín	2001-01-09
3.	[redacted] I.ČLTK Praha	2001-07-26
4.	[redacted] RPM OÁZA Říčany	1998-09-30
5.	[redacted] TK Slavia Plzeň	2004-06-12
6.	[redacted] TK AGROFERT Prostějov	2003-11-13
7.	[redacted] Severočeská tenisová o.s.	2002-08-19
8.	[redacted] TK Sparta Praha	2003-07-26

Figure 2.13 Weekly Rankings Screen

2.3.5.6 Official Rankings Screen

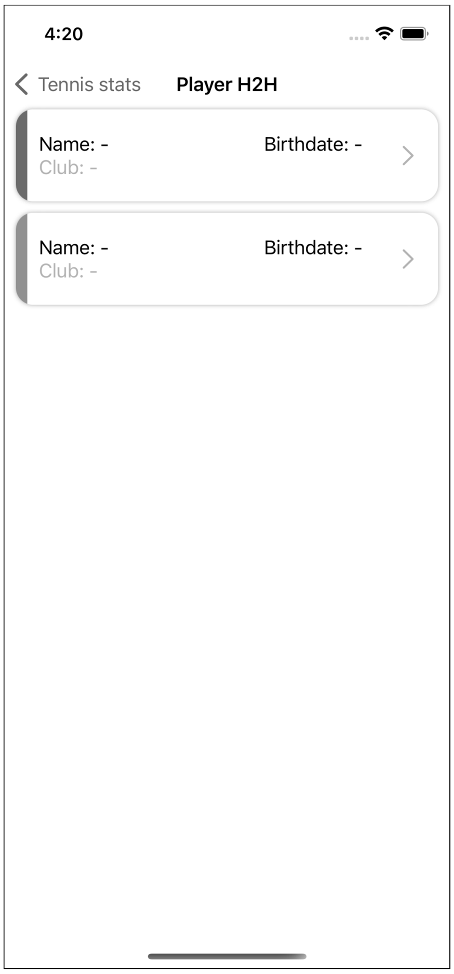
In the figure 2.12 we can see the *Official Rankings* of a player. It displays the official rankings in the given category and season. It fulfills the use case 1.3.3.6.

2.3.5.7 Weekly Rankings Screen

In the figure 2.13 we can see the *Weekly Rankings* of a player. It displays the official rankings in the given category and season. It fulfills the use case 1.3.3.7.

2.3.5.8 Head2Head Screen

In the figures 2.14 and 2.15 we can see the Head2Head Screen without and with selected players respectively. This screen is used for displaying the matches two players played against each other and it fulfills the 1.3.3.3 use case.

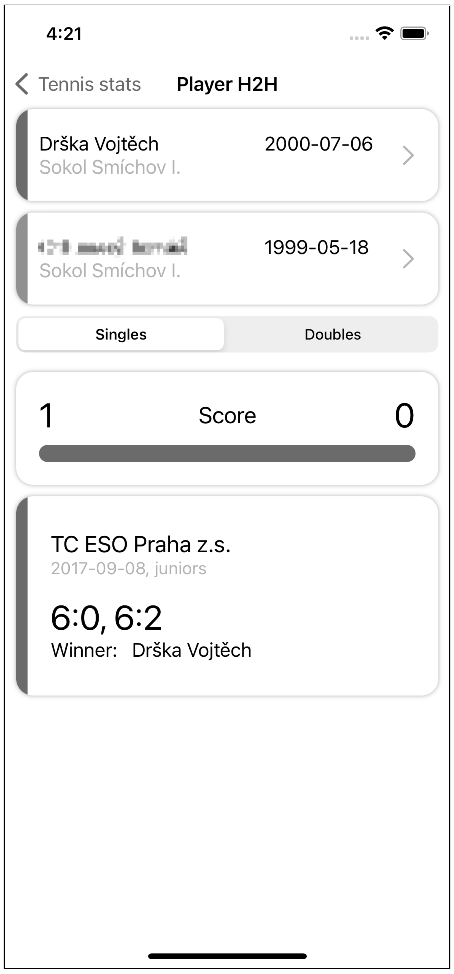

4:20

Tennis stats Player H2H

Name: - Birthdate: -
Club: -

Name: - Birthdate: -
Club: -

■ **Figure 2.14** Head2Head Screen – No Players


4:21

Tennis stats Player H2H

Drška Vojtěch 2000-07-06
Sokol Smíchov I.

[redacted] 1999-05-18
Sokol Smíchov I.

Singles | Doubles

1 Score 0

TC ESO Praha z.s.
2017-09-08, juniors

6:0, 6:2
Winner: Drška Vojtěch

■ **Figure 2.15** Head2Head Screen – Selected players

2.3.6 Mapping screens to use cases

In the table 2.1 the mapping of different screens to use cases can be seen. The use cases 1.3.3.9 and 1.3.3.10 are not displayed in the table as they are not related to the UI of mobile application but rather to the scrappers.

2.4 Scrappers

2.4.1 Programming language

When choosing the best programming language for developing a scrapper there are many options to choose from. The languages that will be considered are the following as these are the most used programming languages for web scraping[15]. Only option that was left out is Go as it has a very sharp learning curve and we have no experience with it what so ever.

- JavaScript
- Python

Table 2.1 Relations between UI screens and use cases

	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
Main Screen		X						
Search Screen	X	X	X	X				
HeadToHead Screen			X					
Official Rankings Screen						X		
Weekly Rankings Screen							X	
Player Details Screen				X	X			X
Match Screen								X
Points Screen					X			

- Ruby

It should also be mentioned that solutions which are able to generate scrappers exist, however these are not suitable in our case as they are quite costly and not very customizable[16].

2.4.1.1 JavaScript

The first option that we have is JavaScript. By itself JavaScript is not very well suited for scrapping data from websites but with the help of Node.JS runtime environment the situation changes dramatically as it enables it to run outside of browser. Apart from using the inbuilt methods for performing requests, there are also third party options such as the Axios JavaScript library which makes working with requests much easier. There are also cases when websites cannot be scrapped using just requests as they may be protected against this type of behaviour. That is when libraries such as Puppeteer come into play. In short what Puppeteer does is it deploys a headless browser which can be operated programmatically. Connecting and using a database is also quite simple with JavaScript as there are many ORM libraries that can be used. When working with website responses, libraries such as Cheerio exist which help immensely with parsing and extracting data from the responses. Even though Node.JS has many great features and libraries it also has its drawbacks the biggest one being the fact that it does not support multi-threading. This could present an issue for us as we might need to scrap multiple pages at a time in order to extract the historical data in a reasonable time. The last drawback that is worth mentioning and is very important in our case is the fact that Node.JS is not ideal for CPU-heavy tasks and thus is more suitable for running shorter one time scrapping jobs. As our scrappers will be reused and will run on a periodical basis as well as they will be scrapping quite large amounts of data, Node.JS is probably not the best way to go.

2.4.1.2 Ruby

Another option that we have is Ruby. Ruby is a great language which even has specific features for web scrapping along with Nokogiri which is a very robust library that apart from parsing and extracting information from HTML responses can also deal with broken responses. Its disadvantages however overweight its benefits. The main disadvantage of using Ruby is the lack of popularity which translates to very little support, resources and community tools compared to languages of mass usage such as Python or JavaScript. Our scrappers also should be deployed in the cloud as server-less functions. This in turn means that the language we choose should be supported by all the major cloud providers such as AWS and Azure. Even though AWS does support its AWS Lambda functions implemented in Ruby[17] the same cannot be said about Azure Functions as they do not support Ruby[18]. The reasons why the mentioned providers are important to us are described in a greater detail in section 2.6. Due to the reasons described above, Ruby also does not seem like the optimal programming language for the implementation of our lambdas.

2.4.1.3 Python

Last option that we have is Python. One of the main reasons as to why choose Python is the requests package which is very easy to use yet capable of everything one can possibly need to do. In case the website cannot be scrapped using requests we have the option of using Selenium whose capabilities are more or less identical to JavaScripts Puppeteer. Connecting and using a database is also quite simple as there are many ORM packages for Python such as SQLAlchemy. When working with the website responses, there is a variety of packages that can be used to parse the responses and extract data from them and also provide great performance. The most notable ones are lxml, beautifulsoup4 and Scrapy. Apart from the positives, the only drawback of Python worth mentioning is its performance as Python is not one of the fastest languages. However, given our options, the performance of Python is not an issue as it is comparable to the ones of JavaScript and Ruby[19]. Another thing that should be noted is that Python has a package called multi-threading which makes it very easy to scrap multiple websites in parallel which will make the process of scrapping data noticeably shorter. There are multiple reasons as to why this is important to us the first one being that lambda functions have a time limit given by the cloud providers in which they have to finish their execution. The time limit is 15 minutes for AWS and 10 minutes for Microsoft Azure. Another reason for doing is to decrease the costs as lambdas are paid by millisecond of execution time.

2.4.1.4 Conclusion

After reviewing all the options above and comparing its benefits and disadvantages Python came out as the most suitable candidate for our scrappers as it is easy to use, has great packages, decent performance and seems to be overall the industry standard for scrapping[15][20][21][22].

2.5 API

2.5.1 Programming language and framework

There are multiple options when choosing a programming language and framework for the API. As there are countless options as far as API development frameworks go, we chose only a few potential candidates. Express.JS was chosen as it is currently the most popular JavaScript framework for back end development[23][24]. Django and Spring Boot were chosen because they are also widely used and we also have prior experience with them which would make the development of API much more time efficient.

- Django
- Express.JS
- Spring Boot

2.5.1.1 Django

Django is a Python framework which is often used for API development. Django is a very popular framework for API development and is used by many well known companies such as Spotify, Mozilla or Instagram. Some of its best features include rapid development, security and great in-built ORM. Being widely used, there are lots of resources and tutorials regarding Django which, alongside with Python, make it quite easy to learn and use. Another feature that should be mentioned is that Django is ”batteries-included” which means that it comes with a variety of different tools and utilities that help when building complex web applications. This last feature is not only Django’s advantage but also in a sense also one of its greatest disadvantages. The fact all the tools are already bundled in Django is great when developing big complex websites as said earlier, because the resources are not that limited and the amount of resources Django and its tools take is negligible in comparison to the resources needed by the application itself. This changes when developing smaller applications as the resources are usually quite a lot limited and at that point the resources taken by Django become far from negligible. Another Django’s drawback is its performance as Python itself is not one of the fastest languages. Another issue that arises when using Python based framework is the fact that Python is a dynamically typed languages which not only makes it slower but also not that well readable compared to statically typed languages like Java[25][26].

