



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

**PLATFORM FOR ORGANIZING
AMATEUR COLLECTIVE SPORTS**

PLATFORMA PRO ORGANIZOVÁNÍ AMATÉRSKÝCH KOLEKTIVNÍCH SPORTŮ

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR

AUTOR PRÁCE

Mgr. Bc. ADAM LÁNÍČEK

SUPERVISOR

VEDOUCÍ PRÁCE

prof. Ing. ADAM HEROUT, Ph.D.

BRNO 2022

Master's Thesis Specification



Student: **Lániček Adam, Mgr.**
Programme: Information Technology and Artificial Intelligence
Specialization: Machine Learning
Title: **Platform for Organizing Amateur Collective Sports**
Category: User Interfaces
Assignment:

1. Get acquainted with the matters of design and development of web and mobile applications.
2. Analyze the field of organizing amateur collective sports; focus on ice hockey.
3. Propose and design the targeted application - analyze use cases, key elements of the user interface, the looks, usability, data management, etc.
4. Prototype elements of the application, test them with users and iteratively improve them.
5. Integrate the individual elements into the application, test it with users and iteratively improve it.
6. Assess the achieved results and propose possibilities for future work; create a poster and a short video for presenting the project.

Recommended literature:

- Tidwell et al.: Designing Interfaces: Patterns for Effective Interaction Design, O'Reilly, 2020
- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN: 978-0321965516
- Steve Krug: Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability, ISBN: 978-0321657299
- Joel Marsh: UX for Beginners: A Crash Course in 100 Short Lessons, O'Reilly 2016

Requirements for the semestral defence:

- Items 1 and 2, considerable progress on items 3 and 4.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Herout Adam, prof. Ing., Ph.D.**
Head of Department: Černocký Jan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: May 16, 2022

Abstract

The aim of this thesis was to design and implement a Platform for facilitating organization of amateur team sports, with the main focus on ice hockey with its arguably largest inefficiencies. As opposed to the traditional systems, focusing on rigid groups management, the Platform introduces an on-demand element by providing a transparent database of games, effectively facilitating a market for ad hoc amateur ice hockey. Additionally, it implements a follow mechanism to enable the athletes to play with their favourite teammates. The Platform is composed of a REST API server written in Python/Flask framework and a web application front end implemented in React TypeScript. The design mock-ups that are appended to this work were created using the Figma tool. The goals of designing the web application and implementing its core features, supported by the corresponding API endpoints, were fulfilled. Numerous features, such as skill rating mechanism completion and an e-mail subscription service for reporting the upcoming games, were left as task for future development.

Abstrakt

Cílem této práce bylo navrhnout a implementovat Platformu pro organizaci amatérského kolektivního sportu s důrazem na lední hokej, kde jsou náklady neefektivní organizace pravděpodobně nejvyšší. Narozdíl od tradičních systémů, zaměřených na organizaci rigidních skupin, přichází tato platforma s *on demand* prvkem v podobě transparentní databáze utkání, která efektivně vzato tvoří trh účastníků a utkání ad hoc amatérského ledního hokeje. Dále implementuje mechanismus sledování, který zdůrazňuje utkání, kterých se účastní oblíbení spoluhráči. Platforma se skládá z REST API serveru implementovaného v rámci Flask/Python a webového front endu naprogramovaného s pomocí knihovny React TypeScript. Designové návrhy přiložené k této práci byly vytvořeny v nástroji Figma. Cíle práce spočívající v návrhu platformy a implementace jejich klíčových částí, adekvátně podpořených ze strany serverového API, byly splněny. Další funkcionality, jako například dokončení mechanismu udělování pozápasového hodnocení mezi hráči a e-mailová služba pro hlášení vybraných nadcházejících utkání, budou předmětem vývoje v budoucnu.

Keywords

amateur sports management, team sports management, ice hockey management, ice hockey management web application, sport organization web application

Klíčová slova

webová aplikace pro amatérské sportovce, správa amatérského sportu, řízení a organizace ledního hokeje

Reference

LÁNÍČEK, Adam. *Platform for Organizing Amateur Collective Sports*. Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Adam Herout, Ph.D.

Rozšířený abstrakt

Bude-li tázán kdokoli, kdo se v minulosti pravidelně věnoval týmovým sportům, na své domy, odpověď bude pravděpodobně vždy podobná — ve správném kontextu skvělý zážitek, zbytečně vynaložené úsilí v opačném případě. V mládežnickém kolektivním sportu jsou sportovci nuceni zvyknout si, že jsou schopni ovlivnit jen omezené množství faktorů. To se však obvykle mění v situaci, kdy sportovec opouští mládežnické kategorie, startuje svoji mimosportovní pracovní kariéru a na svůj sport nechce úplně zanevřít. Najednou si uvědomuje, že jeho volný čas je s každým dalším rokem omezenější, svého herního času si tudíž váží daleko více a chce jej využít na maximum.

Lze namítnout, že výše uvedený vývoj se týká všech kolektivních sportů, autor si však dovoluje zavést předpoklad, že podstata ledního hokeje tuto výzvu činí ještě náročnější. Žádný jiný sport se neodehrává na kluzkém povrchu, vyžadujícím od hráčů ovládnutí pro člověka nepřírodných pohybových vzorců a rozhodování ve zlomcích sekundy. S tolika herními proměnnými bývá velice náročné najít jen několik hráčů, kteří zapadají do hráčova profilu, natožpak dvacet a více jako v případě ledního hokeje. Ostatní hráči mají navíc také své kariéry a související časové preference, což situaci dále komplikuje. Dále je třeba uvědomit si, že lední hokej je ze své podstaty finančně velmi náročný sport, generující nezanedbatelné náklady obětované příležitosti, a to i pokud finanční náročnost posuzujeme jen z pohledu cen pronájmů hokejové plochy.

Výše uvedené faktory přispívají ke všudypřítomným neefektivitám v rámci ad hoc organizovaného amatérského hokeje. Hokejisté hrají v nepraktických termínech, s jinak herně schopnými hráči a v nejhorsím případě i bez brankářů, a to vše za několik stokorun Kč za utkání. Ačkoli platformy podporující organizaci týmových sportů existují, jejich hlavním posláním není podporovat střet nabídky s poptávkou hráčů amatérského hokeje, jedná se spíše o portály zajišťující organizaci v již existujících, často více či méně uzavřených, skupinách. Proto bylo cílem této práce vytvořit platformu pro řešení úskalí spojených s účastníky amatérských kolektivních sportů, a to se zaměřením na lední hokej, kde je z výše uvedených důvodů předpokládáno, že jsou kumulativní neefektivita nejvyšší.

Pro dosažení tohoto hlavního cíle bylo v rámci naplánováno několik cílů dílčích. Zaprvé, bylo nutno provést důkladnou analýzu klíčových aspektů organizace ledního hokeje pro vhodnou definici případů užití Platformy, která se zaměří na největší úskalí. Ta byla identifikována jednak jako nedostatek dostupných amatérských brankářů, způsobený jejich neefektivním, nebo spíše neexistujícím, trhem, a dále velká náročnost udržet počty hráčů konzistentní v podmínkách vysokých fixních nákladů spojených s pronájmem ledové plochy.

Následovala definice uživatelského rozhraní, vytvářející návrhy pro obě cílové platformy – webové rozhraní i mobilní aplikaci. Po úvodním náročném seznamování s nástrojem Figma na tvorbu uživatelských rozhraní, jeho schopnost strukturovat komponenty následně velmi podpořila několik návrhových iterací, kdy byly návrhy opakovaně vylepšovány na základě zpětné vazby od potenciálních uživatelů.

Vzhledem k důrazu na webové rozhraní spolu s plány na pokračující implementaci mobilní aplikace v budoucnu, jako vhodná back end technologie byl zvolen REST API server, implementován v programovacím jazyce Python. Principy této technologie byly důkladně analyzovány, stejně jako základní stavební kameny JavaScript/TypeScript knihovny React, využívané pro implementaci webového rozhraní, a její varianty React Native pro implementaci víceplatformních mobilních aplikací.

Vzhledem ke komplexitě webového rozhraní Platformy bylo úspěšně naplněným cílem vyvinutí jejich klíčových funkcionalit, náležitě podporovaných ze strany REST API serveru. Pro maximalizaci uživatelského prožitku byly v rámci studia asynchronní komunikace zk-

oumány a použity možnosti stránkování dat. Pro rychlejší vtažení uživatele do Platformy byla definice registračního procesu ovlivněna moderními principy odložené registrace. V neposlední řadě byly dále položeny základy pro další vývoj mobilní aplikace, jejíž implementace částečně započala v rámci předmětu Tvorba aplikací pro mobilní zařízení na FIT VUT. Přirozeným důsledkem bylo přijetí *monorepo* přístupu ke správě repozitáře, který poskytuje nutný základ pro využití mechanismu sdílení kódu, dostupného díky volbě knihoven React a React Native jakožto rámců pro implementaci aplikačního front endu.

Závěrem lze podotknout, že cíle této práce byly naplněny, přičemž další funkcionality a nápady ještě na implementaci čekají — nejdůležitějším z nich je v krátkém období implementace interaktivní části hodnocení hráčských schopností, konkrétně vyhodnocení výkonů hráčů po utkání. Dalším krokem je implementace stránky Uživatelského profilu, kde si bude uživatel schopen zobrazit svoje statistiky, což dále může zvýšit jeho angažovanost v Platformě. Dlouhodobým cílem je přidání mechanismu e-mailových hlášení o utkáních, kterých se účastní uživatelem sledovaní hráči.

Platform for Organizing Amateur Collective Sports

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of prof. Ing. Adam Herout, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Adam Láníček
May 17, 2022

Acknowledgements

Hereby I would like to thank my supervisor prof. Herout for pushing me to make things happen over the course of these two last semesters and at the same time for being lenient enough to let some innovative concepts mature before diving into the implementation phase. Additionally, I would like to express my sincere gratitude to my friend and a former classmate Ing. Šimon Pokorný for the creative critique provided and the endless sessions spent together brainstorming.

Contents

1	Introduction	2
2	Amateur Ice Hockey Matches Organization	4
2.1	Key Aspects of Organization	5
2.2	User stories of an amateur ice hockey player	5
2.3	Platform’s extended use case diagram	6
3	User interface creation	9
3.1	Personas	9
3.2	Responsive web application versus dedicated mobile application	10
4	User interface mock-ups	13
4.1	Mock-ups creation process	13
4.2	Mobile application mock-ups	15
4.3	Web application mock-ups	16
5	Technological and Operational Aspects of Web & Mobile Applications	24
5.1	REST API Server Network Communication	24
5.2	API Server Back End Technologies	28
5.3	Application Front End Technologies	33
5.4	Application Deployment Technologies	39
6	Amateur ice hockey organization platform	42
6.1	Analysis of existing services	42
6.2	Facilitating a market of goaltenders and referees	47
6.3	Lazy registration	48
6.4	Game attendance management principles	49
7	Implementation	52
7.1	REST API server	52
7.2	React web application front end	58
7.3	React Native mobile application front end	66
8	Conclusion	70
	Bibliography	71
A	Full web application mock-ups	74
B	Web application screen captures	81
C	Mobile application screen captures	86
D	Included storage medium contents	87

Chapter 1

Introduction

Asking a person who has ever participated in a team sport on a regular basis about his experience will perhaps always yield the same answer – terrific under the right circumstances, a waste of effort otherwise. When playing in youth teams, an athlete has to come to terms with the fact that there is only a few things he can influence in this regard – a coach is a given, teammates are more or less constant and games are played against the same or very similar opponents over the course of multiple seasons. Put simply, either the odds are in his favour or not. However, as he leaves his youth team to start his professional career outside sports with an ambition not to abandon his beloved sport completely, the tides tend to change. Suddenly, multiple constraints are gradually placed around his free time and he cherishes his amateur game time more with every year passing by. An increasingly scarce resource has to be exploited as much as possible.

Arguably, this development affects every existing team sport, though the author dares to assume that the nature of ice hockey renders this exploitation challenge especially difficult. No other team sport is played on a slippery surface, necessitates a control of unnatural movement patterns, stick handling and requires decisions in a collective context at a fraction of a second. With so many variables at hand, it can be a real challenge to find a few teammates that fit in athlete's profile, let alone twenty or more, as generally required. Additionally, other athletes have also their careers and time preference, complicating the situation even further. On top of that, ice hockey is an inherently expensive sport, even when taking into account only the ice rink renting costs, generating significant *opportunity costs*.

All of the factors named above contribute to the ubiquitous inefficiencies within the amateur ad hoc organized ice hockey. Athletes are playing at unfavourable time slots, with inadequately skilled teammates and without goaltenders for a few hundreds of CZK a game. Although there are platforms facilitating team sports management, their core purpose is not to match the athletes' demand with the supply, it is rather to take an existing set of athletes and help with the organization within that specific closed group. Therefore, the aim of this work is to rise to the challenge of facilitating the attendance of amateur sport games, while focusing on the ice hockey where the cumulative inefficiencies are arguably the highest.

To lay the groundwork for the Platform's definition, the work will start with a thorough analysis of the main aspects of amateur ice hockey, followed by an identification of hockey player's needs and a proposal of said Platform's use cases. Having a clear idea of the Platform's mission, its *Personas*, representing the core user groups, will be defined and the selection process of the appropriate target platforms described. To conclude the user

interface creation process, the following chapter deals with the user interface mock-ups creation process using the Figma tool and presents the resulting web application and mobile client mock-ups.

The next chapter describes the technological and operational aspects of the planned Platform, starting with an analysis of the technological background required for implementing a REST API server as the application back end solution of choice. In the second half of the chapter, an in-depth tour into React JavaScript/TypeScript library targeted for web application front end development as well as its variant React Native, which leverages React's abstraction power to bring the technology to mobile devices via cross-platform mobile applications.

The penultimate chapter starts with the analysis of existing services serving similar markets to draw inspiration, and consequently highlights some of the key features of the newly implemented Platform. The actual implementation and the challenges faced are covered in the final chapter, focusing on the Platform's two main building blocks – REST API server and the React web application front end – with the mobile application to be left mainly as a task for future development.

Chapter 2

Amateur Ice Hockey Matches Organization

Ice hockey is among the most popular team sports in Czechia. Its popularity can be illustrated using Figure 2.1, displaying shares of registered ice hockey players within the population in the most notorious hockey countries worldwide. Given the country's population of 10,6 million people (Czech Statistical Office [38]), approximately 1,2 out of 100 inhabitants is registered with the Czech Ice Hockey Association¹. This proportion is, perhaps surprisingly, even greater than in Canada with its 345000 registered players and the population of 38.5 million [4], rendering it third with the value of 0.9%. For example, in the United States of America, where over three-quarters of National Hockey League teams reside², is this proportion approximately *ten times* lower than in Czechia.

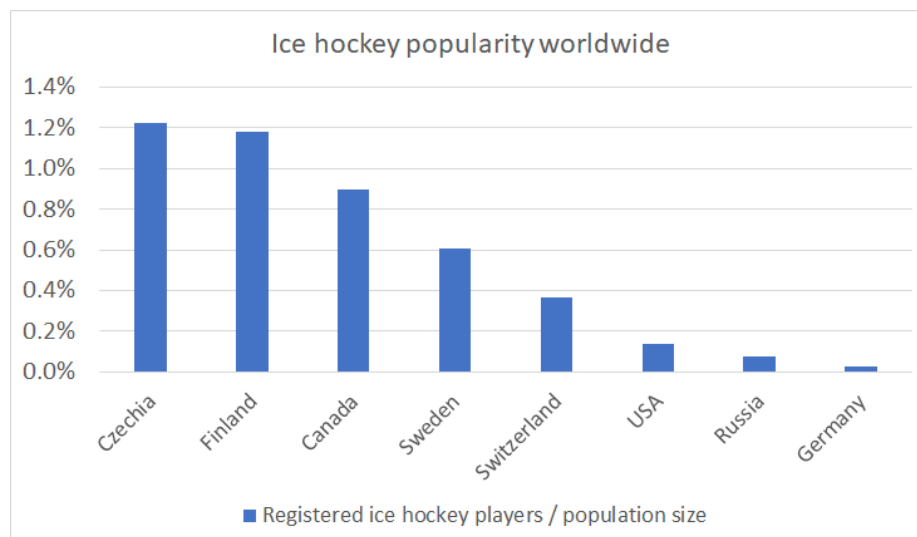


Figure 2.1: Share of registered ice hockey players within the population (2021)
Source: Own calculations based on data from [4], [12] and [38]

Not surprisingly, this popularity in Czechia is reflected on amateur level as well. As of January 2022, an own survey of the existing amateur leagues [2] has shown that there are at

¹Available at <http://www.czehockey.cz/>

²Currently 25 out of 32. Source: <https://www.nhl.com/info/teams> [01-04-2021]

least 110 amateur hockey leagues, being played on 196 indoor ice rinks [11] scattered across the country. Assuming 10 participating teams per league and 20 players per team, the size of the amateur hockey crowd can be estimated at around 22000 people. Additionally, the majority of these players does not limit their ice hockey participation to organized leagues and plays other games which take place more or less regularly. This majority of players is also the main target group of the application developed within this work. Therefore, the rest of the chapter will be dedicated to an analysis of their needs as well as to a discussion of the key aspects of the game to be optimized to maximize the game experience.