2.5.1.2 Express.JS

Another option in the list Express.JS, back end web application framework which runs on Node.JS. As mentioned in 2.4.1 Node.JS is not very well suited for CPU-intensive tasks but it is perfect for I/O focused applications such as web servers. Another advantage of Node.JS is that is has a very active community and therefore there are lots of resources and tutorials available. As for Node.JS’s drawback the most major one is that dealing with relational databases is quite impractical. This is definitely a concern for us as we are using a relational database. Another issue with Node.JS that should be mentioned is that with JavaScript being a dynamically typed language it faces the same issue as Python in not being as readable as statically typed languages. The last concern when using Node.JS is that its API quite frequently changes with updates which results in applications running older versions breaking when they upgrade to the newer ones. This results in developers having to change their code in order to make it work with the newest version. This issue itself is present with all languages however the frequency of its occurrence is the main problem with Node.JS[27][28][29][30].

2.5.1.3 Spring Boot

The last option that we have Spring Boot which is a Java framework that can be used for API development. The most major reason to use Spring Boot is that it can be integrated with the Spring ecosystem. This is very helpful as it provides most of the commonly needed features such as ORM, security related tools, modules that simplify cache configuration or simple connection to database using JDBC drivers. Spring Boot also uses dependency injection which makes not only the development but mainly the testing much easier and faster. Another one of its features is that it is very easily configurable through both XML configurations and Java classes. Spring Boot is also quite time proven as the first version was released in 2014. This means that in case we run into an issue the probability someone else ran into it before us and solved is quite large. Spring Boot applications are developed in Java which means that they tend to be easily readable, and maintainable as Java is a statically typed language. This also means that the applications are ensured a long term support as Java is a very well maintained and developed language with a long past. Spring Boot also has a few of its drawbacks one of them being that it installs many dependencies that are not actually used. This can result in large deployment size especially in smaller projects with many dependencies. Another frequently mentioned concern with Spring Boot is the lack of control that it gives to the developer as it tends to be a bit of a black-box[31][32][33].

2.5.1.4 Conclusion

To conclude, Django does not seem as the ideal framework for our API mainly due to not being the most suitable option for small applications. Both Spring Boot and Node.JS seem like great options for our API. Even though Node.JS presents great features as it would provide us with great performance and little need for resources its tedious cooperation with relational databases is quite the problem. On the other hand, the biggest drawback to Spring Boot seems to be its steeper learning curve. This fortunately is not a problem for us thanks to our previous experience with the Spring ecosystem. Spring Boot not being ideal for larger projects also is not much of a concern as our API is going to be rather small. This leaves with the only major issue being that Spring Boot can be a bit of a ”black-box”. Even though this can be quite frustrating at times, the benefits of Spring Boot far out weight it drawbacks which made decide that it will be used for the development of API.

2.5.2 Architecture

2.5.2.1 REST

REST is a set of architectural constraints that have to be met in order for an API to be considered RESTful. These constraints are as follows.

Stateless
The API is considered stateless when no information regarding the client are stored in between the get request. Each request that the client makes must be made individually, without relation to any other request.

Cacheable
The API responses should be cached as it boosts the performance of the server side making it faster and less heavy on the database. It also improves the user experience of the client applications as the users get the requested data sooner.

Layered
The API must consist of multiple layers in order to make the architecture scalable.

Uniform Interface
This constraint ensures that there is a uniform way to communicate with the API. It consists of multiple rules.

Each resource should have only one URI which should provide a way to fetch all the data that is related to it. No resource should be too large and contain all the data related to it. In order to refer to data related to a resource, links pointing to relative URIs (HATEOAS) should be provided.

The structure of resource representations should follow the structure of selected format such as JSON or XML which ensures that all the responses that API provides are self-explanatory and easily processable by the client.

The resources returned in the responses from API must have a uniform structure which can be used by the clients in order to modify/manipulate the state of resources in the server.

Client-Server
A REST application should be of client-server architecture. This ensures that the two parts evolve independently of each other. The only thing that the client should know about the API are its endpoints.

Our API is designed to be RESTful with one exception. The Uniform Interface constraint contains a rule stating that “In order to refer to data related to a resource, links pointing to relative URIs (HATEOAS) should be provided.”2.5.2.1. This feature is also called HATEOAS and in our case there are multiple reasons as to why not support it. The first and main reason is that our mobile app is developed in Swift which provides very little support for HATEOAS formats. This makes the integration of the API very unnatural and complicated. Admittedly there are some libraries that may help with using HATEOAS APIs. The issue with those libraries is that they are not official developed or maintained by Apple as they are rather hobby projects which are not widely adopted. This makes their usage quite questionable as their development or support may quite likely stop at any moment. Given the fact that our app is supposed to go into production, this could bring many issues in the future. Another reason is related to the fact that the main purpose for HATEOAS is to make changes in routes possible without breaking the clients that use the API as they the traverse the API using returned links and not hard-coded values. Even though this is a great feature it makes sense when the API is used by a variety of different clients. As we are both the developers and the only consumers of the API, we will always know that a change in route will happen in time so we will be able to update the client in order to keep it working. Due to those reasons it was decided that HATEOAS is not beneficial for our application and thus will not be implemented.

2.5.2.2 Endpoints

As our API will only provide data all the endpoints will respond to *GET* requests. As for the MIME types, the API will support application/json. Following is the description of individual endpoints.

/doubles/h2h - this endpoint will returned the doubles matches that the two players specified in the request played against each other,

/official-rankings - returns official rankings of specified season, category and gender,

/players/{playerId} - returns the player specified by *playerId*,

/players/{playerId}/counted-tournaments - returns the tournaments on which player specified by *playerId* gained points counted towards the official rankings in given season and category,

/players/{playerId}/counted-team-matches - returns the team matches on which player specified by *playerId* gained points counted towards the official rankings in given season and category,

/players/{playerId}/tournaments - returns the tournaments in which player specified by *playerId* participated in given season and category,

/players/{playerId}/tournaments/{tournamentId}/doubles - returns the doubles matches played by player of *playerId* on a tournament of *tournamentId*,

/players/{playerId}/tournaments/{tournamentId}/points - returns the points gained by player of *playerId* on a tournament of *tournamentId*,

/players/{playerId}/tournaments/{tournamentId}/singles - returns the singles matches played by player of *playerId* on a tournament of *tournamentId*,

/players/{playerId}/team-matches - returns the team-matches in which player specified by *playerId* participated in given season and category,

/players/{playerId}/team-matches/{teamMatchId}/doubles - returns the doubles matches played by player of *playerId* on a team-match of *teamMatchId*,

/players/{playerId}/team-matches/{teamMatchId}/points - returns the points gained by player of *playerId* on a team-match of *teamMatchId*,

/players/{playerId}/team-matches/{teamMatchId}/singles - returns the singles matches played by player of *playerId* on a team-match of *teamMatchId*,

/players/{playerId}/stats - returns the basic statistics of player,

/players/search - returns the players whose name starts with the passed in term,

/singles/h2h - this endpoint will returned the singles matches that the two players specified in the request played against each other,

/tournaments/{tournamentId} - returns the tournament specified by *tournamentId*,

/weekly-rankings - returns official rankings of specified season, category and gender,

/swagger-ui/index.html - provides the documentation of our API, details of all the previously described endpoints can be found here.

2.5.2.3 Multi-layered Architecture

When building the API we also have to choose the architecture that will be used. When using Spring Boot it is common to use layered architecture in which the layer can communicate only with layers that are one level away from it. The architecture of our API consists of the four layers described below.

Presentation Layer
In this layer all requests made by clients are handled. If there is any JSON passed in as payload of request it is parsed in this layer. In case authentication is present in the application it is also performed here. Also, all the endpoints are defined in this layer in the form of methods with respective annotations. This layer is allowed to communicate only with the Business Layer. The classes which represent this layer are called controllers.

Business Layer
As the name suggests all the business logic is contained in this layer. In case any objects which were parsed by the Presentation Layer from client request are passed in, their validation is performed in this layer. The classes which represent this layer are called services.

Persistence Layers
The Persistence Layer is responsible for handling all the communication with database. It is responsible for converting objects to database rows and vice-versa. The classes which represent this layer are called repositories.

Database layer
This layer represents the actual databases which are being used such as Postgres or MySql. Its only responsibility is to perform CRUD operations on the database.

[34] [35]

2.6 Cloud services

2.6.1 Cloud provider

As our application will be running in the cloud, a cloud provider which will be used has to be chosen. We will be deciding between AWS and Microsoft Azure as these are the two most popular providers at the moment[36].

First of all our requirements on the cloud providers have to be defined so we can choose the most suitable one. It is certain that a virtual server which will be running our API and probably also Redis will be needed. We will also want to host our database in cloud so support of PostgreSQL is very important in our case. It is also needed that the provider enables us to run server-less functions with support for Python runtime as our scrappers are designed this way. It would also be great to automate the deployment of our applications when change happens so a service which would allow us to do sow would be much appreciated. Apart from the features offered by cloud providers there will also be one very important factor which has to be considered and that is the cost of services as required by 1.3.2.6.

After researching both the providers we arrived at the conclusion that both the providers offer all the features we need for marginally different prices and thus they are both suitable for application. The final decision was made based on a personal preference in the favor of AWS as we have previous experience with it which will make the deployment less time expensive.

Chapter 3 Implementation

3.1 Scrappers

3.1.1 Packages

In order to create the scrappers multiple packages will be needed. First of all we will need a package in order to perform requests to the scrapped website as this is not an inbuilt Python feature. The package perfect for this use is called requests. Its main features are that it is very simple to use, is very well documented and is widely adopted amongst the Python community.

Next thing that has to be done is connecting the scrappers to database. For this task a library called SQLAlchemy was chosen. Apart from being widely adopted, SQLAlchemy is often used for its ORM and also has one feature that can be used in our case. If the scrappers were developed while there was no existing DB schema, the best way to go about making one would be creating entities in the scrappers which would represent the DB tables and relationships and generate the schema from them. However, in our case the database schema exists and the database was already created. As we already have the database running we can take advantage of it and use the aforementioned SQLAlchemy feature which will let us generate entities from tables during in runtime and thus will save us quite some time with creating and updating entities.