2.1 Key Aspects of Organization

Ice hockey is played by two opposing teams. In standard circumstances, 5 players from each team, along with a goaltender guarding the net, are present on the ice at any given moment. Given the inherent high intensity nature of the game, it is desirable to have at least the same amount of players on the bench, ready to be substituted, theoretically keeping the work/rest ratio at 1:1. Driving this ratio higher in favour of the work inevitably leads to a decrease of game experience of all participants. As a result, one specific game slot (typically reserved for 60 or 75 min) requires at least 20 players plus two goaltenders that are, preferably, similarly skilled. Moreover, rent prices of an indoor ice rink range from 3800 to 5200 CZK an hour, depending on the time of the day, providing an additional incentive to maximize the amount of players that share its cost.

Requiring 20 similarly skilled amateur players to meet more or less regularly at a specific time and place is a challenge. Therefore, inevitably, a significant subset of the player lineup is constituted from ad hoc players, showing up irregularly multiple times over the course of the year. Irregularity implies the attendance unpredictability, which results in a higher risk of an unfavorable financial burden to all players who have to pay *more* while enjoying *less*.

Another crucial aspect of the game are the goaltenders. Given their highly specialized and rather costly equipment, as well as a specific skillset, it might not be easy to find them for your specific game, let alone find replacements in case of an attendance cancellation. To compensate amateur goaltenders for their higher upfront equipment costs, it is a common practice to let them play free of charge. Even in this setting, many amateur games are nowadays played with one or both goaltenders missing, decreasing the game experience so drastically that players are willing to cover additional costs of having a paid goaltender, ranging from 200 to 500 CZK per a goaltender and a game.

Teams are facing similar organizational issues in every amateur team sport. However, none of them is inherently as expensive as ice hockey. Whether it is insufficient amount of players or missing goaltenders, the inefficiencies expressed financially are really significant, providing a lot of space for optimization.

2.2 User stories of an amateur ice hockey player

What an amateur ice hockey player and goaltender strives for can be identified by taking another perspective on the organizational aspects described in the previous section. Therefore, the user stories can be defined as follows:

As an amateur ice hockey player,

1. I want to be able to play hockey *wherever* and *whenever* it fits in my tight schedule. As life sometimes goes into the way, I want to be able to decide to participate as spontaneously as possible.
2. I want to play with similarly skilled players in order to avoid disappointment in either direction.
3. I want to play with the people I know.
4. I want to play in games where at least 20 players are present.
5. I want to participate only in games where both goaltenders are present.

As an amateur ice hockey goaltender, I want to participate in games where all player's stories are met. In addition to that, crucially,

1. I want to play/prefer to play in games where I am *financially* compensated the most.

All of the user stories mentioned above present the core issues and challenges which the platform developed within this work aims to face and resolve.

2.3 Platform's extended use case diagram

Reflecting the needs of amateur hockey player, as discussed in previous section, the use cases of the target Platform are proposed in the extended use case diagram depicted in Figure 2.2. Rather than focusing exclusively on actors interacting with the system, as is the case in a standard use case diagram, the variant described hereinafter attempts to model the whole application logic to provide an overview of its workings. Extracting the relevant pieces of information from Figure 2.2 to emphasize the user perspective, available user's action can be inferred as described by the following text.

Platform's user is able to:

- **assign** himself/herself one or multiple participation roles in the user profile.
- **participate** in a game in roles restricted by the settings in user profile – theoretically as a player, goaltender or a referee.
- **organize** a game to offer places to attend to all of the platform users.
- **follow** another user in one of the modes determining the ability of the game-organizing followee to sign him/her up for a game without any further action from the added person. The follow relationship is either of
 - the **opt-in** type, to be able to be to see his/her attended or organized games on the Dashboard, or
 - the **opt-out** mode, to reap the benefits of the opt-in variant and allow the automatic sign up.
- **add** the followed athlete to a game, provided that he/she is followed in the *opt-out* mode.

- **evaluate** a game performance of other players to contribute to the rating creation mechanism.
- **create and maintain** a group of athletes who follow him/her in the *opt-out mode* to facilitate and simplify the new game creation process by adding a group of people to the game instead of searching for each and every athlete in the system with every game created.

To conclude the diagram interpretation, it should be noted that a game remunerates goaltenders and referees for their attendance in these crucial roles (namely in goaltender's case) and expects a payment from the attending players to cover the game costs. These relationships, however, do not describe the planned interactions within the system for the time being, as the proposal of payment processing interface implementation has been postponed.

Despite simplifying both organizer's and attendee's workflows by removing the inconvenience of dealing with cash money and providing an opportunity for the Platform to charge a small premium on every transaction to cover for the operational costs, major challenges and complications would arise from the simple fact that only a subset of the athletes attending the games are actively using the Platform.

It is reasonable to assume that over time, based on the fear of missing out on the rating and Platform's social features, a large majority of athletes would register in the Platform. But even then the question remains – In an environment of scarce player resources, especially but definitely not limited to late spring or summer, what leverage does the organizer have to make a player use the Platform for payment processing and thus pay the Platform's premium, however insignificant? The answer lies in providing an adequate value to the athlete. Is the rating system and the ability to choose their games *ad hoc* enough? If yes, implementing a payment interface would be a next logical step. This question will be answered only by a long-term existence of the Platform in the market.

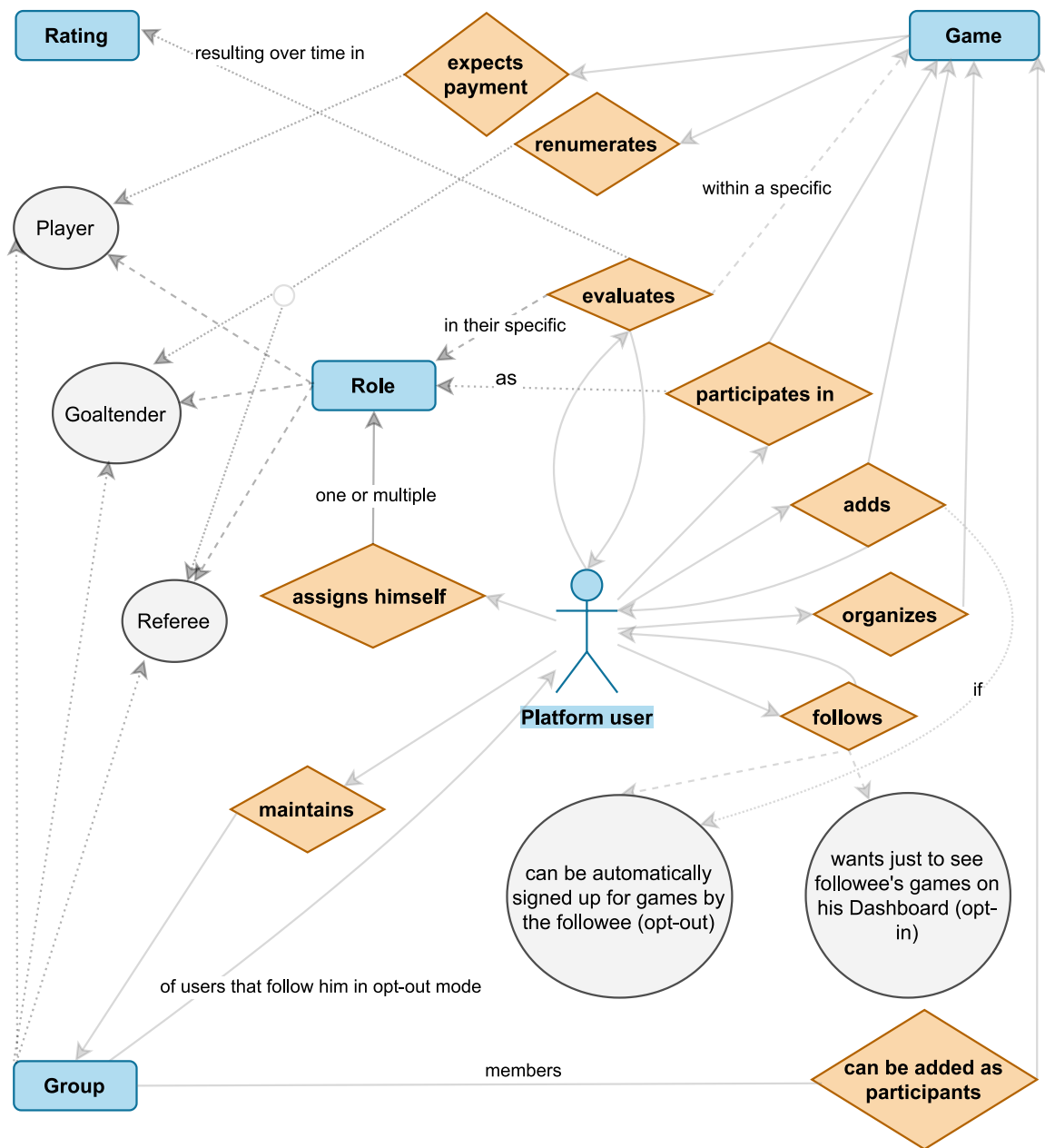


Figure 2.2: Platform's extended use case diagram. Depicts user's use cases in the target Platform as well as other relationships among the entities interacting within the system. Compared to a standard use case diagram, which focuses exclusively on users as actors interacting with the system, this diagram attempts to model the whole application logic to provide a high-level overview of its workings.

Chapter 3

User interface creation

The aim of this chapter is to apply the well-known user interface concepts to efficiently facilitate solving of issues and challenges faced by an amateur ice hockey player/organizer which were introduced in the previous chapter. The following content is structured as follows:

1. To begin with, Platform *Personas* representing typical Platform users are defined.
2. Consequently, the decision process towards the selection of an appropriate target platform for the application is analyzed.
3. And lastly, the chapter is concluded with a description of the process of user interface mock-ups creation and testing, as well as with the presentation of the actual mobile and web application mock-ups created for the Platform at hand.

3.1 Personas

When designing an application, it is crucial to have a clear understanding about who its end users are going to be. As Sauro [32] noted, instead of creating a product based on just abstract demographics, using *Personas* as a basis for the design phase leads to a better focus on real customer needs and goals.

A *Persona* is a fictional character representing the real needs of a larger group of users. Despite being a representative essentially aggregating opinions of a group, *Persona's* characteristics should be defined in a significant level of detail to clearly express major needs and expectations of the most important user groups. Since the goal of creating these fictional characters is, according to Sauro [32], capturing the needs of the most sizeable user clusters, rather than of each and every one, the number of *Personas* should be limited to three or four at maximum.

Analyzing a rough demographic structure of target users significantly narrows down the opinion space for the purposes of *Personas* definition. Therefore, let us introduce some reasonable demographic assumptions:

1. According to IIHF [16], the share of women in registered ice hockey in the Czech Republic is 3.92%. Although the share might be different for amateur ice hockey, for which there is no such statistics available, it is reasonable to assume that around 95% of Platform's users are going to be men.

2. The age range of users can be divided into the three groups, encompassing both former registered players and hockey enthusiasts without any prior structured ice hockey training:
 - (a) **18–27 years**, often referred to as *Generation Z*, the members of which have been living their whole life in the digital age and are accustomed to using smartphones for nearly every web task imaginable. According to data.ai [13], these people visit their favourite websites frequently and keep the visit duration low.
 - (b) **28–49 years**, representing *Millennials*, who divide their attention between mobile and desktop devices equally, do not consume their favourite websites’ content as often, but have a 25% longer attention span than a typical representative of a younger generation.
 - (c) **>49 years of age** generation, which emphasizes desktop activities and the representatives of which extend their visit length by additional 25% compared to *Millennials*.

The shares of the groups described above within the population of ice hockey amateur players in the Czech Republic can be roughly estimated to 25, 60 and 15%¹, respectively. Though the real shares may vary, it is reasonable to assume that three different *Personas* representing the vast majority of users can be defined as indicated in the Table 3.1.

Based on the group shares discussed above, Petr Novotný and his peers are arguably the most important group to focus on, followed by Bartoloměj Březina and Lukáš Terasový on the last place. Bearing in mind that Petr Novotný spends half of his technology consumption time on desktop, it seems that the Platform should definitely offer a web-based client, which offers the most value for the development effort overall when preferences of Bartoloměj and Lukáš are taken into account as well. The importance of a comfortable mobile access cannot be underestimated, though. While the mobile platform is strictly preferred by Bartoloměj and his peers, representing a quarter of all users, even Petr appreciates the possibility to sign up for a late evening ice hockey match during a busy working day. Similarly Lukáš, who generally limits the interaction with a smartphone to work-related phone calls and a few text messages to his kids, is more often than not away from the keyboard, requiring another modality for signing up for his favourite hockey match.

3.2 Responsive web application versus dedicated mobile application

The previous section established that the development of a web-based client for the Platform is reasonable because of the preferences of typical *Personas*. Additionally, it has been concluded that there should be a feasible modality to access the Platform using a mobile device. The selection process of the correct mobile modality is, for its complexity, worth further analysis and discussed in this section.

Generally speaking, there are two high-level approaches towards building a solution for mobile platforms, each of which comes with specific costs and benefits:

- a website built respecting Responsive web design (RWD) principles

¹Author’s estimate based on more than 10 years playing in or organizing 10 or more different groups in total

	Bartoloměj Březina	Petr Novotný	Lukáš Terasový
Age	22	36	52
Employment status	University student	Full-time worker	Self-employed in a small company
Family status	Single	Married with two kids	Married with two high-school kids
Playing time preference	Late morning/early evening	Early morning/late evening	Early morning/late afternoon
Ratio of desktop vs mobile usage	20:80	50:50	80:20
Attention span	8s	12s	16s

Table 3.1: *User Personas* representing the vast majority of Platform’s users. Bartoloměj Březina as a member of Generation Z, Petr Novotný as a *millennial* and Lukáš Terasový representing the oldest age group.

- dedicated native/cross-platform mobile application

Responsive web design (RWD)

Simply put, Responsive web design, as defined by the MDN Web Documentation [7], is a set of best practices used to create a layout that responds to the device being used to view the content. It was first introduced by Ethan Marcotte [26] who described the use of three techniques in combination:

1. **Fluid grids** with fluid-width columns that expand and shrink based on the screen size to fill the available space.
2. **Fluid images** ensuring that images placed within a fluid grid scale down smaller when the grid shrinks down in size but never grow larger when a grid grows, avoiding the pixelation that would occur otherwise.
3. **Media queries** as a feature of Cascading Style Sheets (CSS) that enabled changing the layout based on the screen size by querying the relevant features in the browser.

Nowadays, most modern and popular layout techniques available in CSS are inherently responsive. The Platform implemented within this work, for example, uses both the two most popular layouts - Grid and Flexbox as discussed in more detail in Section 7.2.4.

Utilizing Responsive web design as the only solution for mobile devices without implementing another dedicated one offers significant benefits, the most attractive one being having just one codebase to maintain. Additionally, the site contains the same content across varying platforms, providing a consistent experience.

Having said that, unless a mobile-first approach is taken, the website will never be fully optimized for mobile devices. On top of that, performance of a web application can never be a match for a dedicated mobile application. And lastly, mobile users are used to mobile-specific interfaces and may find interacting with the Platform’s web application

	Responsive website	Dedicated mobile application
Compatibility	Equally displayed in all browsers irrespective of the device	Requires development of at least one extra (cross-platform) application
Audience	All devices with internet connection	Smartphones and tablets
Regular usage convenience	Mediocre	Really good
Personalization	Mediocre – inherently focused on the service	Really good – aim is at the individual user

Table 3.2: Comparison matrix: Responsive web design as the only solution for mobile versus an extra dedicated mobile application.

cumbersome, which may eventually lead to a loss of active users and limit the amount of the up-to-date data available at the Platform at any given time. All of these factors limit the user experience of the interaction with the Platform in a mobile browser compared to a mobile application.

Dedicated mobile application

Dedicated mobile clients undoubtedly offer the best user experience. Optimized for the device they are running on, they are more responsive while interacting with a user via familiar user interface elements. They can even be accessed in the offline mode with a limited functionality. For example in Platform’s case, a user could launch the application to see to which games he is signed up for without requiring any connectivity. Even more importantly, contrary to a web browser variant, a mobile application can exploit all the features of a device, including push notifications that might significantly improve the user experience in Platform’s case. A user can be informed about the upcoming matches organized by his favourite organizers or reminded that his match starts in a few hours. On the flip side, a mobile application requires the user to download it. Additionally, maintaining two separate clients can significantly raise Platform’s operational costs.

Having introduced all the benefits a dedicated mobile client has to offer, it should be noted that there are ways to reduce the costs to increase the incentives of choosing this product strategy even more. For example, as demonstrated in Section 7.2.2, when choosing the React JavaScript/TypeScript framework for a web front end implementation and React Native for a mobile client, it is possible to leverage the reusability of the whole business logic of an application. This implies, in essence, that only the target platform-specific rendering logic (see Section 5.3.2 for details) is kept separate for both the target platforms while its inputs originate from the same codebase.

Chapter 4

User interface mock-ups

Given the level of interactivity of the proposed Platform and the possibility to exploit sharing mechanisms of React TypeScript code described in the previous chapter, it has been decided to pursue the more challenging path of implementing a standalone mobile client along with a standard web application. This chapter describes the process of creating user interface mock-ups for both platforms. The following content is structured into the three sections:

1. Description of the process of user interface mock-ups creation using the **Figma** tool.
2. Short presentation of the key **mobile application's** mock-ups.
3. Presentation of the key **web application** pages and components mock-ups.