Another task that needs to be handled is parsing the website responses and extracting data from them. In order to get these features two packages will be used, lxml and re. Lxml is used for parsing HTML code and provides us with multiple options to extract data such as XPath or CSS selectors. In some cases there might not be the need for parsing the HTML from responses as we will be interested in extracting some strings based on their format and regardless of their position. That is where re package is very useful as it enables us to search the responses as strings using regular expressions. As the official website of Czech Tennis Association[1] does not have many CSS selectors such as names or IDs defined, we will often have to rely on its structure and use XPath when scrapping data which is not very ideal. Due to this situation re package will be used on as many places as possible as it keeps on working even if the web page structure is changed.

3.1.2 Scrapping clubs

IDs of clubs from the official website of Czech Tennis Association[1] (hereafter refereed to as cztenis) are ignored as some of the clubs present in historical data are not present in the system anymore and therefore they need to be added to the database during scrapping of other entities such as Tournaments. Club IDs are not used anywhere else in the system so they are replaced by IDs of our own which are generated by sequence.

3.1.3 Scrapping tournaments

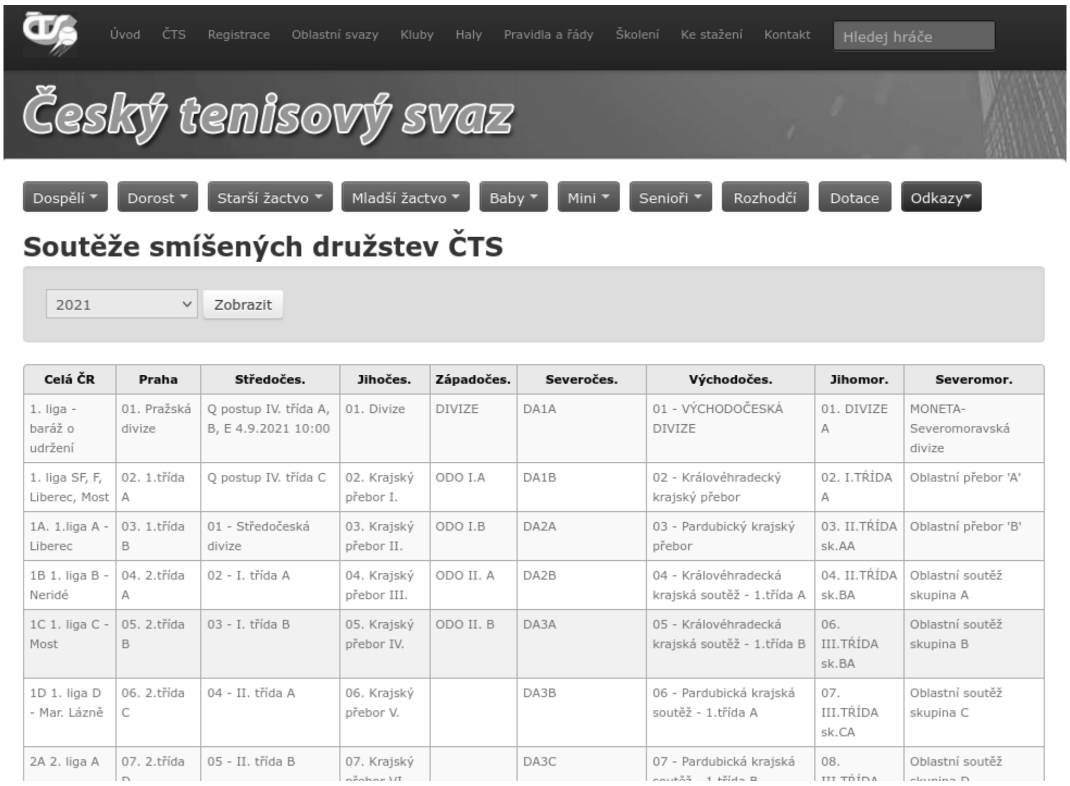
Tournament IDs turned out not to be unique on cztenis which was causing major issues in the database as they were originally used as the primary keys of *tournament* table. As the only way player's matches are bound to tournaments is through their links which contain the IDs we are obliged to store the IDs in some form so we can map the matches to tournaments correctly when scrapping them. The solution that we have decided for at first, was to append the year of season behind the ID thereby making it unique throughout the database and also still being able to use it in relation to cztenis data. This however is not an ideal solution, because some of the newly created IDs do not fit in the value range of Integer. The final solution that we are using in order to prevent duplicate IDs and also overflowing integer boundaries is storing the ID from cztenis in our database as *link_id* as it is part of the link, alongside the season and category, on cztenis when accessing the tournament's details or results. The ID from cztenis is not used as the primary key of the *tournament* table as it is not unique. Instead an auto-generated ID of our own is now used as the primary key of tournaments.

3.1.4 Scrapping teams

To get to all of the teams for specified category and season, we can use the front end of cztenis. After accessing the team competitions page for adults category in 2021 season we are presented with the page displayed in figure 3.1. If we inspect this page, we can see that all the links to different team groups have a very specific format which can be matched by this regex: */[a-z,-]*/druzstva/sezona/[0-9]*/soutez/[0-9]**. Given this fact it would be nice to not parse the page by lxml library and rather use the re package to extract the links using the regex mentioned above, however there is a slight issue with that. If we take a closer look at the figure 3.1 we can see that the position of the group in the table determines what region it belongs to. By parsing the website using regex we are losing this information. Thankfully we do not need the information related to region as it is not displayed anywhere in our application. If the situation was to change and we suddenly needed the information regarding which region does the team belong to, we would not have it, at least not directly. If we take a look at the database model 2.2 we can see that the *region* column is present in the *club* table and as every team has to have its club we can derive the region from it. This means that we can afford to lose the information regarding region when scrapping the groups and therefore can use regex to do so.

While scrapping the teams we also noticed that there are no specific IDs assigned to the teams. This makes the scrapping a little bit more complicated as we have to figure out how to pair players with the correct teams. The best approach seems to consider the category and name as the key identifiers of each team as this combination is unique for among the teams and is also present while we are scrapping the matches of each player.

While scrapping teams, the following problem has emerged. It would make sense to associate each team with its club, although this relationship seems to not be present anywhere in the cztenis and therefore the matching will have to be done by our code through similarity of names. It is possible to find some conventions in the names of the teams such as it usually is the name of its corresponding club appended with some capital letter. Even though it at first seemed like this technique alone will work, we unfortunately ran into issues with some older team names. The main reason for that is, that some of the clubs have abbreviations like a.s., z.s. etc. appended to them. Their respective teams however do not. There is also an issue in the form of multiple blank spaces and sometimes random upper case letters. Thankfully all those issues could be resolved by using functions like *regexp_replace*, *replace* and *lower* in the SQL query that matches the teams to their clubs by their respective names.



Soutěže smíšených družstev ČTS

Celá ČR	Praha	Středočes.	Jihočes.	Západočes.	Severočes.	Východočes.	Jihomor.	Severomor.
1. liga - baráž o udržení	01. Pražská divize	Q postup IV. třída A, B, E 4.9.2021 10:00	01. Divize	DIVIZE	DA1A	01 - VÝCHODOČESKÁ DIVIZE	01. DIVIZE A	MONETA-Severomoravská divize
1. liga SF, F, Liberec, Most	02. 1.třída A	Q postup IV. třída C	02. Krajský přebor I.	ODO I.A	DA1B	02 - Královéhradecký krajský přebor	02. I.TŘÍDA A	Oblastní přebor 'A'
1A. 1.liga A - Liberec	03. 1.třída B	01 - Středočeská divize	03. Krajský přebor II.	ODO I.B	DA2A	03 - Pardubický krajský přebor	03. II.TŘÍDA sk.AA	Oblastní přebor 'B'
1B 1. liga B - Neridé	04. 2.třída A	02 - I. třída A	04. Krajský přebor III.	ODO II. A	DA2B	04 - Královéhradecká krajská soutěž - 1.třída A	04. II.TŘÍDA sk.BA	Oblastní soutěž skupina A
1C 1. liga C - Most	05. 2.třída B	03 - I. třída B	05. Krajský přebor IV.	ODO II. B	DA3A	05 - Královéhradecká krajská soutěž - 1.třída B	06. III.TŘÍDA sk.BA	Oblastní soutěž skupina B
1D 1. liga D - Mar. Lázně	06. 2.třída C	04 - II. třída A	06. Krajský přebor V.		DA3B	06 - Pardubická krajská soutěž - 1.třída A	07. III.TŘÍDA sk.CA	Oblastní soutěž skupina C
2A 2. liga A	07. 2.třída	05 - II. třída B	07. Krajský		DA3C	07 - Pardubická krajská	08.	Oblastní soutěž

■ **Figure 3.1** Cztenis – Groups

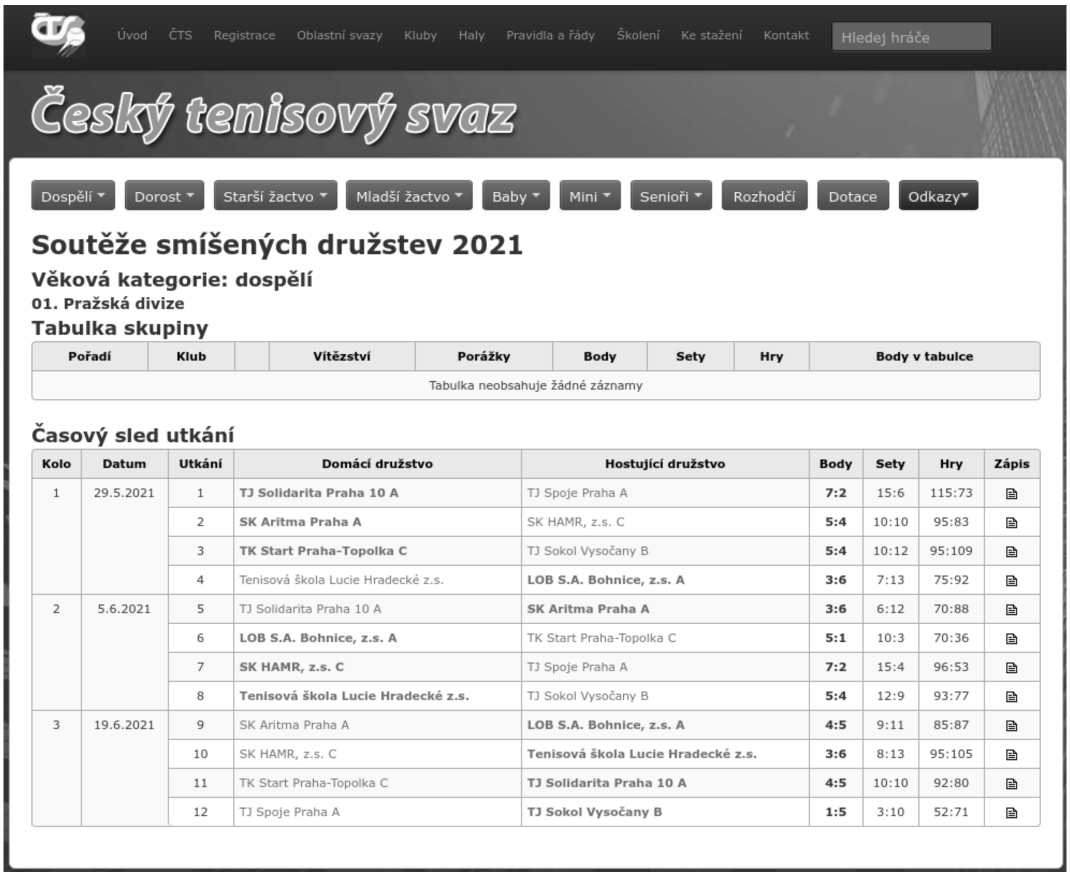
3.1.5 Duplicate matches

While scrapping the players singles matches, each match would be scrapped twice. Once on the profile of the first player and once on the profile of the second player. Presuming we have a unique constraint on the *singles* table it would not introduce duplicates into our database but it would definitely cause some performance overhead while inserting duplicate entries of matches. This issue is solved by fetching the match from player's profile only if the player is the winner of the match.

The same issue as the one described above would occur when scrapping the doubles, only now we would be trying to insert each of the matches four times impacting the performance even more. Through the process of trial and error, it soon became obvious that the cztenis system shows the players in pair in a specific order on both players' profiles. Let's say that the pair in bold were the winner of the match:
player1, player2 def player3, player4
Then on both player1's and player2's account, the order of players would be the same. The solution to this problem therefore is to scrap the match only from the profile of a player, who won the match and is the first listed in the winning pair.

3.1.6 Scrapping player IDs

Scrapping players turned out to be quite a bit more difficult than expected. First we tried scrapping the players from tournament information. This way we would be missing all players that have not ever played a tournament which seems to be a fair amount. Scrapping players from team line-ups would be quite complicated and also not very optimal, given the fact that



Soutěže smíšených družstev 2021

Věková kategorie: dospělí

01. Pražská divize

Tabulka skupiny

Pořadí	Klub		Vítězství	Porážky	Body	Sety	Hry	Body v tabulce
Tabulka neobsahuje žádné záznamy								

Časový sled utkání

Kolo	Datum	Utkání	Domácí družstvo	Hostující družstvo	Body	Sety	Hry	Zápis
1	29.5.2021	1	**TJ Solidarita Praha 10 A**	TJ Spoje Praha A	**7:2**	15:6	115:73	🗎
		2	**SK Aritma Praha A**	SK HAMR, z.s. C	**5:4**	10:10	95:83	🗎
		3	**TK Start Praha-Topolka C**	TJ Sokol Vysočany B	**5:4**	10:12	95:109	🗎
		4	Tenisová škola Lucie Hradecké z.s.	**LOB S.A. Bohnice, z.s. A**	**3:6**	7:13	75:92	🗎
2	5.6.2021	5	TJ Solidarita Praha 10 A	**SK Aritma Praha A**	**3:6**	6:12	70:88	🗎
		6	**LOB S.A. Bohnice, z.s. A**	TK Start Praha-Topolka C	**5:1**	10:3	70:36	🗎
		7	**SK HAMR, z.s. C**	TJ Spoje Praha A	**7:2**	15:4	96:53	🗎
		8	**Tenisová škola Lucie Hradecké z.s.**	TJ Sokol Vysočany B	**5:4**	12:9	93:77	🗎
3	19.6.2021	9	SK Aritma Praha A	**LOB S.A. Bohnice, z.s. A**	**4:5**	9:11	85:87	🗎
		10	SK HAMR, z.s. C	**Tenisová škola Lucie Hradecké z.s.**	**3:6**	8:13	95:105	🗎
		11	TK Start Praha-Topolka C	**TJ Solidarita Praha 10 A**	**4:5**	10:10	92:80	🗎
		12	TJ Spoje Praha A	**TJ Sokol Vysočany B**	**1:5**	3:10	52:71	🗎

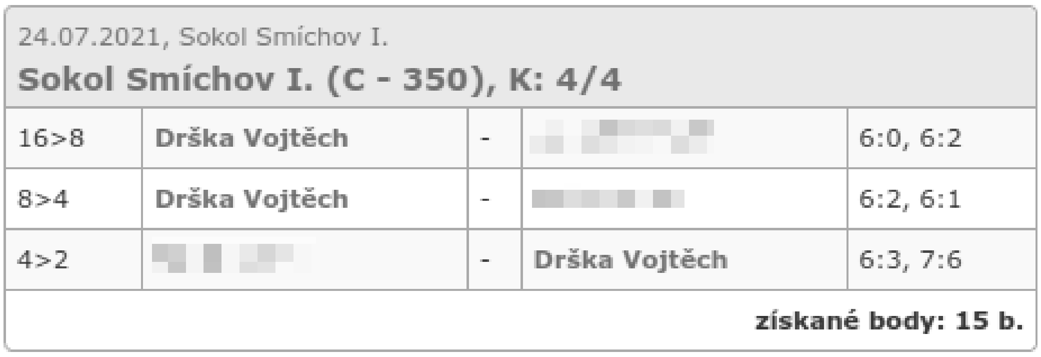
■ **Figure 3.2** Cztenis – Group with empty table the line-ups quite often don't contain the links to the players' profiles and are simply just a text. That is an issue for us as the links are the only means to get to the players' profiles. Another thing that complicates scrapping player data is that the player IDs in cztenis system do not go in order, meaning that there can be players with IDs 1 and 100 while there are no players in the exclusive interval of (1, 100). After doing some research we figured that player IDs go in an ascending order depending on the time when player was added to the system. Given this information we can expect that the youngest players have the highest IDs for the most part and by using the trial and error method we found out, that none of the IDs of the youngest players surpassed the value of 1 200 000. With this information in hand, we have decided that the best way to fetch all existing player IDs will be to ask the server for a player with each ID in interval (1, 1.2 * 10e6) inclusive and decide if such player exists on its response. Even though this method is not the most effective one, it seems to be the most reliable. Using multi-threading and running the scrapper on 5 threads simultaneously we are able to check around 15 IDs per second which means we are be able to get all players' IDs in roughly 22 hours. Given the fact that we are scrapping this data only once makes 22 hours acceptable. It should be noted that we could increase the number of threads and probably get the IDs much faster. The reason as to why we are not doing so is to not put too much stress on the cztenis's server. Even though we expect 5 threads to be safe and not cause any problems to the servers we want to be sure that we do not cause any negative impact on user experience on cztenis. In order to ensure so the scrappers were ran in three stages mainly throughout the night as we expected not many users to be using cztenis at that time. Because of the larger amount of requests needed to obtain all player IDs, we are not scrapping the players' information right away, but we first scrap all the IDs and than use those to scrap individual players' information and their matches.

3.1.7 Multiple occurrences of points

One of the metrics scrapped are the points that player gained on a tournament. When scrapping the data, especially when doing so periodically, it has to be ensured that no points in the database are duplicate as this would negatively affect many features in the app such as the sum of points gained per season or the weekly rankings. This was at first ensured by a unique constraint in the database stating that the combination of tournament's ID, player's ID and match type (singles or doubles) must be unique for every row. Even though this constraint seemed to make sense the following edge case appeared when testing the scrappers. There is a very specific situation when one player may gain points twice on a tournament even though they played only singles/doubles. This happens at tournaments of the highest rank on which qualification is played as the points from qualification and main draw are awarded separately. An example case would be when a player wins the qualification and then wins at least one round in the main draw. The player would then be awarded points twice as they gained them in both qualification and main draw. Even though this situation happens quite rarely it has to be taken into account by both the scrappers and database schema. As the unique constraint cannot be used due to the reasons specified above an alternative approach is used.

When the match data is scrapped from solo events, it is done from the profile of each player. The figure 3.3 shows a table which can be found in the profiles of players. This table presents all the matches that player played on tournament and the number of points they were awarded. The process of extracting data from the displayed table is the following. First we extract tournament details and find the tournament in our database. Then we start to insert the matches one by one and in the end we insert the awarded points.

The solution to our problem hides in this process. It may not be possible to have a unique constraint on the points table, however it most certainly is possible to have it on singles and doubles tables as the combination of players' IDs and tournament ID is always unique. This means that we are always able to determine if the table was already scrapped by having the unique constraints on our singles and doubles tables. When a table was scrapped, the first



24.07.2021, Sokol Smíchov I.
Sokol Smíchov I. (C - 350), K: 4/4

16>8	**Drška Vojtěch**	-	[redacted]	6:0, 6:2
8>4	**Drška Vojtěch**	-	[redacted]	6:2, 6:1
4>2	[redacted]	-	**Drška Vojtěch**	6:3, 7:6
				získané body: 15 b.

■ **Figure 3.3** Cztenis – Tournament table

match we try to insert will throw a unique constraint violation exception a we will terminate scrapping of the whole table. As the process of scrapping and inserting matches occurs before the insertion of points, we will effectively avoid inserting duplicate entries into the points table.

3.1.8 Generic exceptions

Throughout the scrappers generic exceptions are caught in try/except blocks in many places. Even though this is usually considered a bad practice we have a good reason for doing so. Scrappers in general can throw a variety of exceptions in many, sometimes unexpected, places. This is especially true in our case as we use XPaths often when scrapping the data and thus rely on the structure of cztenis to be correct and consistent. This means that if the structure of cztenis changes slightly or is just inconsistent, which we have encountered many times so far, it may make the scrappers throw an exception. As we do not know what kind of exception can the change of cztenis cause we have to be, more or less, prepared for any existing one. The way that we handle this situations in scrappers is by catching generic exception and then carefully logging them. As the scrappers run in AWS as Lambda functions the logs containing the thrown exceptions from each run of a function can be found in Cloud Watch logs. One could argue, and be right, that by catching generic exceptions we catch not only the exceptions caused by source inconsistency/changes but also exception caused by database unavailability or network failure. Continuing to scrap the data after such exceptions occur could cause the database to contain only some of the new data and thus endanger its integrity. As this is a situation that can happen we figured the best way to go about it is to create weekly snapshots of the database. This allows us to return to a different point in time before the data became corrupted.