4.1 Mock-ups creation process

For the purpose of mock-ups creation, Figma¹ web-based graphics editing and user interface design tool has been used. The basic primitives in Figma are *layers* which are stacked on top of others by using simple user interface gestures. These layer groups can then be treated as one entity, i.e. moved or edited together or even abstracted away as a Component, making this layer group reusable across the whole project.

After getting familiar with these concepts, working with Figma is really productive and efficient. This proved to be really useful over the whole mockup design phase – almost every screen or a page underwent a complete overhaul after informal discussions with potential users. To illustrate the Figma's layering logic, Figure 4.1 depicts an instance of the „Skill puck“ component and the corresponding Figma user interface drill down of the layers it is composed of. Notably, its Figure 4.10a demonstrates how a simple feature of hiding/showing layers can help when creating a component with variable properties – the first three black pucks are shown while the first three grey pucks are hidden and the opposite applies for the three remaining pucks.

Being the core of the application, the first screen/component to be designed was the Games overview. One of the important features of the Platform is incorporating attendees' skill levels into the game management process. This presented a first user interface design challenge to be faced. In the first iteration, 6 black or grey stars depicted in Figure 4.2 were used as a measure of the skill expected from a player attending a given game. After

¹Accessible at <https://www.figma.com/>

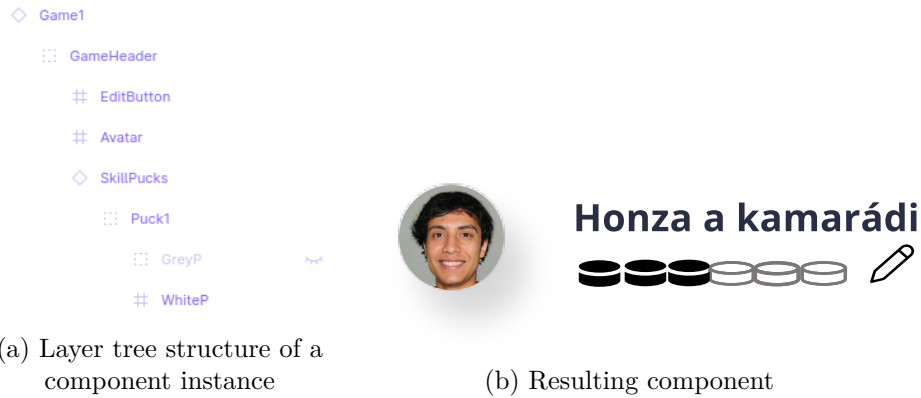


Figure 4.1: Figma and the tree structure of its components. Skill pucks component taking advantage of the layering by hiding/showing only the relevant layers.



Figure 4.2: Stars as the first dismissed prototype of expected skill depiction. Rather than conveying expected skill, according to the potential users, it gives an impression of a rating request.

presenting this suggestion to the potential user base, however, this design proposal was completely dismissed as the stars evoked rather a rating request instead of the expected skill of the game.

In general, at the early phase of the mock-up creation process the Platform's core user interface was composed of the following components:

1. **Games Overview** aiming to provide a user with a comprehensive overview of all games taking place in a given location along with all the crucial details, such as count of players and goaltenders. Another important requirement is a variant of Games overview showing the games the user is already signed up for.
2. **Game Detail** providing a detailed information about the game, containing the following subsections:
 - **Game basic information** displaying location, time and other important features of the game.
 - **Game attendance** comprehensively showing the game attendees divided per their roles in the given game.
 - **Game location** (optionally) showing a map providing information on the location of the ice rink.
3. **User Profile** where the user profile can be edited and where the relevant user statistics are evaluated. Additionally, the component should act as an interface managing the followed players and displaying the user's followers.

The following subsections contain the presentation of Platform’s web application and mobile application mock-ups. While the focus of this work from the design and implementation perspective is mainly on the web client, the next subsection commences the mock-ups presentation with mobile mock-ups for the sake of completeness.

4.2 Mobile application mock-ups

One of the main challenges faced by a designer when drafting a mobile application is not to overwhelm the user with a lots of details, while simultaneously ensuring that everything relevant is present on the screen, not requiring the user to navigate all over the application to find the information he needs. In context of user experience, perfection can definitely be described in words of Antoine de Saint-Exupéry [31], who aptly stated:

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

The aim of the mock-ups was to adhere to this principle as closely as possible while iteratively asking for a feedback a group of 6 potential users. Main pieces of information that should be conveyed by the **Games Overview** can be summarized as follows:

- Time and date of the game
- Goaltenders’ participation
- Expected skill level of the game
- Games I am already attending

Distinguishing the attended games from the rest proved to be a user interface challenge. In the first mock-ups, a dedicated subscreen has been designed for this purpose. Introduction of a new screen, however, required adding a submenu to the Games Overview, rendering the otherwise clean and intuitive tab bar navigation scheme a lot more complex. Therefore a lot simpler approach has been taken – to add the „My Games“ criterion only as an additional filter property as shown in Figures 4.3a and 4.3b. Additional feedback was related to the green colouring of the attended games – some subjects were confused as it, according to them, conveyed a meaning of the game’s favourable status from the perspective of a logged in user. For example, for one test subject, the green evoked a presence of two goaltenders and an attendance of more than 15 players at the same time. This feedback is definitely worth a reflection in future iterations.

Figure 4.3c displays the User profile screen, showing a statistics of the attended games, displaying a list of player groups the user is member of² and listing all the players that are followed by the current user for the game recommendation mechanism purposes, which is analyzed in detail in Section 6.4.1.

Whereas the aim of the Games Overview screen was to show as few details as possible while being informatively dense, the purpose of the Game Detail overview is to display every attribute of a game. In order to avoid screen cluttering and at the same time not to require the user to scroll too much, Game Detail sub-menu was introduced. The first screen depicted in Figure 4.4a presents basic features of a game along with a concise attendance

²In later stages of the Platform’s logic definition, Groups in this specific sense were abandoned altogether – for details refer to Section 6.4



(a) All upcoming games

(b) Games I am attending

(c) User profile

Figure 4.3: Games overview & User Profile mobile mock-ups: First subfigure listing all games and the second, filtered one, listing only games the current user is attending. The third subfigure depicts the User Profile screen, showing user's game statistics and lists of followers/followees.

summary, the second one tries to leverage the available space to naturally sort the attendees based on their roles. The last sub-screen contains a map pointing to the rink where the game is taking place. During user testing, in 25% of cases an additional feature was proposed by the users, namely to be able to see all the upcoming games at a given ice rink after a click-through.

4.3 Web application mock-ups

Whereas the presented mobile mock-ups cover only a part of the Platform's functionality, web application mock-ups, with web application being the focus of this work, were designed to cover all the features available. The following content presents the key elements of the whole design, while the full page mock-ups can be viewed in Appendix A.



(a) Basic information

(b) Attendance overview

(c) Game location

Figure 4.4: Game detail mobile mock-ups: Three sub-screens displaying basic set of information about a game, detailed attendance overview and a map to show the ice rink's location

In general, the aim of Platform's web application design was to divide the functionality to only a limited amount of separate pages to keep the navigation as simple and lightweight as possible. After a careful analysis of the planned logic of the application, the page split was determined as follows:

1. **Dashboard** displaying My Upcoming Games, Games Calendar and Might Interest You components.
2. **Games** containing a filterable Games list.
3. **Athletes** showing a filterable overview of Athletes. along with their follow status with respect to the currently logged in user.
4. **Groups** providing an interface to create Player groups used in new game creation.

Given their importance, the following subsections present selected component mock-ups of the first three pages to provide the reader with the design and the application concept

overview, both of which can be studied further in the later content – selected features of the application logic are analyzed in the second part of Chapter 6 and the full mock-ups are available in Appendix A.

4.3.1 Dashboard

In simple terms, the aim of a dashboard in context of the user interface design is sparing the user an unnecessary navigation through the application, essentially saving user's time and effort to get the information he needs the most. Platform's Dashboard strives to achieve this goal by rendering these three components, as depicted in Figure 4.5:

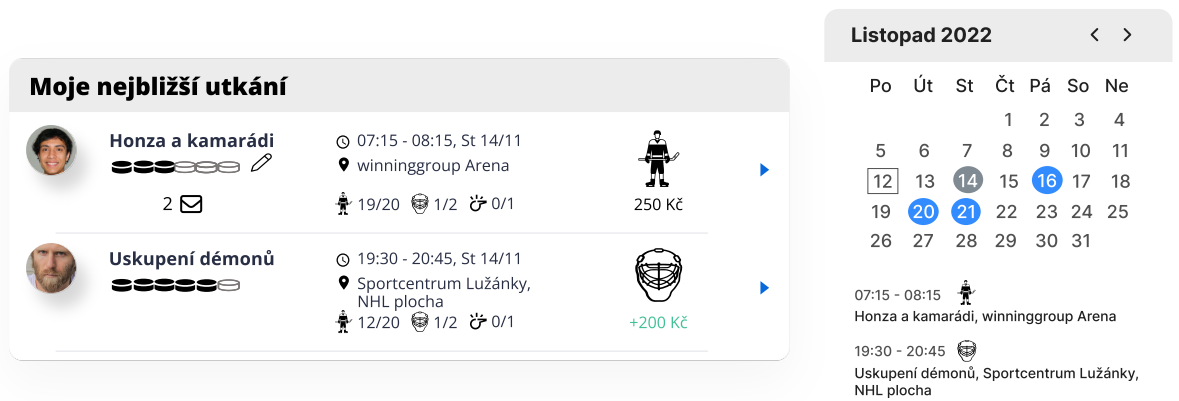
- **My Upcoming Games** containing the list of three closest games the user is attending to be able to directly access the most relevant games.
- **Games Calendar** providing a timeline perspective on the attended games.
- **Might Interest You** recommending games attended by the followed athletes, essentially meeting user's preference to play with the people he knows as discussed in Section 2.2. In addition, this component is also a part of the Game attendance facilitation concept presented in Section 6.4 as it also displays games organized by the followed people.

4.3.2 Games page

The Games overview page is the core of the application. From a high-level perspective, contrary to other platforms currently available³, the Games Overview page aims to display literally all hockey games taking place in a list to make the games market completely transparent, decreasing the information asymmetry between the game's potential attendees organizers. The Games Overview is composed of the following parts:

- **Games List:** A component displaying an infinite list of games sortable by date and athletes' attendance.
- **Games Filter:** An infinite list of data requires an efficient filter to extract the information tailored to a user's preference. As depicted in Figure 4.7, games can be filtered by a specific organizer, day of the week, time within the day as well as by the number of free places in specific game roles.
- **Game:** A component displaying all the important features of the game, such as count of players and goaltenders, as well as the price of attendance for the former group and the remuneration value for the latter. Additionally, to streamline the attendance management process, a user is able to join or leave the game directly from the list using the Attendance Selector Shortcut component, depicted in Figure 4.6 as the penultimate element on the right.

³Such as Týmuj.cz analyzed in Section 6.1.



(a) My Upcoming Games overview

(b) Games Calendar



(c) Might Interest You component

Figure 4.5: Dashboard components, summarizing the most relevant data for a user at one place. My Upcoming Games providing an overview of the three nearest games, Games Calendar providing a time planning dimension and the Might Interest You component showing games attended by the followed athletes.

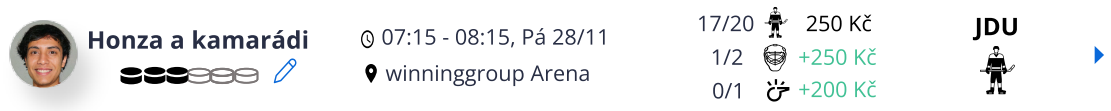


Figure 4.6: Game from the Games Overview list reporting all the relevant game features as well as Attendance Selector Shortcut component, enabling the user to join or leave the game without requiring him to open the Game Detail

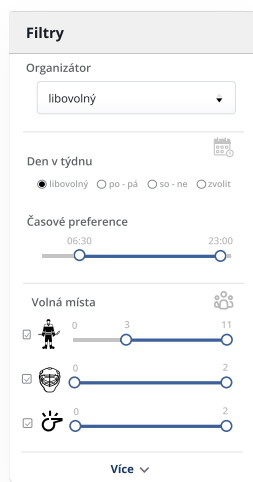


Figure 4.7: Games Overview filter

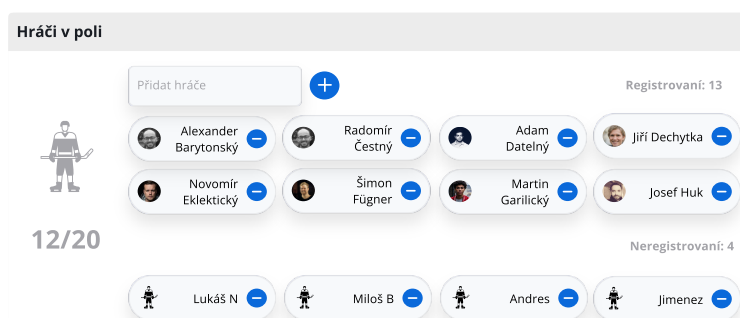


Figure 4.8: Attendance management component containing instances of User Badge component

4.3.3 Athlete page

While the Games Overview page is generally expected to be visited *before* a game as a means of connection with other players at a given time and place, Athlete Overview's main purpose is to foster the post-game social aspect of the Platform. Having recently attended a game with a few players the user does not know, he/she is able to look them up in the Athlete Overview, see how many games they have attended so far and what is their skill rating as rated by other players. If the user enjoyed their playing style and would like to play in the same games as they do in the future, he/she can *follow* them. The exact implications of this action and the comprehensive description of the follow mechanism is contained in Section 6.4.1. As was the case in the Games Overview page, Athlete Overview page also offers various filtering and sorting possibilities (for details refer to Athlete Overview page as depicted in Figure A.1 in Appendix A).

4.3.4 New Game page

Organizing a hockey game is a complex process and the Platform's New Game page aims to minimize the related inconvenience. Apart from the inputs to set the standard game parameters, such as time, place, expected amount of players, goalies etc., it provides three helper components as depicted in Figure 4.10:








				Utkání					
				v posledním	celkem				
				měsíci	kvartálu				
	Alexander Barytonský	15	2.1			5	15	40	 15 Sledovat
		37	3.4			10	37	75	
						2	5	8	

Figure 4.9: Item of the Player Overview list reporting player rating as well as the count of games he has attended in total. Also contains „Follow“ button to initiate a follow relationship

1. **Recently Organized** component listing four of the most recently organized games to be able to copy all the previously defined game settings to the game currently being created.
2. **Estimated Game Costs** as an informational component calculating the total game costs, taking into account automatically determined price of the rink as well as all the planned remunerations, simplifying the process of setting the player attendance price signalled to potential players.
3. **Available Groups** aiming to simplify the basic attendance setup of the players follow the organizer in the *opt-out* mode, i.e. allowing him/her to sign them up automatically on a created event.

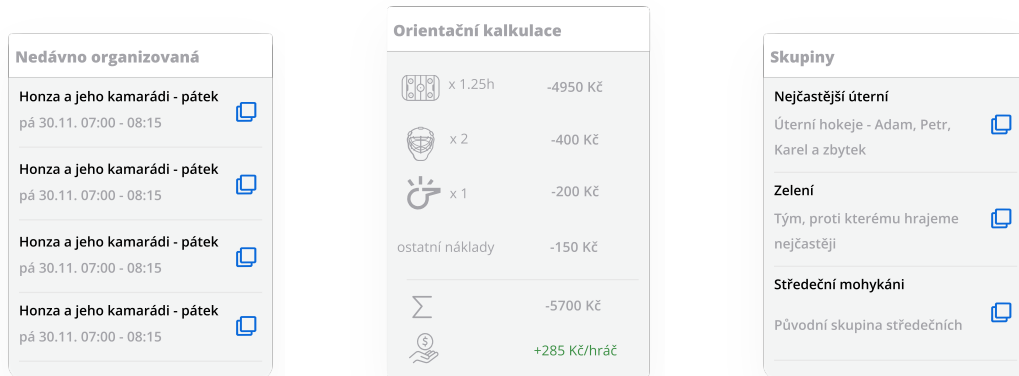
Admittedly, assuming that all the athletes desirable to attend the game are users registered within the Platform is unrealistic. Instead of forcing organizers to implement workarounds using dummy user profiles to avoid attendance count discrepancies, New Game page provides an interface to add unregistered players to a game, as demonstrated in Figure 4.8. Using this feature, apart from adding players from the limited set of the registered players⁴, an organizer can add arbitrarily-named anonymous athletes as players, goaltenders and referees to mirror the game planning reality as closely as possible. Both registered and anonymous athletes are represented by athlete badges, an element representing an athlete at multiple places within the Platform.

4.3.5 Game Detail page

Game Detail page, depicted in Figure A.4 in Appendix A, accessible from the games rendered within the Dashboard components or from the Games Overview component, is composed of four user interface components:

1. **Basic Information** displaying basic features of the game.
2. **Game Attendance** containing an overview of all athletes, both registered and non-registered, assigned to different roles within a game, represented by athlete badges.