3.2 Mobile application

3.2.1 Mock API

While making the application it would be great to have the API providing us some random statistics as testing the UI on random values could help us discover some issues right away. It would also be nice to implement the communication with API right away. Our API is not yet implemented and doing so now does not seem like a great idea as we may find that some changes have to be done to the structure of our database which would make us update the API every time it happens. It would also be unnecessarily complicated to develop the API in a way that it returns random data so we can test our UI. Given our requirements it seems best to use a Mock API at the moment.

It was decided to use Postman as it can not only perform requests but also run mock API servers. The mock API servers are running on servers provided by Postman and so they are accessible from anywhere. Even though this solution worked well as both the iPhone virtual device and real device were able to communicate with the API without any issues complications arose when it turned out that Postman limits the monthly amount of API calls in the free tier to 1 thousand which we ran through in a matter of days. This did not seem as a big issue in the beginning as the limit appeared to be present due to the mock API being run Postman's server and that running it on localhost would help us get rid off the API call limit. This solution ended up being infeasible as Postman does not support running mock servers on localhost. Given this restriction Postman is no longer a viable option for us as the only way to get around the limit is to pay for a premium plan.

As this seems like an unnecessary cost and is against the 1.3.2.6 requirement, we started to look for an alternative and ended up deciding between multiple options. Those options were namely Nock, Mockoon and Beeceptor[37]. With Beeceptor we arrived at the same issue as with Postman because it allows only 50 requests per day which does not fit our requirements. Both Nock and Mockoon are great tools and we ended up using Mockoon in the end as it seems to be better maintained than Nock and also supports faker module in it's responses which is exactly what we need.

Moving all the responses from Postman to Mockoon took just a few minutes and after doing so we have a running mock API on our localhost with an unlimited amount of requests. One thing that has to be pointed out is that Mockoon does not support having URL query parameters in URL which is a bit disappointing. On the other hand, even though we cannot have query parameters in the URL, they can be accessed whilst constructing a response to the request which is admittedly sufficient for a mock API.

There is also one more problem that has to be addressed regarding the mock API and that is the following. The mock API is now running on localhost and so it is not accessible outside of the local network. This is an issue, as we want to test the application on a real device whilst developing it. After doing some research we found out, that computer's network can be shared with the iPhone when it is connected to the computer by cable[38]. Even though this would be quite the perfect solution it unfortunately did not work in our case. We were not able to figure out why this solution did not work for us but the most probable reason is that we are using fairly dated hardware. After looking for another solution we found out about a project called ngrok. Using ngrok is extremely easy and what it does in short is that it exposes a port of your choice and connects it to a publicly accessible address in it's cloud over an encrypted tunnel. After using Mockoon with ngrok, the mock API is accessible both on a local network as well as from any real device connected to the internet.

3.2.2 Custom loader animation

In order to make the application more original we wanted to implement a custom loader in the form of a spinning tennis ball in the mobile application. Even though implementing custom loader should not be a difficult task it proved to be quite the opposite. After the implementation the behavior of the loader unfortunately is not as expected. Even though the ball is of the right size and is rotating it is also sliding down the screen and reappearing on the top after a second. This behavior is also happening only on some of the screens so it is probable that the structure of the screen is playing a role in this undefined behaviour. After double-checking the code and doing some research we arrived at a related post on the Apple developer forum[39]. It seems as there is currently a bug present in the SwiftUI framework which causes the animations to behave in an undefined manner under some circumstances. Using the *withAnimation* method makes the loader work on some of the screens which were previously broken, however there are still some screens that remain broken. For those reasons we decided to replace the custom loader by the standard one provide by SwiftUI as the custom loader has the tendency to be problematic and is not worth the risk.

3.3 API

3.3.1 Models

After creating a new project using Spring initializr, the first thing we have to do is create models of different entities in the database.

As an ORM framework, we decided to use JPA as it is quite straight forward and very well documented. As we have already filled our database with data from cztenis using scrapper, we already know the structure and relationships of all the entities and therefore just have to create models that map correctly to the database schema. This can definitely be done by hand, however we also have the option of using JPA Buddy, an IntelliJ IDEA plugin, to generate all the models for us which seems to be the better option as doing it by hand would be quite tedious and error prone compared to generating them.

Our code should also be as readable and easily maintainable as possible and having lots of unnecessary code in our classes would not help. That is why we will be using project Lombok in our classes. In short, project Lombok is a Java library that allows you to include methods that can be generated (getters, setters, constructors etc.) through annotations and therefore saves you from explicitly typing them out or generating them inside of the class.

Even though project Lombok is an amazing library, there are some things that we need to keep in mind when working with it, especially while using it alongside JPA. As all our models need to have getters, setters and no-argument constructors and having *equals* and *hashCode* methods could also come in handy, *@Data* Lombok annotation could seem like a good solution. In our case, however, we should not use the *@Data* annotation as using *@Data* annotation with entities could cause not only performance and memory related issues, but also could (and will) break *HashSets* and other structures consisting of our entities. The main issue of the *@Data* annotation are the generated *equals* and *hashCode* methods which cause the previously mentioned bugs when used with JPA entities. This issue is very well known and is explained in detail in the following resource [40]. With this information in hand we now know that a better solution is to use only *@Getter* and *@Setter* annotation and write the *hashCode* and *equals* methods ourselves if necessary.

3.3.2 Team match model

After creating the *TeamMatch* model and running the application we were greeted by an exception saying that a column category is mapped multiple times. After reviewing the *TeamMatch* model, we could see that we are actually joining on one column three times with different attributes, namely category, team1 and team2. The *Team* entity has a composite primary key in the form of *name* and *category* attributes. We decided to add an auto-generated integer primary key to Team entity which, considering the fact that we are still in the beginnings of the application development, seems like quite a hassle free solution and will be much more easily maintainable in the future. Moreover, having composite primary keys is in general considered a bad practice and was admittedly a bad design choice to begin with.

3.3.3 Participation

When scrapping the data from cztenis we did not scrap information regarding players' participation in tournaments and team matches. The reason for doing so is that this information is already present in our database as it can be derived from the matches that the players played. As the scrappers are quite error prone and rely heavily on the structure of cztenis we decided that handling participation in database is also better as it makes the scrappers at least a bit less complex. The initial though was to get and obtain the information only for a specific player on demand (when the API asks for it). As both the *Singles* and *Doubles* tables have couple hundred thousand records it was thought that generating the participation for all players will take a long time. This theory proved to be wrong as the process of generating the participation for all players takes roughly 15 seconds when using the procedure shown in 3.3.3 which is significantly better than anticipated. Given the great performance of the procedure it was decided to truncate and recreate all the participation each time the data in database is updated. Even though this solution is not the most efficient one it is much simpler to implement than trying to obtain the participation data whilst scrapping the data and therefore it is less error prone. It should also be noted that the procedure will be running only a couple times a week and therefore its efficiency is not much of a concern to us.

3.3.4 Gender of player

An important attribute which the players do not so far have is a gender. This attribute is required as the weekly rankings must be split into male and female rankings. Given the fact that two players playing against each other always have to be of the same gender having combined rankings would not really make sense as it would not bring in any information of value. The problem we face now is that when we view a player's profile, there is no sign of their gender whatsoever 3.4.

Thankfully, there is a way to derive the gender for majority of the players and that is through the tournaments that they have played. In the following figure 3.5, it can be seen that even though the gender of tournament is not explicitly stated anywhere, its positioning in the list of tournaments gives this information away. That is if the tournament is in the left column, it is a male one and if it is in the right column, it is a female one. Fortunately we thought of this whilst scrapping the tournaments and therefore included this information in our database. Even though this solution helps us determine the gender of majority of players, it is no "silver bullet" as there are quite a few players that play only team matches and do not play tournaments. For most of those players there is a way to determine the gender in a different manner. As stated above, player's opponent is always of the same gender as they are and so all we have to do is find one opponent whose gender could have been derived from a tournament a take the gender from them. After using these two methods we are still left with a few players whose gender we can not derive from anywhere. Even though this situation is unfortunate, those players are probably not the most active ones as the inability of deriving a gender implies that they have never participated in a tournament and therefore the chance of them using our app is quite marginal. We will consider all those players an "edge-case" which does not need to be addressed.

In order to implement this solution in the simplest and most efficient manner we use a stored procedure. The way the procedure works is that it performs the steps described above repeatedly till the number of players with known gender changes after the steps are performed. The procedure is ran each time the participation are refreshed as new tournament participation are the only way the gender can be determined for more players than previously.

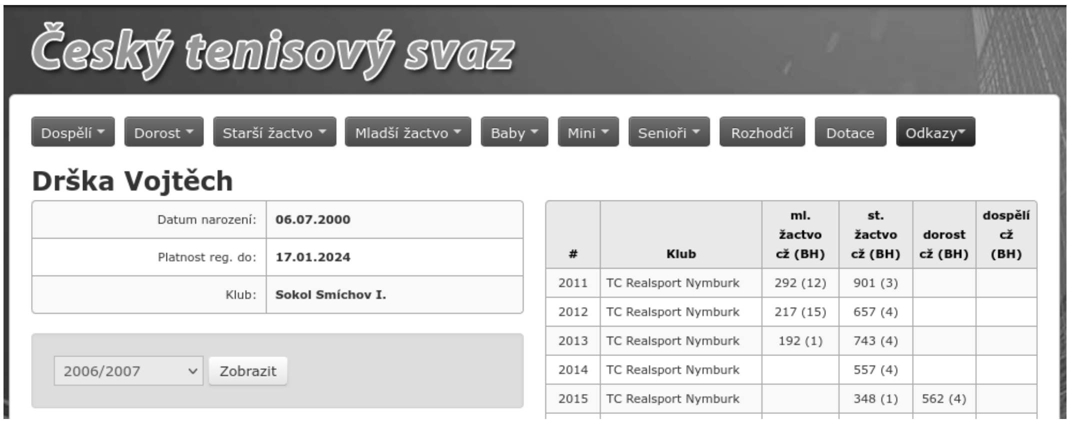
3.3.5 Weekly rankings

One of the features that our API will provide is weekly rankings. As the query that creates the weekly rankings is quite complex, the time that it takes to execute it can be up to two seconds. It definitely would not be ideal to let the user wait for two seconds each time they want to view the weekly rankings. It would also be quite inefficient to compute the weekly rankings for each user individually as the result of the query will be the same to all of them.