⁴Not all players can be automatically signed up for an event – only those in the *opt-out mode* (see Section 6.4.1 for details)



(a) Recently Organized

(b) Estimated Game Costs

(c) Available Groups

Figure 4.10: Helpers for new game creation: Recently Organized component to provide an a helper to copy the details of the previously organized games, Estimated Game Costs component to help determine the final price for a player and Available Groups component enabling the organizer to easily add players who follow him in the *opt-out* mode to the game (see section 6.4.1 for details)

3. **Game Chat** providing an interface for simple chat-based discussion related to the specific game.
4. **Game Evaluation** component is an interface to evaluate the attendees of the game, accessible after the game. The evaluation provides anonymous feedback regarding the performance in the specific game for athletes generated by the system.

4.3.6 User profile page

Apart from an interface towards modification of user profile, the User Profile contains three additional components:

1. **Played Games Summary** summarizes the count of games played within the last 30 and 90 days and within the last half a year.
2. **Evaluation Summary**, depicted in Figure 4.11, provides overview of the monthly development of user's rating score as evaluated by his peers in the post-game evaluation in the Game Detail.
3. **Followers List** lists all athletes that follow the current user along with the information on the follow relationship type – whether it is currently set up as *opt-out* or *opt-in* (see Section 6.4.1 for details).
4. **Followee List**, analogically to the Followers List, enumerates all the athletes that are followed by the current user, with the possibility to toggle the follow relationship type.

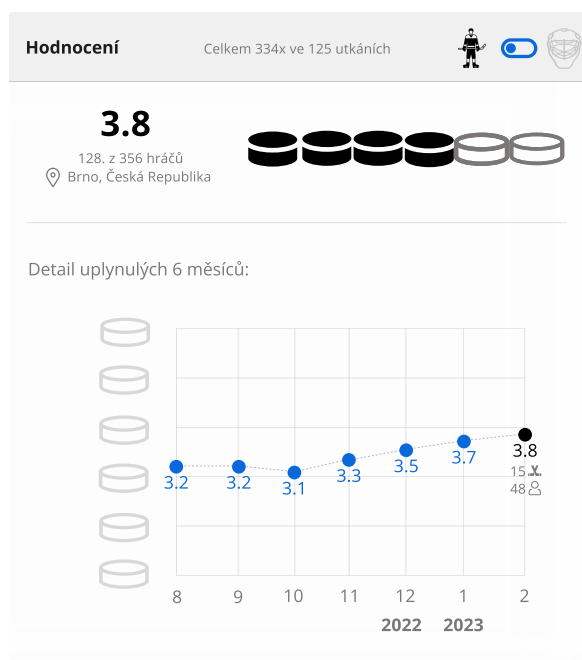


Figure 4.11: Rating component displayed in User's Profile showing the current skill rating of the user as rated over time by players who he played with

Chapter 5

Technological and Operational Aspects of Web & Mobile Applications

This chapter aims to introduce main parts of technology stack upon which the platform has been implemented. The introductory Section 5.1 will cover key parts of HTTP protocol specification and their relation to the REST API specification and a corresponding traditional REST API implementation. Consequently, main building blocks of the platform will be introduced. Starting from the top level, the platform can be decomposed into two parts:

1. **Python-Flask REST API server back end**, underlying principles of which are described in Sections 5.1 and 5.2, and
2. **React.js/React Native front end**, encompassing both the web application and cross-platform mobile application, both of which are discussed in Section 5.3.

Having introduced both front end and back end implementation starting points, the chapter will conclude with Section 5.4, describing utilities and technologies used for the facilitation of server API deployment, namely:

- Docker application containers and the Docker Compose utility,
- Elastic Container Service (ECS)¹ Docker container runtime provided by Amazon Web Services

5.1 REST API Server Network Communication

As the principles of building REST APIs are derived from the definition of the underlying HTTP protocol, the first part of this section will be dedicated to introducing those. In the second part, implications of HTTP features will be discussed in the context of REST principles and consequently towards building a REST API.

¹Available at <https://aws.amazon.com/ecs/>

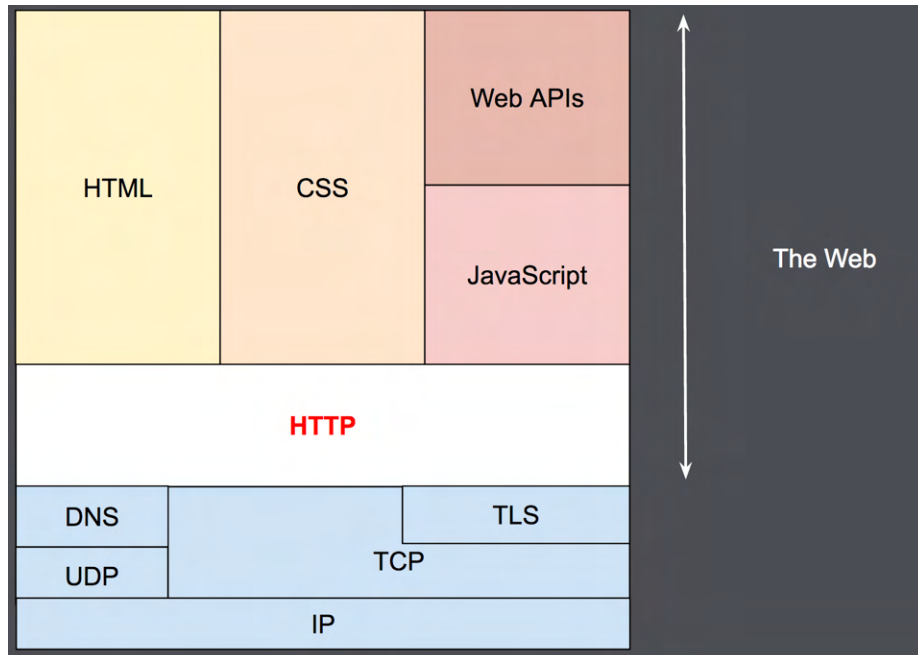


Figure 5.1: Network stack and HTTP protocol communication happening in the application layer on top of the transport layer [6]

Hypertext Transfer Protocol

The Request for Comments 2324 (RFC) defines the the Hypertext Transfer Protocol as follows [27]:

HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a *generic, stateless*, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers.

Being *generic* allows systems to be built independently of the data being transferred based on typing and negotiation of data representation. Being *stateless* allows the server to process each request independently, without any knowledge of the requests that were executed beforehand.

Figure 5.2 shows the HTTP placement on the *Application layer* of the *network protocol stack*. Built upon a reliable data-transmission Transfer Control Protocol (TCP), residing on the *Transport layer*, HTTP guarantees the integrity of the data at both communicating ends. As Totty et al. [34] summarize it, once a TCP connection is established, messages exchanged between the client and server will never be lost, damaged, or received out of order. From the developer’s point of view, that implies creating applications without worrying about data consistency, significantly reducing the burden of implementing sophisticated error handling mechanisms for every HTTP request created.

HTTP protocol is based on a simple scheme where a HTTP *request* is always followed by a HTTP *response*. The upper part of Figure 5.2 depicts most typical payload types of HTTP responses which together with the protocol itself represent the *Web*:

- HTML to provide web pages content,

- CSS styling web pages layouts,
- JavaScript to introduce dynamic elements into a webpage, and, finally,
- Web APIs providing application data during client-server communication.

All of the resources named above are fetched using several different request commands in HTTP syntax, called HTTP methods. Every HTTP request declares which method it uses and the server responds accordingly. The typical actions taken by the server based on four most common methods can be found in Table 5.1.

HTTP method	Description
GET	Send representation of a resource from the server to the client.
PUT	Store representation from the client into a named server resource.
DELETE	Delete the named resource from a server.
POST	Create a new resource based on the provided representation.

Table 5.1: Common HTTP methods [34] [29].

On an intuitive level, just by looking into Table 5.1, reader can gain a solid understanding of what a client wants a server to do - whether it's trying to get a representation, delete a resource, or connect two resources together. This intuition is referred to as *protocol semantics* of HTTP. However, based on this understanding, it is not possible to tell whether the representations being transferred are players in a team or books from a bookstore, since HTTP does not introduce any *application semantics*. That said, *application semantics* should always be driven by the *protocol* one. For example, as Richardson et al. note [29], no matter what the application domain is, a GET request should always fetch a resource representation from the server, ensuring semantic *consistency*.

Representational State Transfer (REST)

In the preceding paragraphs, a term *resource* has been mentioned multiple times. In a Web environment, a *resource* is anything that is important enough to be referenced as a thing in itself, the only condition being that it is assigned a URL. From the client's point of view, the resource is not important. What really matters is, as defined by Richardson et al. [29], a *representation* describing resource state.

REST is a software architectural style created to guide the design and development of web applications. In simple terms, this paradigm sets standards for a server transmitting a representation describing the current state of a resource (e.g. GET response) to a client, as well as for a client sending a *desired* representation of the resource [29] it wants to have created/updated on a server (e.g. POST request). Fielding [18] introduced six guiding constraints of REST architecture:

1. *Uniformity of interface* as a prerequisite of functioning interactions among the Web's components (clients, servers etc.).
2. *Client-server architecture* implementing the *separation of concerns paradigm*, allowing clients and servers to evolve independently and keeping the server components reasonably simple.

3. *Statelessness* implying no storage of client's state on the server. As a consequence, all requests are required to provide all the information needed to complete them.
4. *Cacheability* requiring the server response to label itself as cacheable or non-cacheable.
5. *Layered system* ensuring that each component is unable to see beyond the immediate layer they are interacting with.
6. *Code on demand* representing applets or scripts that are provided by the server in the form of a code that is run on client's side.

RESTful Application Programming Interface (REST API)

An API is a set of definitions and protocols for building and integrating application software. For example Red Hat [28] defines it as a contract between an information provider and an information user. REST API is a special case of this contract where the above defined REST constraints are respected. Table 5.1 contains the typical HTTP methods to be implemented within a REST API. To illustrate an usual communication with a REST API server in practice, as depicted in Figure 5.2, let us analyze in more detail perhaps the most complex one - a POST request:

1. Client (e.g. mobile application) creates a JSON object containing the *desired representation* of a *resource* on the server.
2. This JSON object is inserted into *body* of an HTTP request and sent over HTTP to a server.
3. REST API server receives the request and:
 - (a) Recognizes a POST request based on the „Method“ field in request header
 - (b) Checks if request's *endpoint* is registered with the given method.
 - (c) Looks for a handler of this endpoint and waits for it to process the request body.
 - (d) Handler informs the server if the request body was processed successfully or not.
4. Server returns a HTTP response to the client, containing the corresponding HTTP status code. The extract of the most frequently returned HTTP status codes for a POST request can be found in Table 5.2. Comprehensive overview of all HTTP status codes is provided by the corresponding RFC 7231 [17].

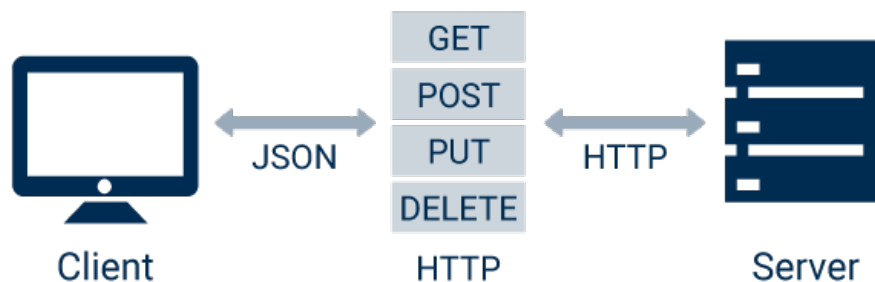


Figure 5.2: REST API communication flow [30]

HTTP status code	Potential reason
200 OK	The resource was successfully created and is enclosed in the body of the response
201 Created	New resource was created as a result of the request
400 Bad Request	Server could not understand the request due to malformed syntax
401 Unauthorized	Unauthenticated access to the resource has been attempted
403 Forbidden	Despite being authenticated, client does not have access rights to the content.
404 Not found	Server cannot find the requested resource.
500 Internal Server Error	Server has encountered a situation it does not know how to handle.

Table 5.2: Most frequently occurring HTTP status codes as defined by the corresponding RFC 7231 [17]

The actual syntax of HTTP communication is indicated by the GET request listed in Listing 5.1. Every HTTP request has to define its *method*, *endpoint* and HTTP protocol *version* based on which the communication will be conducted. This version is then confirmed right in the beginning of a HTTP response header, which also defines type of the content enclosed in the request body. In this case it is a JSON object, therefore **Content-Type** header is set to `application/json`. What follows is the actual request body referencing a representation of `/api/athlete` resource, containing a collection of players, each described by his attributes.

5.2 API Server Back End Technologies

In the previous section, principles of REST API communication as well as of underlying HTTP protocol were introduced. This section will focus on the engine that ensures the requests (as for example 5.1) are properly handled, i.e. the API itself. Luckily, there is a whole array of efficient frameworks in virtually all the most popular programming languages, abstracting the complexity of HTTP communication away from a developer. In Python language, for example, developer can choose (among others) between full-stack frameworks, like Django², and rather light-weight ones, such as Flask³ micro framework. While the latter focuses on the main things web framework has to offer, like URL routing, the former integrates many features like sessions, authorization, authentication, templating and database access out of the box [21]. That said, less complex frameworks can easily be extended with those, the main advantage being developer's freedom to choose whichever extension he prefers.

²<https://www.djangoproject.com>

³<https://flask.palletsprojects.com>

```

GET /api/athlete HTTP/1.1                                1
Host: www.platform.com                                  2
-----                                                3
HTTP/1.1 200 OK                                         4
Last-Modified: Thu, 10 Jan 2021 01:45:22 GMT           5
Content-Type: application/json                          6
{                                                       7
  "athletes":                                           8
  [                                                       9
    {                                                    10
      "id": 1,                                           11
      "name": "John Doe",                                12
      "skill_level": 3                                   13
    },                                                  14
    {                                                    15
      "id": 2,                                           16
      "name": "Martin New",                              17
      "skill_level": 2                                   18
    },                                                  19
  ]                                                    20
}                                                       21

```

Listing 5.1: HTTP GET request & response

From the perspective of web framework features, a Python REST API server back end requirements are mainly the following:

1. *Routing* of API endpoints to the corresponding request processing methods.
2. *Authorization* of a user to verify client's identity.
3. *Authentication* of a user with each request sent. This is required given the *statelessness* requirement of REST discussed in Section 5.1.
4. *Database access* to be able to persistently store data to be processed.

Flask micro framework was chosen based on its rather shallow learning curve compared to the full stack frameworks like Django. Therefore, the rest of this section will be devoted to possibilities that Flask offers in this regard. But before diving into that, a protocol ensuring smooth communication between a web server and a Python application has to be introduced.

Web Server Gateway Interface

Python Enhancement Proposal (PEP) 333 [14] presented a standardized interface between Web servers and Python Web frameworks/applications, called *Web Server Gateway Interface* (WSGI). This interface is supported by all Python frameworks, aiming to provide [3]:

- relatively simple yet comprehensive interface capable of supporting all (or most) interactions between a Web server and a Web framework
- support for middleware components for pre- and post-processing of requests

Flask

WSGI protocol is used to pass all the HTTP requests from client to a Flask application instance that is instantiated as follows:

```
from flask import Flask                                     1
app = Flask(__name__)                                     2
```

Once the application is started using `app.run()` command, it enters a loop and waits for the incoming request to be processed. Flask engine has to keep a mapping of URLs to Python functions in order to know what code should be run for each URL requested. This mapping is called a *route* and can be used via the `app.route` function *decorator* as follows [20]:

```
@app.route('/')                                           1
def homepage():                                           2
    return '<h1>Hello World!</h1>'                         3
```

This way of mapping ensures that when a user enters `http://www.myflaskweb.com/news` into a web browser, Flask routes this GET request using WSGI protocol to a Python function decorated by `app.route('/news')`. Also, this decorator accepts a second parameter *methods* that is used to enumerate HTTP methods that will be accepted at this URL (endpoint). If omitted, its value defaults to GET method. If multiple methods are provided, the decorated function is called for all of them and a developer is able to implement actions depending on the request type within the same view function.

To facilitate request processing, Flask application *context* is available within those decorated *view* functions. For example, details of the currently processed request can be accessed via the *request* variable. Context variables are accessed in a similar way as the global ones, even though they are technically *not* global due to a multi-threaded nature of a web server. Once started, a web server launches a pool of threads and selects a thread to process an incoming request. As Grinberg notes [20], each such thread needs to have access only to the specific request currently processed which is exactly the object the *request* context variable refers to.

It is entirely feasible to implement a REST API server using the Flask features described above. The process would follow this pattern for each API endpoint:

1. Creating a *view* function and decorating it with the endpoint URL as well as enumerating all the HTTP methods this function is able to handle.
2. Bearing in mind that each *view* function implements behaviour for all stated HTTP request methods, the function is likely to be branched into many directions - a GET request on a specific endpoint should probably trigger very different actions to its POST counterpart.
3. Each *view* function returns HTTP status code that is processed by the WSGI engine and returned to the client. Along with it, especially in case of a GET request, it returns a representation of a resource requested by the client, most often a JSON object.