We also know that the rankings will not be updated frequently (once a week as the name suggests) and therefore the query results will be valid for quite a while. The optimal solution

■ **Code listing 3.1** Refresh participation procedure

```
create procedure refresh_participation()
    language plpgsql
as
$$
BEGIN
    truncate table participation;
    set session_replication_role = 'replica';
    insert into Participation (player_id, tournament_id)
    select *
    from (select distinct winner_id, tournament_id
          from singles
          union
          distinct
          select distinct loser_id, tournament_id
          from singles
          union
          distinct
          select distinct loser_id1, tournament_id
          from doubles
          union
          distinct
          select distinct loser_id2, tournament_id
          from doubles
          union
          distinct
          select distinct winner_id1, tournament_id
          from doubles
          union
          distinct
          select distinct winner_id2, tournament_id
          from doubles) as participations
    where tournament_id is not null;
    insert into Participation (player_id, team_match_id)
    select *
    from (select distinct winner_id, team_match_id
          from singles
          union
          distinct
          select distinct loser_id, team_match_id
          from singles
          union
          distinct
          select distinct loser_id1, team_match_id
          from doubles
          union
          distinct
          select distinct loser_id2, team_match_id
          from doubles
          union
          distinct
          select distinct winner_id1, team_match_id
          from doubles
          union
          distinct
          select distinct winner_id2, team_match_id
          from doubles) as participations
    where team_match_id is not null;
    set session_replication_role = DEFAULT;
END
$$;
```

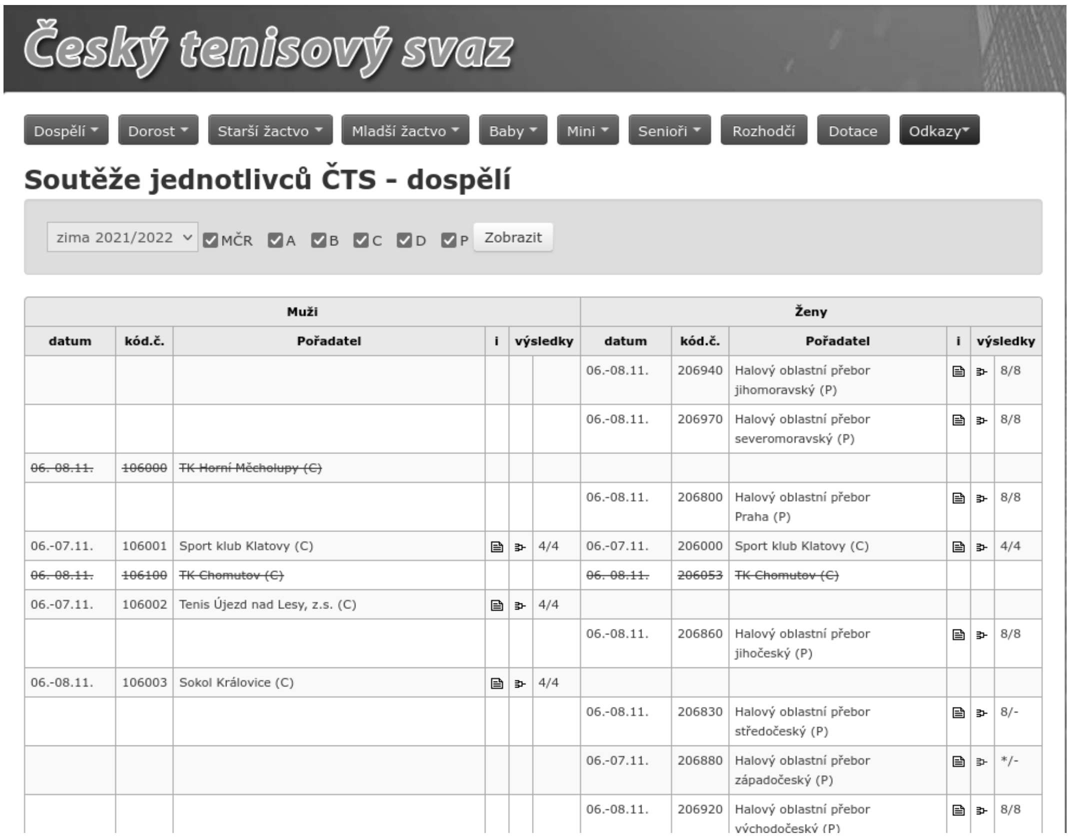


Drška Vojtěch

Datum narození:	**06.07.2000**
Platnost reg. do:	**17.01.2024**
Klub:	**Sokol Smíchov I.**

#	Klub	ml. žactvo cž (BH)	st. žactvo cž (BH)	dorost cž (BH)	dospělí cž (BH)
2011	TC Realsport Nymburk	292 (12)	901 (3)		
2012	TC Realsport Nymburk	217 (15)	657 (4)		
2013	TC Realsport Nymburk	192 (1)	743 (4)		
2014	TC Realsport Nymburk		557 (4)		
2015	TC Realsport Nymburk		348 (1)	562 (4)	

■ **Figure 3.4** Cztenis – Player detail



Soutěže jednotlivců ČTS - dospělí

Muži						Ženy					
datum	kód.č.	Pořadatel	i	výsledky		datum	kód.č.	Pořadatel	i	výsledky	
						06.-08.11.	206940	Halový oblastní přebor jihomoravský (P)	🗎	⊳	8/8
						06.-08.11.	206970	Halový oblastní přebor severomoravský (P)	🗎	⊳	8/8
~~06.-08.11.~~	~~106000~~	~~TK Horní Měcholupy (C)~~									
						06.-08.11.	206800	Halový oblastní přebor Praha (P)	🗎	⊳	8/8
06.-07.11.	106001	Sport klub Klatovy (C)	🗎	⊳	4/4	06.-07.11.	206000	Sport klub Klatovy (C)	🗎	⊳	4/4
~~06.-08.11.~~	~~106100~~	~~TK Chomutov (C)~~				~~06.-08.11.~~	~~206053~~	~~TK Chomutov (C)~~			
06.-07.11.	106002	Tenis Újezd nad Lesy, z.s. (C)	🗎	⊳	4/4						
						06.-08.11.	206860	Halový oblastní přebor jihočeský (P)	🗎	⊳	8/8
06.-08.11.	106003	Sokol Královice (C)	🗎	⊳	4/4						
						06.-08.11.	206830	Halový oblastní přebor středočeský (P)	🗎	⊳	8/-
						06.-07.11.	206880	Halový oblastní přebor západočeský (P)	🗎	⊳	*/-
						06.-08.11.	206920	Halový oblastní přebor východočeský (P)	🗎	⊳	8/8

■ **Figure 3.5** Cztenis – Tournament list to all the issues and constraints mentioned above seem to be materialized views as they provide us with exactly the functionality that we need. As we need to have weekly rankings for each combination of gender and category, creating the materialized views manually is not a viable option as we would need to recreate them each week. This can be achieved by using a stored procedure.

Now that we have all the materialized views created we also have to create endpoints that will be using them. As we started implementing the endpoints two issues, preventing us from using the JPA, arose immediately. The first of them is that in order to create an entity in JPA we also have to define the table name that it is be representing. That is something that we do not really want as there are multiple tables with identical row structure that will be represented by this entity. Admittedly, there is a way to define multiple tables that the entity is representing, however we deem that solution as not very elegant in a sense that all the table names would be hard-coded. That is something we do not want, as the procedure that creates the materialized views creates their names dynamically based on the values that are present in AgeCategory and MatchType tables. Therefore, each time we would change the values in those tables, we would also have to manually change the hard-coded table names. The second reason is that there is a bug in Hibernate which caused it to not see materialized views during schema validation. Even though this issue can be solved by turning the validation off it is not something that we should be doing in our project, most certainly not in the development phase.

Given the issues described above, we decided to take a different path. Our project is using Spring Boot which is able to auto-wire us an instance of entity manager. From entity manager we can then execute any queries that we need effectively eliminating the need for entity and extension of JPA repository. Even though this approach solves our problem it also comes for a price. The main drawback is that we need to specify the table name from which we are querying the rankings from dynamically which prevents us from using the *prepareStatement()* method as it enquotes all the parameters that are present in the SQL statement in order to sanitize it. As the table name cannot be enquoted, we have to use an alternative in the form of *createNativeQuery()* method. This method is unfortunately not safe from SQL injection and when putting the table name directly into the string, we are creating a security vulnerability. As we do not want to have any vulnerabilities in our code, we decided to take a different approach yet again.

The solution we chose in the end is to create a table called *weekly_rankings* and insert the rankings of all categories, genders and match types into it. This results in having about 100 000 records in the table, but given the fact that all the queries filter the entries by columns which have only a few distinct values PostgreSQL is able to optimize them to the point that such queries take sub 5 milliseconds. This means that even though we have to filter through the rankings each time we want to fetch them it creates very little performance overhead. The main advantage of this solution is that we can map an entity to the table and use JPA repository to query all the rankings we need thus avoiding the creation of any security vulnerabilities in the process which is the main priority.

3.3.6 Caching layer

When implementing caching in the API it had to be decided what layer to cache the methods at as were three options:

- Controller
- Service
- Repository

Even though there were three options in our case two of those were not very suitable, namely Service and Repository layers. The reason for this is that the methods in both of these layers have entities as their return values. This is an issue as many of the entities have lazy loaded collections as a part of them. There are multiple methods in both the Service and Repository layers that are used on multiple places in our code. On those places the lazy-loaded parts of entities are mostly not used but there are some places where the lazy-loaded parts are required. This is an issue as we would have to include the lazy-loaded parts of entities in the cache all the time even though they would be only used sometimes. Due to those reasons it seems to be more appropriate to cache the methods on the Controller layer as these return DTOs. The queries that are the main reason for caching are the more complex ones consisting of joins across multiple tables. All such queries are cached right away and correctly when implementing the caching on Controller layer. The only drawback to this solution is that many methods in controllers request a player by ID from the *Player Service* in order to check if it exists. This means that the same player will actually have to be found in the database for each of the methods as the service layer is not cached. This, however, is a minor issue given the fact that the *player* table has roughly 100 000 entries and the ID of player is its primary key which results in the player being found rather swiftly.

3.3.7 API Documentation

In order to make the API easily usable and also to simplify its integration to the mobile application we need to create a documentation for it. Even though this can be done by hand it is not an optimal solution as the documentation has to be updated each time we change the API. As we are using Java with Spring Boot there is also an option to make use of *springdoc-openapi-ui* library which automatically generates a Swagger documentation from our code. This library is very easy to use as the only thing that has to be done is adding it as a dependency to our *pom.xml* file.

When using this library we had to make slight change to our DTOs. As the controllers don't return the entities but rather their respective DTOs, the names of the objects also appeared with the DTO suffix in the documentation. This issue was fixed rather easily as all that had to be done was to add annotation to the DTOs in the form of *@Schema(name = "desired_name").*

3.4 Cloud - AWS

3.4.1 Database

As we wanted to move the application to cloud in order to improve its reliability and availability we also had to move the database. AWS allows us to easily create and run database using AWS RDS feature. AWS RDS allows many different relational database management systems including PostgreSQL that we are using. When creating and RDS instance we can choose to run it inside a VPC, Virtual Private Cloud, which we definitely want to do. This will help us make the database better protected against all kinds of attacks as it will not be accessible outside of the VPC.

3.4.2 Scrappers internet access

The scrappers are going to be run as AWS Lambda functions. As the scrappers update the database with fresh data they need to have an access to it. Given the fact that our database runs inside a VPC the scrappers also have to run inside it. Given the lambda functions can have a VPC assigned it is no problem to run them inside the same VPC as the database. Scrappers also need to have an internet access which resources running inside a VPC do not have by default. In order to allow the scrappers to access the internet we need to route the traffic from the private sub-nets that they run in into a NAT Gateway and from the NAT Gateway to an Internet Gateway. This is the only way lambdas can be given internet access and it presents a slight issue for us. Even though this solution works perfectly it is quite costly as running a NAT Gateway costs roughly 60 dollars per month. As this is far out of our budget we had to come up with the following solution. Given the fact the scrappers are run as timed events we know when they will be running. Given that they are the only resource which require a NAT Gateway it would be sufficient to have the gateway deployed only at the times scrappers are run. At first we hoped that the gateway could be created once and then only started and stopped as needed the same way it can be done with virtual servers. That is unfortunately not the case as AWS doesn't allow such actions with NAT Gateways. After doing some research we arrived at the following article [41] which explains how to set up and tear down NAT Gateways, along with other resources, using lambda functions and boto3 package. Using parts of the code displayed in the article we were able to construct two lambdas that do what we need, set up NAT Gateway and other required resources, set the correct routes in the route tables and roll back all these changes when needed. We also added timed events which trigger the set up/tear down lambda fifteen minutes before/after each of the events that trigger scrapper lambdas. This way we are able to run the NAT Gateway only when it is really needed. As AWS bills its resources on an hourly basis we are able to reduce the costs dramatically. If we estimate that the NAT Gateway will now be running only 3 hours a week instead of 168 hours, we are reducing the costs by more than 98 percent resulting in a monthly cost of roughly 1 dollar for the NAT Gateway.

3.4.3 API IP Address

A very important thing that has to be ensured is that the API is available on the same IP address even after actions such as server restart or even deletion. If the IP was to change the application would lose the access to it and we would have to update it to make it work again. This can be ensured by not using the default IP address given to the virtual server but rather by attaching an Elastic IP to it. An Elastic IP is a static IP address provided by AWS which can be attached to resources like virtual servers. The Elastic IP remains available until we manually delete it.

3.4.4 API Deployment

When we push code to the master branch of the repository containing the API a webhook is called which triggers AWS CodePipeline. This pipeline first tests the code, compiles it and then uses AWS CodeDeploy to deploy it to our virtual server. When deploying the code to the server a shell script is used which fetches the database credentials using aws-cli tool from AWS Parameter Store, creates environment variables from them, checks if the Redis server is running using redis-cli's ping command and if it is not starts it and only then it starts the API.

3.4.5 Scrappers Deployment

In order to deploy the scrappers and the lambda functions for NAT Gateway set-up/tear-down we use Serverless[42] which is a JavaScript framework. Serverless requires a Node.JS runtime and is rather easy to use. It only requires one .yaml file in which we define the individual lambda functions, their AWS specific attributes such as Security Groups or VPCs and other information such as timed events that tell the scrappers when to run. It also requires one GitHub job which starts a server with Node.JS environment and tells the Serverless framework to deploy the functions.[43]

Chapter 4
Testing

As testing is an important part of software development all parts of our application are properly tested in order to make them reliable. It also makes the application easier to modify in the future as we can always check if all its features are still working after doing some changes to it. There are multiple types of tests that were used to test the application. These will be described below alongside with where they were used in our application.

4.1 Types of tests

4.1.1 Unit tests

Unit testing is used to test the smallest pieces of code that can be isolated, in other words, units. In our case these are mostly functions or methods. The purpose of unit tests is to validate if those units perform the way they are meant to. In our application we use unit tests in both Scrappers and API.

4.1.2 Integration tests

Integration tests are used to inspect and verify that all the different units of code are combined and communicate correctly. Apart from unit tests, integration tests therefore use real dependencies and not mock objects.

4.1.3 User tests

User tests are usually performed on a web or mobile applications in order to verify their functionality. These tests, as the names suggests, are not performed automatically by machine but rather by real users who test the application by putting it in use or going through specified test scenarios. These tests are meant to simulate real world usage of the applications.

4.1.4 User tests

4.2 API

In order to implement the tests we used multiple frameworks and libraries the first one being JUnit. JUnit is a Java framework used for unit testing which provides us with many useful features such as method annotations that allow us to initialize variables before each test and many more. Another library that we are using is Mockito. Mockito is used to create mock objects and verify if and how they were called. It is especially important when creating unit tests as they would not be isolated from the rest of the code without it.

There is also a special series of integration tests which is focused on the functionality of the Redis cache. These tests focus on repeated calls to the methods in the controller layer as these are the methods whose results are cached and check if the calls retrieved only one result from database and the rest from the Redis cache. These test along with others require a functioning Redis server which can be quite tricky when running the test inside, for example, a GitHub job. In order to make running the tests simpler we use a dependency called *embedded-redis* which provides us with an embedded Redis server so we do not have to run it separately. As these tests actually use the Redis server, albeit an embedded one, the data from the method calls actually get cached inside it. This caused some issues as some of the tests were affected by the data cached during previous tests. In order to fix this issue we used Jedis, which is a library that allows us to communicate with Redis server, to flush all the data from Redis after each test.

Another library which is used is JaCoCo. JaCoCo stands for Java Code Coverage which quite perfectly sums up what it does, provides us with code coverage that our tests offer. In the API the unit tests cover majority of code except for the code that was automatically generated, mostly getters and setters, where coverage is not very important. JaCoCo also provides us with a website that displays the code coverage in an easily readable form.

4.3 Scrappers

In order to implement unit tests in scrappers we used multiple modules. The first module used is called unittest. It provides and easy way to creating unit tests and is also used very similarly to JUnit. Another package that we used is called responses. This package can catch the requests going out to the internet and provide mock responses to them. This is especially useful when testing software such as scrappers as we do not want to deal with issues such as website or network unavailability. The last package that we use is called freezegun. As some of the methods that we test work with current date and obtain it from datetime package whose methods we cannot mock, freeze time allows us to change the environment date when running the tests to the one we desire. In order to makes sure our code is properly covered by the tests a Python module called coverage was used. Similarly to JaCoCo, this module also provides us with a website that lets us easily display the coverage of our code.

4.4 Mobile application

Our mobile applications was tested with user tests. It was given to a group of people who were all part of the tennis community and participate in competitions organised by Czech Tennis Association on a frequent basis. Such people are part of our target group and therefore are the perfect individuals for performing the tests. Another reason for choosing the specified group of people is that they have a great insight into how to the competitions work as well as what the displayed data mean. This means that if there are any discrepancies in our data there is a good chance that users would be able to notice. As the application consists only of multiple screens the users were not given any scenarios which they should go through but were rather asked to but the application into a real world use and view tennis related data through it when they need. During the user testing multiple bugs with low severity were found, such as typos in titles, incorrectly sorted team matches in list views or inverted scores in specific cases. All these issues were addressed and fixed right away.

Chapter 5 Conclusion

The task of this thesis was to design and implement an application which is able to scrap data from the website of Czech Tennis Association[1], create statistics using said data and provide both the data and statistics to its users through a mobile application. Another task was to ensure the applications availability by deploying the back end to cloud. Both of these tasks were successfully completed.

In the chapter number 1 devoted to Analysis the existing solutions were analysed. It was concluded that apart from the scrapped website there are no other sources displaying data related to tennis competitions in Czech Republic. Some background on the scrapped website and tennis competitions organised by Czech Tennis Association was provided in order to help the readers of this thesis fully understand its topic and the problems addressed. Also the functional requirements and non-functional requirements were defined alongside with use cases which were mapped to respective functional requirements.

Chapter number 2 was focused on both the high-level architecture of the application as well as architectures of its parts and database schema. It was decided that the best way to split the application into parts is by using Client-Server architecture enriched by the scrappers used for extracting data from the website of Czech Tennis Association[1]. In the selected architecture the mobile application played the role of client and a REST API with database and Redis server acted as the server side. Apart from the architecture the technologies which were used for implementation of the application were selected in this chapter. This was especially important as there were multiple options available to choose from for most of the parts of the application.

Chapter number 3 discussed the implementation of the whole application which was based on the requirements stated by the Analysis [1]. The technologies chosen in Design chapter [2] were used to implement the application. For all the parts of the application we have mostly discussed which problems had to be dealt with during the implementation and how we managed to do so. The deployment to the cloud and some issues that were encountered while doing so are also addressed in this chapter. As a result of this chapter a fully functioning application, satisfying all the constraint from Analysis [1] and Design [2] chapters, was implemented.

In the fourth chapter [4] we described the different types of tests that were used to test the different parts of the application. After that we described how each of the parts was tested in greater details and what packages/tools/frameworks/libraries were used to do so.