This approach might be suitable for very simple APIs where number of endpoints and resources is rather low and obtaining resources' representations is not computationally complex. For a more complex Flask REST API project, such as the one built within this work, however, it might be better to take advantage of a specialized extension tailored for

this purpose. One of such extensions is called Flask-RESTful and will be described in the last part of this section. But before that, it is important to know how *resources* and their representations will be represented in various parts of the application. This is a task for the SQLAlchemy Python module and the Flask-SQLAlchemy extension.

SQLAlchemy

SQLAlchemy is a high-level open-source code library simplifying the process of working with relational databases such as Oracle, DB2, MySQL, PostgreSQL, and SQLite for Python developers. Its main mission is to abstract away all the peculiarities in working with relational databases directly, enabling developers to seamlessly migrate from one database system to another. Copeland summarizes [9] SQLAlchemy's two-mode operation as follows:

1. *SQL Expression Language* (SQLAlchemy Core) providing only a very light abstraction on top of SQL and preserving the schema-centric way of thinking about relational databases and querying their content.
2. *Object-Relational Mapping* (ORM) providing a high-level approach to data management with main focus on the domain model of the application and more idiomatic way of querying the database, known also from ORM frameworks from other popular programming languages.

These two modes are not mutually exclusive - quite on the contrary. ORM mode is built upon the SQLAlchemy Core, meaning that a developer generally preferring the ORM approach is able to switch into the schema-centric one whenever a situation requires it.

As the API *resources* of the server implemented within this work are managed using the SQLAlchemy ORM system, it might be helpful to analyze an example of a *player* entity and its setup and processing using the framework. This insight is provided by Listing 5.2. Especially noteworthy is line 15 with its idiomatic and, often referred to as „Pythonic“, way of querying the database.

```
from flask_sqlalchemy import SQLAlchemy      1
...                                          2
db = SQLAlchemy(app)                        3
class Player(db.Model):                    4
    id = db.Column(db.Integer, primary_key = True)  5
    name = db.Column(db.String(100))         6
    skill_level = db.Column(db.Integer)      7
                                          8
def __init__(self, name, skill_level):     9
    self.name = name                       10
    self.skill_level = skill_level         11
...                                         12
player = Player("John Doe", 5)             13
db.session.add(player)                     14
skilled_player = Player.query.filter_by(skill_level > 4).all() 15
```

Listing 5.2: Database entity setup and processing in SQLAlchemy.

Flask-RESTful extension

Flask-RESTful is an extension for Flask that simplifies building REST APIs. Its key building block are *resources* that built on top of Flask pluggable *views*⁴. Resources are represented by classes extending `flask_restful.Resource` class. That way, as Flask-RESTful authors [23] note, a developer is easily able to access all HTTP methods requested at a given endpoint just by defining methods of corresponding names on those resources. Listing 5.3 demonstrates this concept. Similarly to Flask, Line 8 shows that you can return a (*iterable*, response code) tuple and the back end automatically converts it into a proper HTTP response with the provided status code.

```
from flask_restful import Api 1
from flask_restful import Resource, reqparse 2
class Player(Resource): 3
    parser = reqparse.RequestParser() # For request body parsing 4
    parser.add_argument(...) 5
    6
    def get(): 7
        return { 'players': [ {'name': 'John Doe'} ] }, 200 8
    def post(): 9
        data = Player.parser.parse_args() 10
        # Process data dictionary containing a parsed request body 11
    12
app = Flask(__name__) 13
api = Api(app) 14
api.add_resource(Player, '/api/player') 15
```

Listing 5.3: Flask-RESTful in practice

Building a Flask REST API

A combination of Flask framework core along with its SQLAlchemy and Flask-RESTful extensions constitutes a powerful tool for building a flexible and easily maintainable REST API server. Synthetizing knowledge of Flask high-level principles and a typical usage of both extensions as demonstrated in Listing 5.2 and Listing 5.3, it is reasonable to approach the implementation as follows:

1. Every resource (endpoint) is represented by a class extending *flask_restful.Resource* class
2. In general, following methods are generally implemented within the resource class, though some of these might be skipped:
 - *get* for obtaining representations of the resource
 - *post* for resource representation creation
 - *put* for updating the resource representation
 - *delete* for resource representation deletion
3. Each of the methods representing HTTP methods might be regarded as *views* in Model-View-Controller architecture.

⁴<http://flask.pocoo.org/docs/views/>

4. These *views* are calling *controller's* API to perform a desired action with the data relevant for the given resource. *Controller* is representing a layer between these *views* and the *model* layer, encapsulating implementation of underlying data processing and therefore keeping the *views* codebase as simple as possible.
5. The *model* layer models the underlying entities as Python objects extending SQLAlchemy's ORM base classes. This mechanism enables leveraging of SQLAlchemy APIs for querying the actual database engine.

5.3 Application Front End Technologies

The last section focused on REST API and server back end fundamentals to lay the groundwork for implementation of Platform's data management engine. This section aims to provide the same basis for Platform's user-facing applications. To begin with, reasons for selecting TypeScript as the implementation language for Platform's front end will be provided. Secondly, underlying principles of React JS framework will be introduced. And lastly, this section will be concluded with an analysis of React Native as a derivate of React JS for cross-platform mobile development.

TypeScript versus JavaScript

Traditionally, JavaScript has been a synonym for the front end web development. Starting in an era where only a handful of dynamic elements were present on a web page, lately it has been getting even greater traction with the paradigm shift towards the complex web *applications*. Supposedly, that is why some of JavaScript's features, or weaknesses, for that matter, have recently started to cause inconvenience to corporations managing extensive web-based applications.

The main weakness of JavaScript is the absence of *type safety*. For example, in *statically-typed* programming languages, such as C++ or Java, the following expression is invalid due to the obvious type incompatibility:

1 + []

1

JavaScript, on the other hand, silently evaluates the expression with the result being the string „1“. Instead of throwing an exception, or at least signal a warning, it does its best to evaluate it, even though this expression makes no actual sense. Crucially, this type of errors is not identifiable until the application's *runtime*. As Cherny [5] states, this creates a very large time gap between the moment when an error is *made* and when it is *discovered*, wasting a lot of development effort.

TypeScript was introduced in 2012 as a type layer above JavaScript aiming to prevent these and similar situations from happening. Let us analyze its compilation process described on Figure 5.3. Firstly, TypeScript compiler compiles TypeScript into an *abstract syntax tree* (AST) which is consequently type-checked by the *typechecker*. This process often happens directly in developer's IDE *without* needing him/her to compile the code, let alone to run it, strongly facilitating prompt error fixes. Having passed the type check, TypeScript AST is compiled into plain JavaScript. From this point on, the process continues as it normally does for vanilla JavaScript with the crucial distinction that all of the type-related errors were already eliminated without running the program even once.

Based on the reasons above and despite placing additional type management burden on a developer, TypeScript development has proven to be more time-efficient. For un-

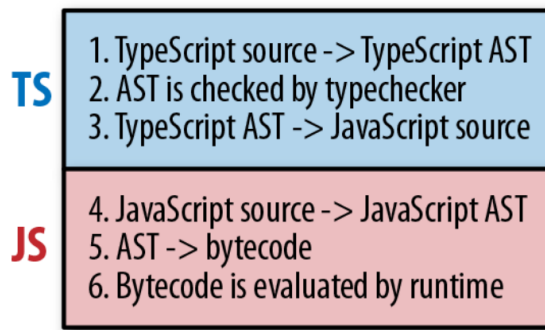


Figure 5.3: Compiling and running TypeScript. TypeScript compiler compiles TypeScript into an abstract syntax tree (AST) which is consequently type-checked by the typechecker. Having passed the type check, TypeScript AST is compiled into plain JavaScript. [5]

derstandable reasons, this holds especially true when dealing with large-scale applications. Therefore, it has been chosen as a programming language for development of Platform’s both web (ReactJS) and mobile (React Native) front end clients.

Asynchronous JavaScript And XML (AJAX)

The very same simple principles of HTTP request and response described in earlier sections that enabled the rapid proliferation of the Web in 2000s posed a serious challenge for web interaction designers. At that time it seemed technically impossible to reach levels of smoothness and responsiveness provided by the user interfaces of desktop applications due to the inherent nature of HTTP communication. As Garrett [19] notes, most user actions in the web user interface triggered a HTTP request back to a server whose task was to respond with the HTML code of the page that should be displayed to the user based on his action. Most importantly from the user experience perspective, the *synchronous* nature of this communication implied that the only thing the user could do in the meantime was to wait.

The first and most important step towards solving this issue was the introduction of AJAX (Asynchronous JavaScript And XML), which implemented *asynchronous* client-server communication by adding an intermediary – AJAX engine – between the user and the server. In this scheme, a user action generates a call to AJAX engine instead of directly generating HTTP request. Often, this call can be a simple request to validate form data, not requiring any server action whatsoever. If resolving the request requires server communication, the AJAX engine takes care of creating the corresponding HTTP request without stalling a user’s interaction with the application, as noted by Garrett [19].

5.3.1 React

Taking advantage of AJAX introduction, Single Page Applications (SPA) have been gaining momentum in front end web development since 2010. Contrary to the traditional browsing model described above, SPAs interact with user by dynamically rewriting only a single one⁵ with the data provided from the server.

⁵Precisely speaking, usually there is a comparatively low number of *base* pages, not just only one.

Taking advantage of AJAX, multiple frameworks facilitating SPA creation in an more efficient manner than vanilla JavaScript and JQuery were developed - among others AngularJS or Ember.js. All of these frameworks, however, are inherently limited from a performance perspective due to the way they approach Document Object Model (DOM). In essence, interactive web pages are created using DOM manipulation. First generation of SPA frameworks reloaded the whole DOM with each its update, regardless of its size, wasting a lot of memory and lowering the performance.

With DOM reload representing the front end performance bottleneck, one of the key features of React is its *Virtual DOM*. It serves as a layer between the JSX description of the page defined by the developer, i.e. what the page should look like, given the application state at that point in time, and the actual steps that are needed to be taken in order to get to the layout desired. This process is depicted in Figure 5.4. The main purpose of Virtual DOM is to ensure the smooth user experience by calculating the smallest set of required changes possible to end up with the layout required by the new application state.

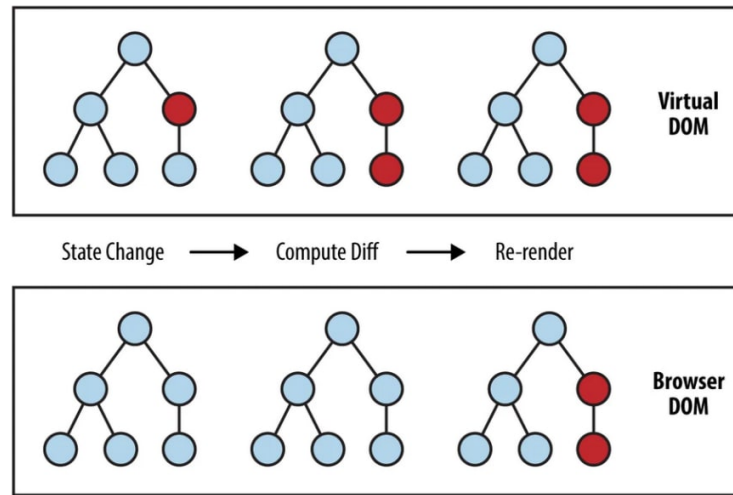


Figure 5.4: React Virtual DOM, serving as a layer between the JSX description of the page defined by the developer, i.e. what the page should look like and the browser DOM manipulation necessary to get the result required. [15]

Contrary to the first generation of SPA frameworks, which were aiming to cover multiple parts of the *Model-View-Controller* paradigm, Wieruch [36] stresses React's sole focus on the *View* part. From an architectural standpoint, React application is composed from hierarchically organized *reusable components*, each of which provides a rather small piece of HTML mark up of the resulting page. A HTML code output of a component is generated, as nicely put by Eisenman [15], using a mixture of JavaScript and XML-esque markup, known as *JSX*.

JSX is a syntax extension of JavaScript. It is essentially an implementation of React's paradigm to embrace the fact that rendering logic is inherently coupled with other UI logic. Event handling, state changes, preparation of the data for display are all inherently intertwined. Therefore, JSX implements a separation of *concerns* rather than a separation of *technologies* paradigm.

There are two basic approaches towards implementing React components. They can be implemented either as:

1. *classes* extending `React.Component` class from `React` and implementing its `render` method, or
2. *functional components* that are ordinary TypeScript functions with a JSX object as a return value.

Recently, React community has been recommending the *functional* approach, especially with the introduction of *hooks*⁶ that cannot be used in the *class* one. Listing 5.4 shows a common structure of a React application implemented functionally, while lines 9 - 12 demonstrate a typical JSX return value of a React component.

```

import React from 'react'                                     1
                                                                2
type PlayerProps = {                                         3
  name: string                                               4
  skillLevel: number                                         5
}                                                             6
                                                                7
const Player = ( { name, skillLevel }: PlayerProps) => {     8
  return (                                                   9
    <h1>Hello, {{ name }}!</h1>                               10
    <p>Work harder, your skill is only {{ skillLevel }}!</p>  11
  )                                                         12
}                                                            13
                                                                14
function App() {                                             15
  return <Player name="Adam" skillLevel=3 />;                16
};                                                            17

```

Listing 5.4: Simple React TypeScript application. A functional component `Player`, taking props arguments according to the predefined interface and returning a JSX value.

5.3.2 React Native & Multiplatform Mobile Applications

React Native (RN) is a JavaScript library for cross-platform mobile development based on React JS, created by Meta Inc. (Facebook Inc.) in 2015. The introduction of React Native enabled a large community of web front-end developers to write mobile applications that feel like being written natively with low additional study overhead compared to native mobile development. Additionally and crucially, cross-platform development enables developers to target audience in both main mobile platforms iOS and Android with just one codebase, requiring only relatively minor platform-specific adjustments.

React Native was developed taking advantage of its React's Virtual DOM. While preceding paragraphs related to web application rendering praised its performance implications, it is its power of abstraction that really stands out. Being the abstraction layer between developer's code and the actual rendering process, Eisenman points out [15] that React already "understands" what your application is *supposed* to look like. Therefore, it can be linked to whatever target platform (rendering engine) at hand, provided there is a *bridge* implemented for it. This capability was exploited by a group of engineers at Meta, Inc., who implemented a *bridge* between JavaScript and the native rendering engine (iOS/Android) to render native mobile views, effectively creating React Native framework.

⁶<https://reactjs.org/docs/hooks-intro.html>

Analyzing the architecture of React Native, three main communicating building blocks can be identified, as depicted on Figure 5.5:

- **JavaScript** part handling the app logic and sending objects (properties that need to be reflected in final layout, function callbacks) to Native part,
- **RN bridge** providing the interface between JavaScript and Native parts, and, finally
- **Native part** taking care of rendering native UI elements and sending events back over the bridge to the JS part

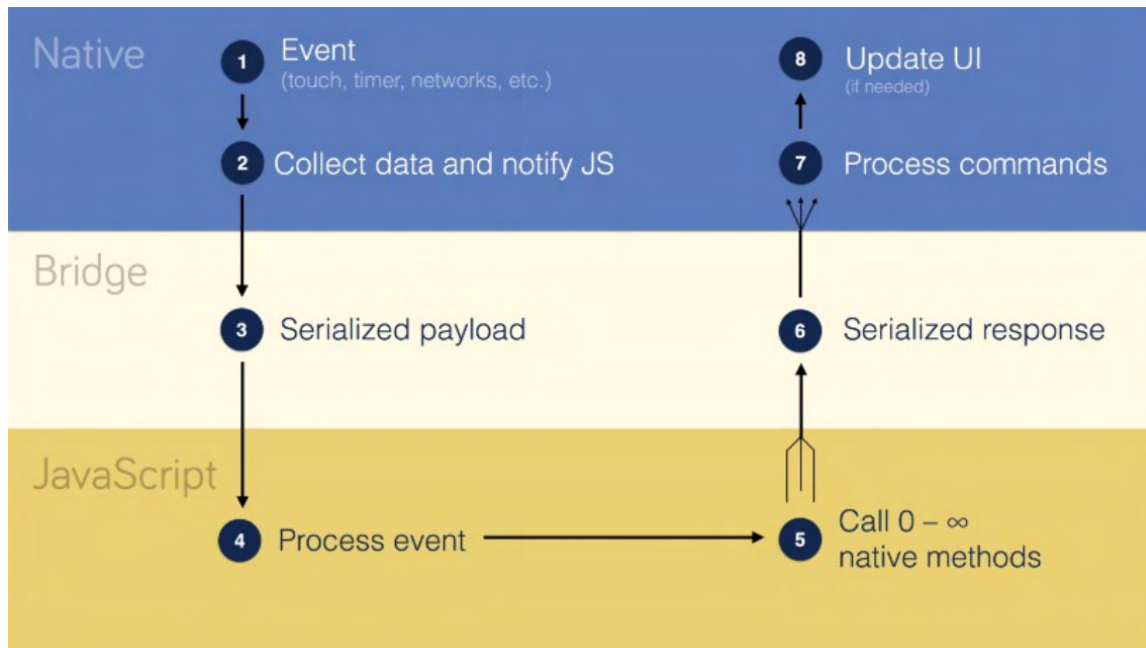


Figure 5.5: JavaScript side handling the application logic, the Native part rendering the native UI elements and the RN bridge providing the interface inbetween.

Once an app is launched, its main (UI) thread is launched by the operating system (iOS/Android). Afterwards, as Cook [8] describes, the JavaScript and the *shadow* thread are started by the *main* thread. The mobile *view* defined in JavaScript (using JSX) is calculated in the *shadow* thread and finally sent to UI thread for rendering.

For example when a text needs to be changed in the UI, its new contents are serialized to be sent over the bridge. Simultaneously, it might be required to disable a button. All these updates are batched together and sent to the native side at the end of each iteration of JS *event loop*⁷. The payload of the message sent over the bridge can be virtually anything – for example serialized function callbacks that should be triggered after a specific UI event. Since the UI knows there is a callback associated to the event, it lets the JavaScript part know once that specific event happened. Actually, the UI can send the information about an event even without specifying the recipient on the JS part, letting the event listeners capture it themselves. This approach might, however, make debugging of the application much harder and for example Cook [8] discourages developers from it.

⁷Event loop is the core concept of the JavaScript’s asynchronous programming. For more details please refer to <https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>

Naturally, the benefits of implementing just one application for both two most important platforms comes with a trade-off. From an performance perspective, communication between the JavaScript side and the native UI over the RN bridge works flawlessly in most applications that do not require heavy computational effort. As an illustration of a potential computational bottleneck let us assume an app displaying an infinite list of complex animations provided as an example by Cook [8]. As a user scrolls down through the list, these scroll events are sent from the UI into the JavaScript thread where appropriate actions are dispatched, the *shadow* thread recalculates the *view* layout and sends it back to the UI. The volume of these messages sooner or later overwhelms the RN bridge’s capacity, leaving the user with a blank screen.

Being derived from React JS, the React Native’s *views* are created by using previously introduced JSX as well. Then, under the hood, as described by Eisenman [15], the *bridge* invokes the *native rendering APIs* in Objective-C (iOS) or Java (Android). Contrary to React JS, where the JSX gets converted into a HTML template, creating *web views*, mobile UI generated by RN engine will be composed of *native UI components*, making the user experience native-like indeed. The path from JSX into the actual generated UI can be studied on Figure 5.6.

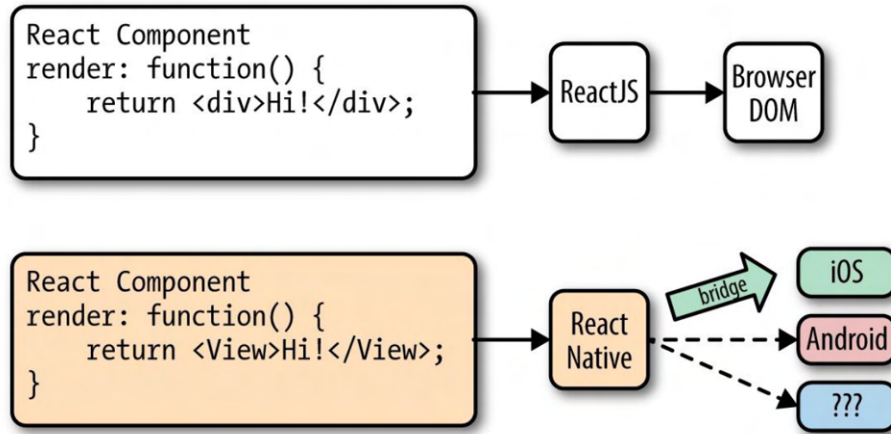


Figure 5.6: Rendering in ReactJS and React Native [15]. JSX return value converted into a HTML template in case of ReactJS and into native UI elements in case of React Native.

Predictably, React Native’s different rendering targets manifest themselves in JSX return values of React functional components⁸. As shown in Listing 5.5 on Lines 5 - 9, `<View>` and `<Text>` elements are generated instead of HTML’s `<div>` and `<p>`. Additionally, whereas in plain React the styling of elements is usually kept in a separate CSS file, reflecting traditional web development practices, in React Native the layout is defined using `StyleSheet` class from the React Native’s core, the instance of which is usually tightly coupled with the styled component and kept within the same file.

⁸Or implementation of the `render` method in case of the `React.Component` class extension approach

```

import React from 'react';
import { StyleSheet } from 'react-native';

export const GameExcerpt = (game: IGame) => {
  return (
    <View style={[ styles.container ]}>
      <Text style={[ styles.name ]}>{ game.name }</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: { ... },
  name: { ... }
});

```

Listing 5.5: An example React Native component. Styled by the `StyleSheet` class instance and returning `<View>` and `<Text>` elements instead of HTML's `<div>` and `<p>`

5.4 Application Deployment Technologies

There are various options to consider when planning to publicly deploy a Flask server API. A developer can either use his/her own machine equipped with a HTTP server and a Python interpreter or purchase services from one of the cloud server providers. Choosing the cloud option, a developer usually aims to mirror the development one as closely as possible to finally create development environment that is *nearly identical* as the development one. Achieving 1:1 relationship is, however, usually hardly possible. As a result, different runtime environments may lead to different application behaviour, which in turn significantly complicates both development and debugging efforts.

Docker, a containerization open source project that started in 2013, has been developed to protect developer community from this inconsistency, or, more precisely, to prevent it from occurring altogether. Since Docker is used for deployment of Platform's server API, the following text introduces its high-level principles and indicates their potential applications to solve the deployment task at hand. To publicly deploy a Docker container, it is necessary to have an access to a Docker runtime provider. An example of such a service is the Elastic Container Service, managed by Amazon Web Services (AWS), briefly discussed at the end of this section.

Docker

Virtualization in general can be implemented taking two different approaches, clearly distinguishable by locating the computer architecture layer where it resides:

1. *Hypervisor* virtualization running on a physical HW via an intermediate layer. The purpose of this layer, *hypervisor*, is to abstract computer hardware into software, enabling multiple virtual machines to run on just one set of hardware.
2. Virtualization based on operating system's kernel, providing an application deployment engine on top of a virtualized container execution environment, using the operating system's normal system call interface and thus requiring limited overhead, as stressed by Turnbull [35].

Docker is an example of the second approach. Not suffering from the overhead imposed by the *hypervisor* layer, it is a very lightweight and highly performant virtualization technique, providing developers with consistent development environments. There are two main components within the Docker environment – *images* and *containers*:

1. **Image** is an immutable (read-only) template of the application environment, containing a set of instructions for creating a *container*. This template defines all the prerequisites for the specific application - its source code as well as all the dependencies required. From the perspective of traditional virtualization, it can be regarded as a *snapshot* of a virtual machine. A Docker image is defined in the *Dockerfile*. An example, shown in Listing 5.6, demonstrates its *layered* structure. For example, Line 1 states that a Python image should be used as a base image, while each line represents an additional *immutable* layer that should be stacked on top of the preceding one.

```
FROM python:3.9.1           1
EXPOSE 5000                 2
RUN mkdir /api             3
WORKDIR /api               4
COPY requirements.txt /api/requirements.txt 5
RUN pip install -r requirements.txt 6
```

Listing 5.6: Dockerfile defining Docker image

2. **Container** represents an instance of an *image* completely isolated from the host system. While all the layers stacked during the image creation are strictly immutable, container creates a *read/write copy* of an image, leaving the original image intact, and places an additional *writable* layer on top of those. Importantly, since the underlying image is a *copy*, this writable layer adheres to the image immutability principle.

REST API containerized deployment

There are two main building blocks of a REST API - an application processing HTTP requests and implementing all data processing logic and a database engine ensuring data persistence. As they are independent, it might be reasonable to put each of them in a separate *image*, making them run in a different Docker *container*. Fortunately, Docker environment provides an utility called *Docker compose* that allows its users to run multi-container Docker applications based on a single YAML⁹ file configuration. Using *Docker compose* is basically a three step process [10]:

1. Setting up application environment using Dockerfile.
2. Defining the *services* that should run together, each in a separate *container*.
3. Launch the whole application setup using `docker compose up`. This command ensures that all the required images are either built or fetched and initiates the corresponding application containers.

Listing 5.7 demonstrates a minimal Docker compose configuration for the two *services* - an API server and its database. Each of this services needs to have its base image provided. For the API application service, the basic semantics of its YAML configuration is the following:

⁹YAML: YAML Ain't Markup Language - <https://yaml.org/>

```

version: '3'                                1                                19
volumes:                                    2                                20
  dbdata:                                    3                                21
                                             4                                22
services:                                    5                                23
  api:                                        6                                24
    image: api-image                          7                                25
    container_name: api                       8                                26
    build: .                                  9                                27
    environment: # DB access                  10                               28
    links:                                    11                               29
      - db                                    12                               30
    depends_on:                               13                               31
      - db                                    14
    volumes:                                  15
      - ./api                                 16
    ports:                                    17
      - "80:80"                               18

```

Listing 5.7: Defining application & database services in `docker-compose.yml`

1. Wait till the `db` container is instantiated and running (line 13).
2. Create an image named `api-image` based on the Dockerfile present in the current directory (lines 7 - 9) and create a container named „api“ running an instance of this *image*.
3. Map the host's port 80 to the port 80 in container¹⁰.

Contrary to the service representing the main application, the database service configuration is based on the prebuilt MySQL Docker *image*, maintained by the MySQL team and available in Docker repository. During the first container launch, this image is downloaded and made locally available to consequent launches. In this case, host and container ports are set to different values - the host port is set to 6033 in order not to interfere with the 3306 port traditionally reserved for the MySQL communication. Practical implication of this setting is a redirection of all traffic to the local 6033 port to the container's 3306 MySQL port. While database data persistence is its essential feature, the immutable nature of images along with the independence of container runs render any data persistence almost impossible. This issue is resolved by mounting a dedicated host's local data storage into the container using *volumes* element as demonstrated on lines 30 – 31.

¹⁰tcp:80 is a network communication port reserved for HTTP communication - see TCP Port Number Registry at <https://www.iana.org>

Chapter 6

Amateur ice hockey organization platform

While Chapter 2 summarized the status quo of the amateur ice hockey matches organization and indicated the key potential inefficiencies, namely lack of goaltenders available for amateur ice hockey games, the aim of this chapter is to emphasize the Platform's approach towards addressing it. Before doing that, it is reasonable to conduct an analysis of existing platforms that try to address some of the inefficiencies to identify their strengths and weaknesses.

To conclude this chapter, an insight into Platform's features is provided, such as the *lazy registration* approach to facilitate user engagement or specifics of the follow mechanism variant proposed for the Platform.

6.1 Analysis of existing services

There are multiple applications that foster organization of sport teams, be it amateur groups or even more professional ones. Most of them, however, are focused on management of a rather rigid group of people, i.e. a fixed number of players over time. When playing amateur ice hockey outside of organized leagues, as discussed in Section 2.1, the groups playing together tend to be all *but* rigid. A typical and in Czechia very popular platform falling into this category is Týmuj.cz¹, features of which are to be analyzed in the following text.

In soccer, for example, there is an application introducing the *on-demand* element into games organization, called CeleBreak². Its features and user stories will be discussed in the second part of this section as they might present a valuable source of inspiration for introducing a variant of such an *on-demand* element into ice hockey organization.

Týmuj.cz

Týmuj.cz is an online platform for facilitation and management of sport teams. It is a prime example of the approach mentioned above - with the focus on management of rigid teams, a *team* being the main entity of the platform. The application usage scheme implemented within the application can be summarized as follows:

¹<https://tymuj.cz/>

²<https://celebreak.eu/>

- A team manager (coach) creates a team and invites another members into the team via an e-mail link.
- Once the team is set up, team manager (or other person with administrative role) creates events specific to the given team for which only members of the given team can sign up.
- A team can be later joined by new players via an e-mail link.

As already mentioned, this approach is reasonable for groups where the amount of players over the year/season remains fairly stable, i.e. the set of players attending games throughout a year remains more or less constant. Put differently, games created within the team are of *no interest* to external players. Especially managers and members of professional/semi-professional sport teams can really leverage Týmuj.cz’s versatility of team management.

On the other hand, platform’s versatility inevitably implies that sport-specific needs are either completely unsolvable or require cumbersome workarounds. Clearly, applying these notably reduces the benefits Týmuj.cz offers for the team managers as well as for the individual team members. For example, in Section 2.1, the importance of participation of a pair of goaltenders has been discussed. Within Týmuj.cz platform, there is no way to distinguish player roles. With goaltenders being the key components of any game, it is important for an organizer to signal their presence to other players, making them more likely to participate as well. Bearing this in mind, as a workaround, the team manager might increase the desired required count of players for the game. Looking at Figure 6.1a to provide an example, let us increase the participation limit from 20 to 22 players for the goaltenders to be able to sign up. Since these two additional places are not role-bound, two issues inevitably arise:

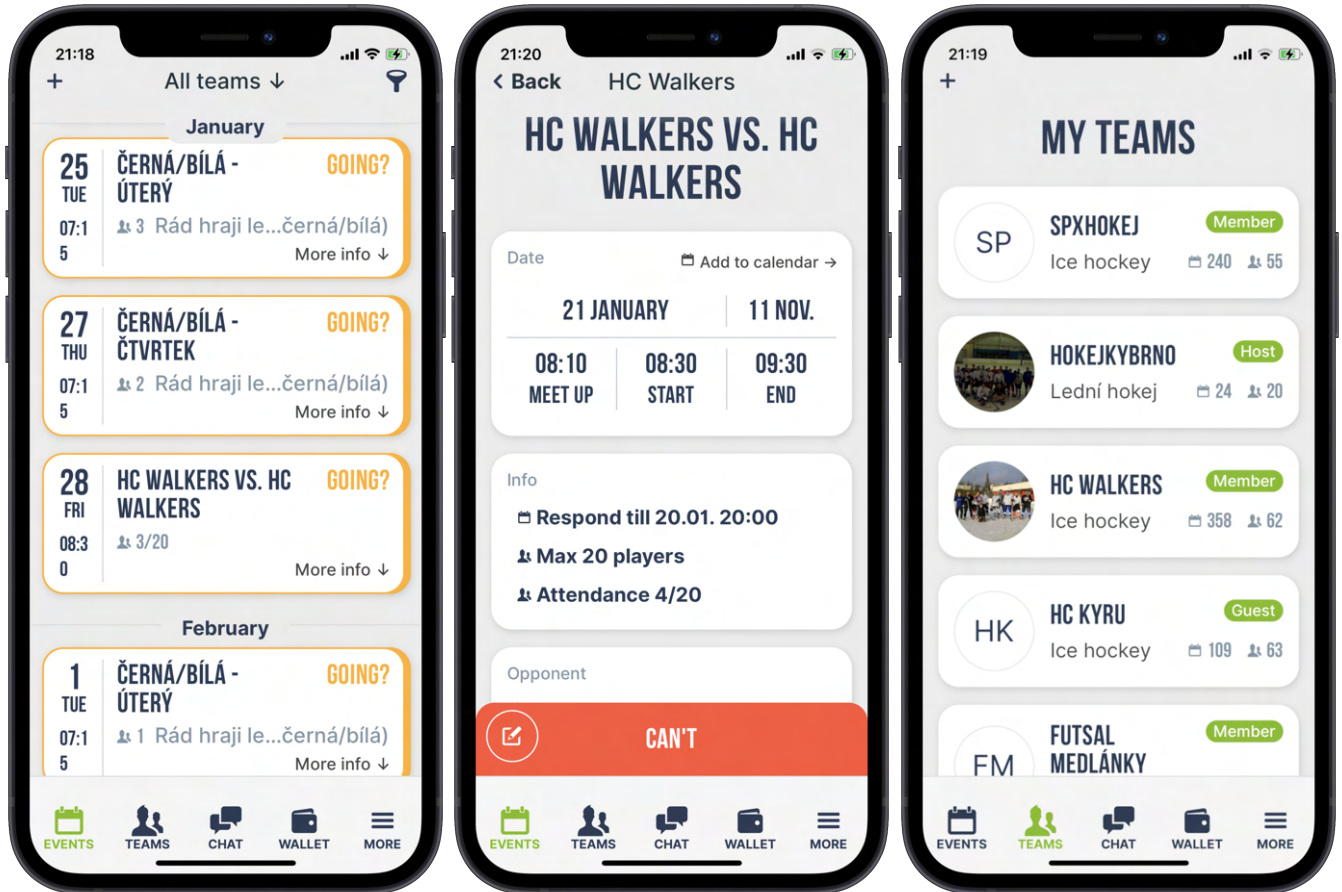
1. Without sufficient prior knowledge of the group on a personal level, a player is not able to recognize goaltenders in the list of attendees. Therefore, assuming no goaltender participation, he might decide to skip the game altogether.
2. Adding two extra places for goaltenders effectively removes the organizational convenience of an automatic participation limit. The organizer has to make sure that no other *player* signs up for the event as soon as player count reaches a desired level, while players themselves are technically still able to join.

Additionally, as already stated, the games are by definition private, i.e. visible to only members of the team. Putting private events (such as team training sessions) aside, this barrier might unnecessarily prevent the *demand for amateur hockey*³ from being realized. Equivalently, games are played with the sub-optimal amount of players, effectively preventing the market from reaching its equilibrium. In context of Týmuj.cz, this barrier can be overcome only by extending your team’s roster, i.e. by inviting every external player into the team by e-mail, even if he never joins your games again. Looking closely at Figure 6.1c, specifically HC Walkers and HC Kyru teams, this is exactly what has been happening in practice. Their roster has risen up to 62 and 63 players, while at least quarter of it will probably never play in this group again.