To conclude we would love to say that working on this thesis was of great benefit to us as we were exposed to many new technologies whilst working on it. We believe that the knowledge gained will be useful to us not only throughout our studies but also when working on our projects and in the real world in general.

5.1 Visions for the future

As the topic of this thesis was chosen mainly due to our personal interest in it we plan to continue on development of the application eventually progressing into production. Before doing so, we also need to get an agreement from Czech Tennis Association which will allow us to use the scrapped the data in production. In order to obtain this agreement the application is going to be presented and potentially pitched to the Czech Tennis Association as we believe it could help us with obtaining the agreement and potentially could earn us some financial support from Czech Tennis Association.

In case we are successful at obtaining the agreement from Czech Tennis Association we would also have to implement and Android application so we could give access to the data to majority of mobile users. In the next iterations we would also like to implement a web application so the data could be accessed from computers as with it we would cover more or less all the potential users.

Bibliography

1. CZECH TENNIS ASSOCIATION. *Cztenis.cz* [online] [visited on 2022-01-05]. Available from: `https://cztenis.cz/`.
2. CZECH TENNIS ASSOCIATION. *Czech Tennis Association - Soutěžní řád* [online] [visited on 2022-01-07]. Available from: `https://cztenis.cz/docs/soutezni_rad.pdf`.
3. MATHEWS, Martin. *Functional Vs. Non Functional Requirements: Differences* [online]. 2022-02 [visited on 2022-02-10]. Available from: `https://www.guru99.com/functional-vs-non-functional-requirements.html`.
4. SINGLA, Chitra. *Functional vs Non Functional Requirements* [online]. 2020-05 [visited on 2022-02-10]. Available from: `https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/`.
5. ATP. *Roger Federer* [online] [visited on 2022-02-10]. Available from: `https://www.atptour.com/en/players/roger-federer/f324/overview`.
6. BRUSH, Kate. *use case* [online]. 2020-02 [visited on 2022-01-20]. Available from: `https://www.techtarget.com/searchsoftwarequality/definition/use-case`.
7. IBM CLOUD EDUCATION. *Three-Tier Architecture* [online]. 2020-10 [visited on 2022-04-23]. Available from: `https://www.ibm.com/cloud/learn/three-tier-architecture`.
8. PETERSON, Richard. *Database Architecture in DBMS: 1-Tier, 2-Tier and 3-Tier* [online]. 2022-04 [visited on 2022-04-23]. Available from: `https://www.guru99.com/dbms-architecture.html#4`.
9. AWS. *AWS Calculator* [online] [visited on 2022-05-07]. Available from: `https://calculator.aws`.
10. POSTGRESQL. *About PostgreSQL* [online] [visited on 2022-03-03]. Available from: `https://www.postgresql.org/about/`.
11. STATISTA. *Ranking of the most popular relational database management systems worldwide, as of January 2022* [online] [visited on 2022-03-03]. Available from: `https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems/`.
12. BAROT, Saurabh. *Flutter Vs Swift – Comparison of IOS App Development Tool in 2022* [online]. 2021-04 [visited on 2022-01-28]. Available from: `https://aglowiditsolutions.com/blog/flutter-vs-swift/`.
13. ELENA. *Swift vs. Objective-C: What language to Choose in 2021?* [Online]. 2022-02 [visited on 2022-02-02]. Available from: `https://gbksoft.com/blog/swift-vs-objective-c/`.

14. GILL, Bobby. *SwiftUI vs UIKit in 2022, Which Framework Should Your App Use?* [Online]. 2022-01 [visited on 2022-02-02]. Available from: `https://www.bluelabellabs.com/blog/swiftui-vs-uikit/`.
15. TARUN. *Top Programming Languages For Job Data Scraping* [online]. 2020-08 [visited on 2022-03-05]. Available from: `https://www.jobspikr.com/blog/top-programming-languages-for-job-data-scraping`.
16. OZSAHAN, Hatice. *12 Best Web Scraping Tools in 2022 to Extract Online Data* [online] [visited on 2022-03-05]. Available from: `https://popupsmart.com/blog/web-scraping-tools`.
17. AWS. *AWS Lambda FAQs* [online] [visited on 2022-03-05]. Available from: `https://aws.amazon.com/lambda/faqs/`.
18. MICROSOFT. *Supported languages in Azure Functions* [online]. 2021-10 [visited on 2022-03-05]. Available from: `https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages`.
19. SETHI, Anmol Singh. *Comparison of 10 Programming Languages.* [Online]. 2020-05 [visited on 2022-03-05]. Available from: `https://reubenrochesingh.medium.com/comparison-of-10-programming-languages-f43b0ac337a4`.
20. ÖZCAN, Bengüsu. *Best Web Scraping Programming Languages in 2022 with Stats* [online]. 2022-02 [visited on 2022-03-05]. Available from: `https://research.aimultiple.com/web-scraping-programming-languages/`.
21. ARSALAN. *Top Programming Languages For Job Data Scraping* [online]. 2021-06 [visited on 2022-03-05]. Available from: `https://it-s.com/5-best-programming-languages-for-web-scraping/`.
22. RODRÍGUEZ, Ander. *Web Scraping with Javascript and Node.js* [online]. 2021-09 [visited on 2022-03-05]. Available from: `https://www.zenrows.com/blog/web-scraping-with-javascript-and-nodejs#scraping-the-basics`.
23. STATISTICS AND DATA. *Most Popular Backend Frameworks – 2012/2022* [online]. 2022 [visited on 2022-03-10]. Available from: `https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2022/`.
24. PATEL, Jeel. *10 Popular Web Frameworks for Web App Development in 2022* [online]. 2021-11 [visited on 2022-03-10]. Available from: `https://www.monocubed.com/blog/most-popular-web-frameworks/`.
25. PATEL, Jeel. *Why to Use Django Web Framework for Web Development Projects?* [Online]. 2021-11 [visited on 2022-03-10]. Available from: `https://www.monocubed.com/blog/django-web-framework/`.
26. TECHVIDVAN. *Pros and Cons of Django as Web Framework* [online] [visited on 2022-03-10]. Available from: `https://techvidvan.com/tutorials/pros-and-cons-of-django/`.
27. OPENJS. *News from 2022* [online] [visited on 2022-03-12]. Available from: `https://nodejs.org/en/blog/`.
28. SHAH, Krunal. *Pros and Cons of Node.js Web App Development* [online]. 2021-07 [visited on 2022-03-10]. Available from: `https://www.thirdrocktechkno.com/blog/pros-and-cons-of-node-js-web-app-development/`.
29. HIRANI, Vipul. *The Positive and Negative Aspects of Node.js Web App Development* [online]. 2021-06 [visited on 2022-03-12]. Available from: `https://www.mindinventory.com/blog/pros-and-cons-of-node-js-web-app-development/`.

30. TONDON, Sophia. *The Pros and Cons of Node.js Web App Development: A Detailed Look* [online]. 2022-02 [visited on 2022-03-12]. Available from: `https://javascript.plainenglish.io/the-pros-and-cons-of-node-js-web-app-development-a-detailed-look-c91a22f013c`.
31. KUSHNIR, Anastasia. *Pros and Cons of Using Spring Boot* [online]. 2021-07 [visited on 2022-03-20]. Available from: `https://bambooagile.eu/insights/pros-and-cons-of-using-spring-boot/`.
32. JAVATPOINT. *Java Spring Pros and Cons* [online] [visited on 2022-03-20]. Available from: `https://www.javatpoint.com/java-spring-pros-and-cons`.
33. SCAND. *Pros and Cons of Using Spring Boot* [online]. 2020-06 [visited on 2022-03-20]. Available from: `https://scand.com/company/blog/pros-and-cons-of-using-spring-boot/`.
34. JAVATPOINT. *Spring Boot Architecture* [online] [visited on 2022-03-30]. Available from: `https://www.javatpoint.com/spring-boot-architecture`.
35. PAL, Ankur; KUMAR, Avtar. *Spring Boot – Architecture* [online]. 2022-03 [visited on 2022-03-30]. Available from: `https://www.geeksforgeeks.org/spring-boot-architecture/`.
36. DIGNAN, Larry. *Top cloud providers: AWS, Microsoft Azure, and Google Cloud, hybrid, SaaS players* [online]. 2021-12 [visited on 2022-04-10]. Available from: `https://www.zdnet.com/article/the-top-cloud-providers-of-2021-aws-microsoft-azure-google-cloud-hybrid-saas/`.
37. SANDOVAL, Kristopher. *10+ Tools To Mock HTTP Requests* [online]. 2020-01 [visited on 2022-03-13]. Available from: `https://nordicapis.com/10-tools-to-mock-http-requests/`.
38. WUTTISASIWAT, Nitipat. *Macbook Personal Hotspot to your iPhone, iPad or iPod via iPhone USB* [online]. 2017-12 [visited on 2022-03-18]. Available from: `https://medium.com/@kennwuttisasiwat/macbook-personal-hotspot-to-your-iphone-ipad-or-ipod-via-iphone-usb-979e492a1314`.
39. APPLE. *Unwanted frame animations in SwiftUI 2* [online]. 2021 [visited on 2022-03-13]. Available from: `https://developer.apple.com/forums/thread/670836`.
40. OGANESYAN, Andrey; STUKALOV, Aleksey. *Lombok and JPA: What Could Go Wrong?* [Online]. 2021-05 [visited on 2022-03-13]. Available from: `https://dzone.com/articles/lombok-and-jpa-what-may-go-wrong`.
41. DUMBRE, Abhinav. *Working with VPC in Python using Boto3* [online]. 2021-11 [visited on 2022-04-20]. Available from: `https://hands-on.cloud/working-with-vpc-in-python-using-boto3/`.
42. SERVERLESS. *Serverless Framework Documentation* [online] [visited on 2022-04-20]. Available from: `https://www.serverless.com/framework/docs`.
43. AWS COMMUNITY BUILDERS. *Setup CI/CD for your AWS Lambda with Serverless Framework and GitHub Actions* [online]. 2022-02 [visited on 2022-04-20]. Available from: `https://dev.to/aws-builders/setup-cicd-for-your-aws-lambda-with-serverless-framework-and-github-actions-4f12`.

Obsah přiloženého média

```
src
├── thesis.pdf ............ thesis in the form of a .PDF file
└── thesis.zip ............ LaTex source code of the thesis
```