As already mentioned, it is very cumbersome and inefficient to manage goaltenders using the platform, Even more importantly, it does not help an organizer to *find* any. Analogically to the *players’ market*, there is a market for amateur goaltender’s services at a specific time

³Set of players willing to play amateur hockey at a given time and place.

and place. Increasingly often, its existence manifests itself in current trend of financial compensation for amateur goaltenders (see Section 2.1 for relevant discussion). This aspect of ice hockey games organization is not facilitated neither by Týmuj.cz nor by any other existing platform.



(a) All upcoming games

(b) Game detail

(c) All teams the user is member of

Figure 6.1: Týmuj.cz mobile application screen captures: List of all upcoming games for the logged in user, Game Detail displaying details of one specific games and lastly a list of all teams the user is member of.

Celebreak

As opposed to Týmuj.cz, a quick glance on the screenshots contained in Figure 6.3 reveals that CeleBreak completely ignores the idea of teams. It rather adopts the ad hoc approach of attending games, meeting the need of an amateur player to be time flexible, as discussed in Section 2.2. Match organization within the CeleBreak platform has the following properties:

- All the games are organized by a CeleBreak Match Host, so all the burden related to a game's organization lies on the CeleBreak team.

- Being the game organizer, CeleBreak is able to request an upfront payment for the participation, significantly increasing the probability of actual attendance of all the signed-in players. Player are able to cancel their attendance 15 hours before the game with 0% fee, automatically obtaining game credits available for them to spend in any other game. Later attendance cancellation automatically implies a 100% cancellation fee.
- A user is able to find a game based on soccer fields' location. This feature is especially convenient in larger cities where commutes to and from a game might be a decisive factor whether to actually attend or not. On a similar note, the Game Details screen provides a map with the location of the soccer field.

To motivate users to attend as many games as possible, CeleBreak also provides Games played, Streaks and Achievements overview as depicted on Figure 6.2. In this regard, Týmuž.cz only offers the participation percentage figure, calculated as a ratio of attended games and all the games organized within a single team.

From the perspective of the Platform developed within this work, especially CeleBreak's organization scheme is worth further analyzing. Organizing games by the Platform's support team would potentially generate two very important benefits described below.

Payment flow control. Requiring all players to pay for the match via the Platform would enable it to charge a small fee to cover operational expenses. Simultaneously, it would ensure that *all* players sign up for the event via this channel, requiring them to install and use the app in the first place.

Player skill level relevance. One of the selling points of the Platform is its capability to reflect the skill level of the players by advertising what kind of players are expected to join the game. Naturally, this is a very abstract and subjective concept. A player is expected during the registration to *subjectively* evaluate his skill on a scale one to six and is later expected to attend games that correspond to his/her skill level, while this number is *subjectively* preset by the game's organizer. Keeping the games organization in-house would gradually enable the Platform's team to *objectify* this criterion, at least to a limited extent, by adjusting the skill levels of players based on a physical presence at a game.

Game statistics gathering. Being able to gather more detailed statistics of the game, such as goals or goal assists, and consequently to evaluate players over long term based

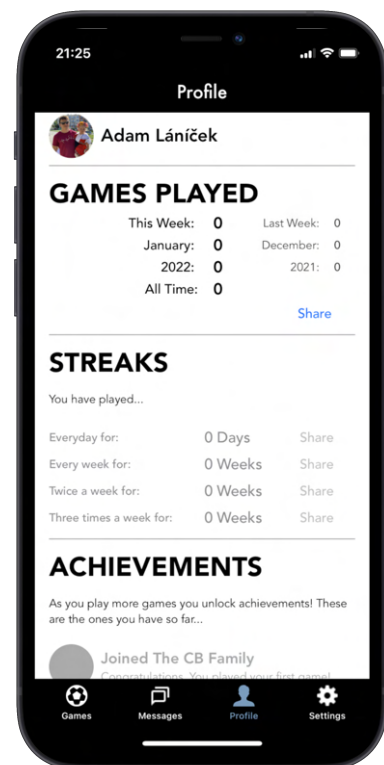
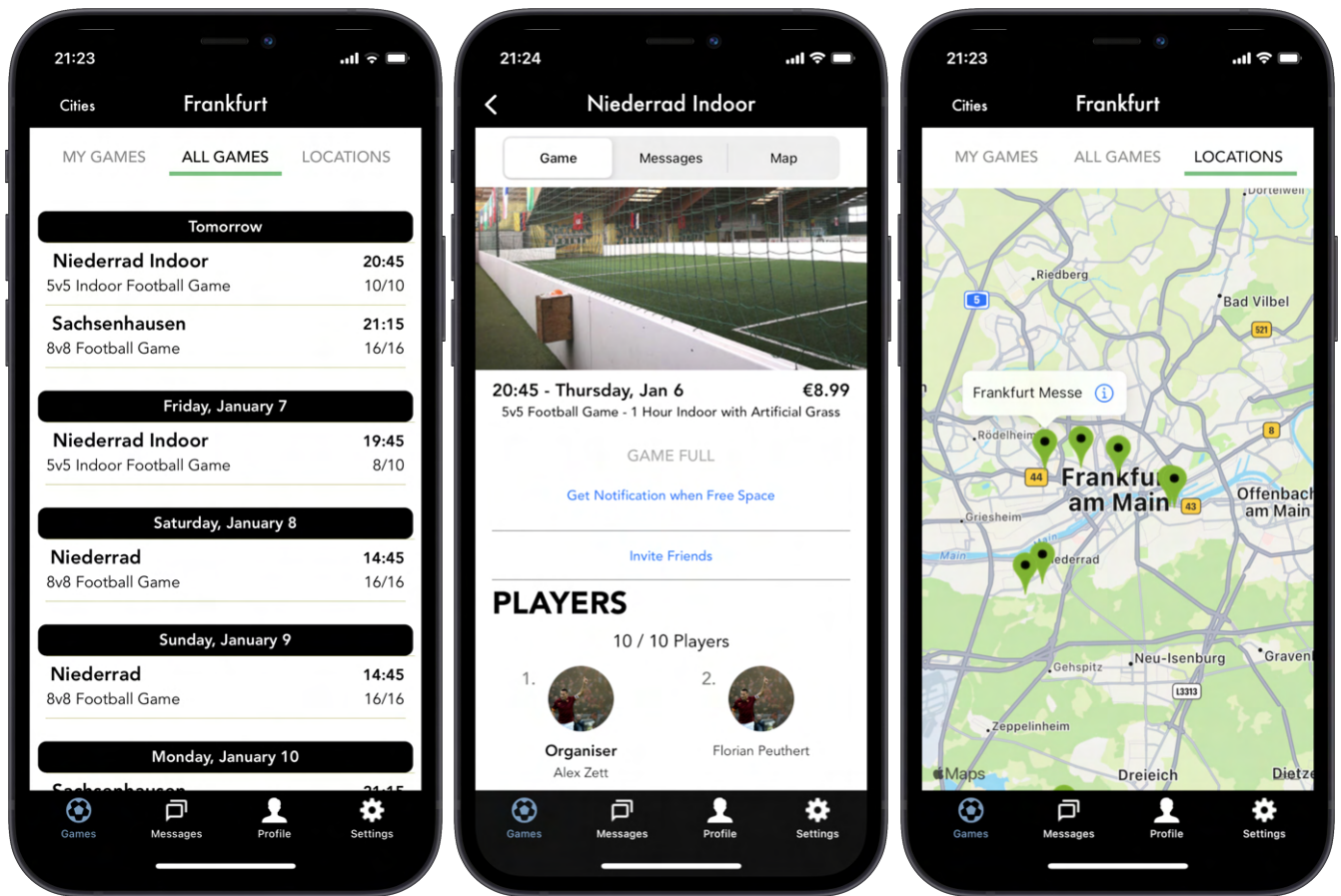


Figure 6.2: CeleBreak - user profile with the overview of played games a user's streaks and achievements.

on these figures, would be another significant selling point of the Platform. Providing this feature while leaving the statistics gathering task in the hands of the community would, however, render it completely irrelevant. Thus, a presence of Platform’s organizer is mandatory.

Despite all the benefits described above, there is, unfortunately, a very significant drawback related to this organization scheme. Naturally, organizing the games in-house would require one person to always be present at the games to facilitate its organization and gather the data, significantly raising its operational costs. Even more important are the implications for the Platform’s business model. With every new city added an additional Platform’s representative would be required, limiting Platform’s *scalability*, as the additional revenue coming from the new customers would be partially offset by the rise of the fixed costs related to this new employee.



(a) All games in the selected city

(b) Game detail

(c) Game search based on location

Figure 6.3: CeleBreak mobile application screen captures: A list of all soccer games organized by CeleBreak in the selected city. Game Detail of a specific game. Overview of all soccer games organized by CeleBreak in the selected city grouped by the specific location.

VašeLiga.cz

VašeLiga.cz is a subscription-based platform organizing virtual leagues in individual sports like badminton, squash and tennis. From a high level perspective, the platform's core idea is to provide a set of opponents from a pool of similarly-skilled players that a user plays against over the course of a month. Results of these matches are then entered into the platform and player's skill is reevaluated accordingly. Additionally, a part of the platform's business model is also organizing face to face tournaments. A workflow of a typical user of VašeLiga.cz looks like this:

1. User chooses a city where he wants to play and estimates his skill.
2. At the beginning of each month, the platform generates 4 different opponents the user should face within the upcoming 4 weeks.
3. With each of the assigned opponents, a match is arranged. Organizational details are discussed with the specific opponent and the court is paid for by the players themselves.
4. After the match, its results are entered into the platform.
5. At the end of each month, user moves into a pool of players with a higher or lower skill depending on his results.

Similarly to the Platform proposed within this work, an important feature of VašeLiga.cz application model is the athlete's skill level. Whereas capturing player's capabilities in team sports can be a real challenge, always suffering from a subjective bias, the „one on one“ nature of the sports supported by VašeLiga.cz enables it to base the skill characteristics on an objective measure - results of the matches.

6.2 Facilitating a market of goaltenders and referees

As thoroughly elaborated upon in Section 2.1, there is an ongoing imbalance between the supply and the demand for amateur ice hockey goaltenders. Despite being not as critical as goaltenders, the same can be stated about amateur referees, making the reasoning below partly applicable to them as well.

In simple terms, the free market economic theory states that an excess demand in a competitive market generates incentives to rise the price of a given scarce commodity, which eventually rises to the level where all the demand is satisfied and there is no excess supply, establishing a market equilibrium in the long term. When this concept is applied on the amateur ice hockey goaltender market, the supply is represented by a group of goaltenders and the demand by the games requiring their presence at a given time and place.

A long standing consensus, *not* a market-driven equilibrium, has been that goaltenders are compensated for their higher equipment costs by participating in the game free of charge. Empirical evidence, such as a discussion with at least 10 different hockey games organizers in Brno, suggested that each of them struggles to find goaltenders for their games, while some of them pay their amateur goaltender up to 500 CZK per game while the rest is willing to compensate an attending goaltender with lower hundreds of CZK.

This situation is a textbook example of an *underserved market*. What if, in the short term, were all goaltenders systematically incentivised to attend the games and other players

even motivated to accept a goaltender in the longer term? For this to happen as efficiently as possible, the market transparency is key and the Platform implemented within this work is a perfect opportunity to facilitate the transition towards the market equilibrium. Additionally, both organizers and goaltenders might have different time preference, naturally implying a potential existence of different equilibrium compensation levels for different time slots throughout the week – a challenge entirely solvable by the Platform by its very nature via openly advertising all games at a given time and place along with their planned goaltender/referee remuneration values.

6.3 Lazy registration

For an application like the Platform in question, the purpose of which is to provide a marketplace for users' interaction, each registered additional user generates a significant value. To convince a user to actually perform the sign up procedure, however, can be a challenge [33] – as as many as 54% of users abandon a website if the registration process is too cumbersome. On top of that, 86% of users dislike lengthy forms so much they make them quit registrations and almost 9 users out of 10 will simply enter incomplete or false information if the sign-up process is too exhaustive or intrusive.

So even if the registration process follows all the best practices, the application is likely to lose some of its potential users in the process. This is where the concept of *lazy registration* comes into place. At its core, it is an approach of letting the users to explore the application for themselves, engaging with it and seeing its benefits without requiring any sign-up effort. Duolingo, for example, the language educational platform, lets the user finish a whole language lesson before reminding the user to sign-up. And even then, instead of being compulsory, the completed registration just unlocks additional features to the user, such as achievement badges. Wroblewski [37] illustrates the impact of the lazy registration approach in practice, mentioning Twitter's sign-up process redesign in 2010 and the consequent soar of their conversion rate by 29%.

At the first design stage of the Platform's registration process, the sign-up process was bound to happen before any interaction with the actual application was allowed. The user was required:

- to enter his name, e-mail and the year of birth and
- to choose the roles in which he wants to participate in the games as well as estimate his skill for the player and goaltender roles, if applicable.

Apparently, this approach of having all the required data available *before* letting the user log in into the application significantly reduces application logic complexity. Additionally, these requirements might even not seem that demanding at a first glance, so a notion of a win-win situation can be established. This way, however, we are forcing user's cooperation without actually showing what he/she gets in return.

Learning a lesson mentioned above, it has been decided the Platform will apply the principles of *lazy registration*. After navigating to the Platform in a web browser, a user is offered the usual two options – either to log in as an already registered user or to register. After choosing registration, the user is prompted for an e-mail for its later verification and a password. Once this step is finished, the user has technically created an account, though only with a limited functionality. Redirected to the second step, displayed in Figure 6.4, the user is able to skip the profile finalization altogether and be redirected directly to the

Figure 6.4: Lazy registration implementation: a sign-up form in the second step of the registration process. User can skip it and continue browsing in the Platform with a limited functionality – the ability to attend a game is conditioned on having at least one attendance role assigned.

Games Overview page from where he/she can navigate all over the Platform. Without a finished registration, it is not possible to neither to attend a game, nor to set up a follow relationship to another athlete.

6.4 Game attendance management principles

This section is dedicated to the exploration of the game attendance management principles; specifically the proposed mechanism of attendance facilitation using a follow relationship will be introduced. Provided the user has assigned himself/herself the desired attendance roles, there are multiple ways available to sign up for a game:

1. To choose a game using the Attendance Shortcut Selector by just analyzing basic features of the game in the Games Overview.
2. To attend a game using the Attendance Selector in the Game Detail.
3. To attend a recommended game in a manner similar to the two described above.
4. To be added to a game by an organizer automatically, provided the user follows him in the *opt-out* mode

As already discussed in Chapter 2, given the high requirements of amateur ice hockey on the player head count, it is, based on author's experience, as well as on the experience of 9 other seasoned organizers, a real challenge to build a group of people large enough to ensure that every game is properly manned. Hence the need for a Platform to mediate

the missing ad hoc players. That said, there is also another subset of players, representing approximately a third of the expected attendance count, which is present at 9 games out of 10. If the Platform facilitates the game attendance management for the ad hoc players, as important as they are for the game, the regular dedicated players should perhaps not be forgotten about, either. Arguably, there should be a way to simplify the attendance procedure for them and the organizer himself as well.

Moreover, the importance of the social aspect of a game cannot be underestimated, either. When choosing between multiple otherwise similar games, a player will always choose by a large margin the one attended by the most people he knows or who he enjoys to play with the most. The solutions for both of these recently introduced challenges are presented in the two remaining subsections in this chapter.

6.4.1 Follow mechanism

Generally, there are two different ways to initiate a relationship with another user within platforms with a social element, both of which express the wish to consume the content or activities of the counterpart. As summarized by Yu-Hao and Chien [24], this tie can be either:

- a *symmetric* one when a person sends a request to another person, and that individual needs to accept the request for mutual information disclosure, often referred to as *friendship* or
- an *asymmetric* one in case a user *follows* another person without requiring the other party's approval.

To keep the application logic from the perspective of the user as straightforward as possible, the second variant has been selected for the Platform's purposes. The use case for the follow relationship has already been discussed in Section – in the Athlete Overview, a user can initiate a follow relationship with a person he/she wants to play with or whose games he/she wants to attend. Having initiated the relationship, a user can see the list of the followed athletes in the User Profile and potentially unfollow these⁴. The follow relationship setup directly influences the Dashboard page, specifically the Might Interest You component as its purpose is to search for the games with the biggest amount of the followed athletes and to render them with the most interesting games at the top.

6.4.2 Opt-in vs opt-out

The Platform's follow mode described in the previous section is further divided into the two modes, determining the strength of the follow relationship, both of which influence solely the possibility of an automatic sign up for a game:

- the default, *opt-in* mode is just a standard follow relationship with no further implications, and
- the *opt-out* mode encompasses a standard follow relationship with a transfer of the game sign up rights towards the followed athlete.

Following in the *opt-out* mode, i.e. letting the trusted organizer to sign an athlete for a game without any further action required, is a feature aimed at organizers and those

⁴Unfollow action can be performed in the Athlete Overview as well

dedicated regular players discussed above. The occurrence of the activated *opt-out* mode is going to be rather scarce among the set of followed athletes – letting an organizer to perform an automatic sign up on user’s behalf is a sign of a substantial levels of both mutual trust and user’s resolve to attend the vast majority of the games organized by that specific organizer. In the end, by signing an athlete automatically for a game, an organizer is exposed to a risk of athlete’s ignorance, resulting in an absence without cancelling the attendance in advance. For this reasons, every time a new follow relationship is established, *opt-in* mode is the default one. Activating *opt-out* mode is a matter of clicking a single toggle switch either in Athlete Overview or in the User Profile page. From an application business logic perspective, this action results in the availability of the follower in the athlete search results for a given organizer in the New Game Attendance form (as depicted in Figure 4.8).

Chapter 7

Implementation

The aim of Chapter 5 was to lay the technological foundations for the Platform implemented within this work. Its content was structured according to its main building blocks – a REST API server, a web application implemented in React TypeScript framework and a cross-platform mobile application leveraging the React Native framework. The same structure is going to prevail in this Chapter, describing the actual implementation of the three mentioned pillars.

Because of the expected user base structure, reflected by the Personas defined in Chapter 3 and given the amount of effort required by the design phase, the focus of the implementation phase has been on the web application’s core features implementation, supported by the development of all the API business logic necessary. That said, a groundwork for the development of the cross-platform mobile application was laid nonetheless, as explained in Section 7.3.

7.1 REST API server

The Flask microframework written in the Python programming language, whose properties and features were thoroughly discussed in Section 5.2, has been selected for the implementation of the REST API server. The following subsections are dedicated to selected aspects of its implementation, namely:

- API server architecture and its main features
- User authentication process necessary for restricting the requests only for authorized users
- API endpoints available for client front end applications

7.1.1 Architecture and main features

From the technological perspective, the API server is built upon the following pillars

- Flask microframework facilitating the client requests processing via its WSGI engine introduced in Section 5.2.
- Model-View-Controller (MVC) architecture, the favourable implications of which encompass a modular code structure and a straightforward extensibility, maintenance and testing.

- Object-relational mapping of entities using the SQLAlchemy Python library.
- A data seeding script exploiting the SQLAlchemy's API to seed the database in a way which mirrors the real life reality as closely as possible.
- Docker and Amazon Elastic Container Service (ECS) integration ensuring fast automatic redeployment.

Model-View-Controller

As its name suggests, the MVC is an architectural pattern dividing the application into three main logical components: the *model*, the *view* and the *controller*. Contrary to other design patterns, however, as noted by Bucanek [1], the MVC is less clearly and strictly defined than many other patterns, leaving a lot of latitude for alternate implementations, basically being more a philosophy than a recipe. The server API discussed within this work adopts the MVC philosophy. Manifestations of the MVC components in its architecture can be found represented as files in the file tree structure depicted in Figure 7.1. Using the MVC terminology, they can be described as follows:

- **Model** as layer holding raw data and providing the APIs to query the underlying MySQL database engine, represented by the `app/core/model/models.py` file further discussed in the next section.
- **View** as layer processing the incoming and outgoing HTTP requests, representing the actual *resources* and implementing their corresponding API *endpoints*. Their implementation can be found in the `app/resources` directory.
- **Controller** as a middle layer containing the business logic for calculating the content to be wrapped in the outgoing HTTP request by the *View* layer based on the input also provided by the *View* layer. The corresponding implementations can be found in the `app/core/models/*_handler.py` files.

Used entities and their ORM representation

Entities used within the application logic are encapsulated in the corresponding model classes defined as children of the base SQLAlchemy class that provides all the APIs required for querying the MySQL database engine. A comprehensive list of semantically described models located in `app/core/model/models.py` with their corresponding database table names available in Figure 7.2 follows:

- **GameModel** (*game*) represents a game entity, contains all the relevant attributes such as the start time and the expected skill as well as the 1:1 relationship to *IceRinkModel* referencing a game location.
- **AthleteModel** (*athlete*): represents an athlete entity with all its important attributes
- **AnonymousAthleteModel** (*athlete_anonymous*) represents an anonymous athlete entity added within a game, referencing *GameModel* as the game for which the athlete has been added and the *AthleteModel* as the id of the athlete who added him/her.
- **IceRinkModel** (*icerink*) represents ice rink entity and stores its properties, most importantly its rent price.

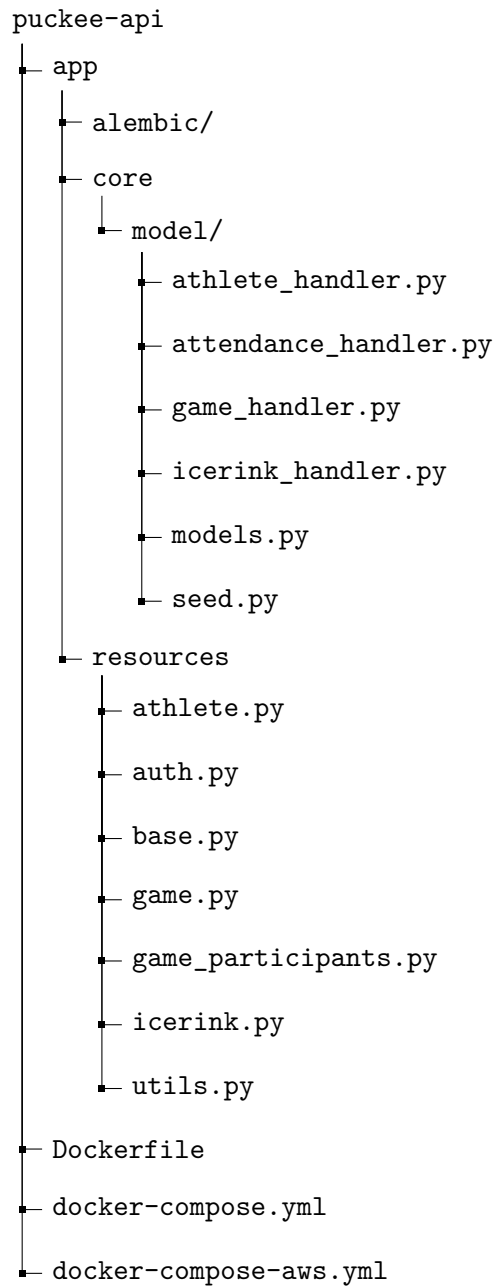


Figure 7.1: Highlighting the core components of the Platform’s API server directory structure, reflecting the Model-View-Controller architecture. Resources directory contains the implementation of the API endpoints handling (*View*), while `*_handler.py` files implement the actual business logic (*Controller*) and are called from the *Views*. The file `models.py` contains the ORM representation of the application entities (*Model*). The `seed.py` file utilizes the *Models* to fill the database with the relevant mock data. `Dockerfile` defines the API Docker image and `docker-compose.yml` files ensure the connection of the API container to the underlying MySQL database Docker container, with the `docker-compose-aws.yml` containing additional settings required when deploying the container to Amazon Elastic Container Service.

- **AthleteRoleModel** (*athlete_role*) represents athlete role entity, the corresponding table is filled with values user, player, goaltender and referee.

To be able to capture the M:N relations among the entities, as for example required in case of players/goaltenders/referees attending a game, or the 1:N relations as needed for athlete roles, SQLAlchemy's *relationships* were introduced to the corresponding models. These relationships are based on the *association* tables generated using the SQLAlchemy Core APIs, i.e. using a thin wrappers around the traditional SQL syntax instead of leveraging the ORM feature.

This raw *association* table approach, however, does not work as required by the ORM engine when an additional attribute has to be stored along with the mapping. For example, looking at the *followers* table in Figure 7.2, it contains two foreign keys referencing the *athlete* table determining who follows whom as well as an additional attribute determining the type of the follow relationship. Using the SQLAlchemy Core approach, it is not possible to later access the attribute using the standard ORM queries. Therefore, in this specific case, an *association model* instead of table had to be used to set up the follow relationship.

Mock data generation

The development of a data-intensive application such as the Platform requires large sets of data to be efficient. Without a continuous corner cases testing, exceptions are being thrown as late as in the user testing phase, instead of much earlier during the development phase on a developer's machine as the code is being written.

Bearing in mind the importance of having a large dataset at the developer's disposal, a significant effort was put into developing a data seeding script (stored in the file `app/core/model/seed.py` as shown in Figure 7.1). There are multiple parameters related to the distribution of the athlete roles and the game participants counts, which influence the database seeding process – a comprehensive list of all those is listed in Table 7.1. As it currently stands, this script is run with every database initialization, i.e. as part of the first Docker Compose launch, initializing the API and MySQL database containers.

7.1.2 User authentication

A vast majority of the Platform's API endpoints deal with the processing of user actions or providing data tailored for a given user. Therefore, a mechanism to verify the user's identity with every request is necessary. The authentication process is facilitated using the *JSON Web Token (JWT)* – an open standard defining a compact and self-contained way for securely transmitting information between parties as a JSON object (RFC 7519 [22]). From the technological perspective, the actual authentication process, facilitated by the Flask-JWT-Extended Python module, is as follows:

1. The Flask application's *secret key* needs to be defined since the created JWTs are signed with it.
2. Once a user is authorized, the success HTTP response contains an access token generated by the `create_access_token()` method provided by the Flask-JWT-Extended module.
3. All the protected endpoints, i.e. `get()`, `post()`, `put()` and `update()` methods implemented in resource files located in `app/resources`, are *decorated* using the

Parameter	Value
Games total count	300
Athlete total count	400
Anonymous athlete total count	50
Player role occurrence	Randomly selected 2/3 of athletes
Goaltender role occurrence period	5
Referee role occurrence period	13
Count of followed players (min - max)	3 - 20
Opt-out mode occurrence probability in a given follow relationship	10%
Game participant counts ranges	Minimum - maximum value
Organizers	1 - 2
Registered players	7 - 16
Unregistered players	1 - 4
Goaltenders	0 - 1
Unregistered goaltenders	0 - 1
Referees	0 - 1
Unregistered referees	0 - 1

Table 7.1: Data seeding script parameters: Two out of three athletes are players, every fifth athlete is a goaltender, every 13th athlete is a referee. Every athlete follows 3 - 20 other athletes, while 10% of those are followed in the *opt-out mode*. There are 14-20 players attending a game on average, with one attending goaltender and one referee.

`jwt_required()` decorator which checks every incoming request for a presence of the corresponding authorization header in the format:

```
Authorization: Bearer <access_token>
```

7.1.3 Available API Endpoints

There are 12 endpoints available within the server API, each of which has implemented at least one of the methods corresponding to the traditional HTTP methods – the available are the following: `get()`, `post()`, `put()` and `delete()`. The purpose of this section is to describe their specific usage within the Platform’s context.

Athlete-related endpoints

1. `/athlete`: An endpoint serving the Athlete Overview by providing a *paginated*¹ list of athletes via its corresponding `get()` method, requiring `page_id` and `per_page` GET request parameters.
2. `/athlete/search`: With only the `get()` method available as well, it returns a list of Athletes searched by name. The required GET parameters are these:
 - `name`: athlete name substring of an arbitrary length
 - `role_id`: restrict the search only to athletes with a given role
 - `requesting_id`: identification of the athlete performing the search for the follow status determination

Additionally, two optional parameters can be passed as well:

- `followers_only`: restrict the search results to only the followers of the `requesting_id` athlete.
 - `opt_out_mode`: restrict the search results to only the followers following the `requesting_id` in the *opt-out* mode.
3. `/athlete/<follower_id>/follow/<followee_id>`: An endpoint for setup, modification and cancellation of the follow relationship from `follower_id` towards `followee_id`, with the corresponding `post()`, `put()` and `delete()` methods implemented. `post()` method requires boolean parameter `opt_out_mode` present in request body.

Game-related endpoints

1. `/game`: Analogically to the `/athlete` endpoint, this endpoint serves the Games Overview by providing a *paginated* list of games. Its `get()` method also requires `page_id` and `per_page` request parameters. On top of the methods available at the `/athlete` endpoint, this one implements also a `post()` method for game creation, requiring the presence of a JSON object describing the created game in the request body.
2. `/game/date`: Implements only the `get()` method to fetch all the games taking place between specified dates. Requires `start_date` and `end_date` parameters.

¹As discussed in more detail in Section 7.2

3. `/game/user/<athlete_id>`: An endpoint implementing the `get()` method, used in the My Upcoming Games component, listing all the games attended by the athlete with a given `athlete_id`. The length of the returned list can be limited by an optional parameter `game_limit`.
4. `/game/user/<athlete_id>/followers`: Analogically to the previous one, this endpoint also implements only the `get()` method for the purposes of the Might Interest You Dashboard component.
5. `/game/<game_id>`: An endpoint for game modification, update and deletion implementing the corresponding methods.

Game participants endpoints

1. `/game/<game_id>/participants`: An endpoint whose:
 - `get()` method returns all registered as well as unregistered participants of a game identified by the `game_id`. Requires the `requesting_id` parameter referencing the requesting athlete for determining the follow status of all the returned athletes.
 - `post()` method adds a registered or unregistered participant in a specific role to a game based on the `athlete_id` or the `athlete_name` parameters, respectively, present in the request body. The `athlete_role` request body parameter is also expected.
 - `delete()` method removes a registered or unregistered athlete from a game based on the corresponding GET parameters `athlete_id` or `athlete_name`.
2. `/game/<game_id>/organizers/<athlete_id>`: An endpoint for obtaining, adding and removing game organizer identified by the `athlete_id` GET parameter. Corresponding `post()` method requires `game_id` and `athlete_id` parameters, while `get()` and `delete()` methods require only the `game_id` parameter.

Ice rink endpoints

1. `/icerink`: An endpoint whose `get()` method returns a list of all the ice rinks available in the database including their properties.
2. `/icerink/<rink_id>`: An endpoint for fetching only one specific ice rink referenced by the `rink_id` GET parameter.

7.2 React web application front end

Having described the implementation of the REST API server, encapsulating all the business logic and providing data for the Platform's user-facing applications, the following content will take a deep dive into Platform's front end, namely the web application implemented using the React TypeScript library. At the start of the implementation phase, the Platform has been assigned a name – Puckee. Therefore, from this point onward, it will be referred to using this name.

7.2.1 Architecture

Starting from the very top level, as depicted in Listing 7.1, the application is created by the React *functional component* called Puckee, which returns a JSX (recall Section 5.3.1 discussing React functional components). The parent element of this JSX is the `div` element of the class `App`, following the usual naming convention in React applications. What follows is a React component tree where each wrapper component has a specific purpose:

- **NotificationsProvider:** Creates a React *context* for notifications generation and removal, to provide users feedback on their actions. This context is accessible in all the child components throughout the application after calling the `useNotifications()` *hook*².
- **AuthProvider:** Creates a React *context* for the user authentication. This encompasses information on the authentication status as well as the details of a logged in user. The context is accessible in all the child components calling the `useAuth()` hook.
- **Routes:** Provided by the *react-router-dom* library for web routing, `Routes` and `Route` are the primary ways of rendering a component based on the current URL location. After a location change, `Routes` component iterates over all its child `Route` elements to find the best match and renders the corresponding user interface branch.
- **Route:** Defines a `path` for which the specific React component will be rendered or, alternatively, serves as a wrapper of a `Route` placed lower in the tree. In Puckee's instance, the wrapper feature is used to wrap the corresponding components in a layout that is shared among multiple components. This wrapper `Route` renders a `Outlet` which in turn renders the path-matching lower-level child component. For example, `AuthLayout` component defines a layout specific for the login and sign-up pages. Within this layout, the specific child components, such as the `SignInForm` or the `SignUpForm`, are rendered based on the URL entered in a browser – `/sign-in` for the former, `/sign-up` for the latter.
- **RequireAuth:** A wrapper leveraging the `AuthProvider` context to enforce the authenticated status of a user who is trying to access a given path.

7.2.2 Sharing the front end code

As mentioned in the preceding text, the long term aim is to provide two fully-featured clients for Puckee. Starting from the web client written in React TypeScript, it is a sensible choice to use React Native as a platform for the mobile client. Although developing React Native applications requires adopting a different paradigm in selected areas, such as navigation, the overlaps are significant, saving a lot of development effort by reusing the learned knowledge. Luckily, the knowledge is not the only thing that can be reused. The architecture of both React and React Native applications is composed of many layers, the majority of which can be shared. Examples of a shareable code might be (but are not limited to) the `NotificationsProvider` and `AuthProvider` components introduced in the previous subsection.

To ensure the shareability of the code, the *monorepo* approach towards repository management had to be adopted. A *monorepo* is a version-controlled repository holding multiple

²For more details refer to <https://reactjs.org/docs/hooks-intro.html>

```

import React from 'react'           1
import ...                             2
                                     3
export default function Puckee() {     4
  return (                             5
    <div className="App">              6
      <NotificationsProvider>         7
        <AuthProvider>               8
          <Routes>                    9
            <Route element={}<AuthLayout />> 10
              <Route path="sign-in" element={}<SignInForm/>> /> 11
              <Route path="sign-up" element={}<SignUpForm/>> /> 12
              <Route path="sign-up-details" 13
                element={}<RequireAuth><SignUpDetailsForm/></RequireAuth>> /> 14
              <Route path="*" element={}<NoMatch />> /> 15
            </Route>                  16
            <Route element={}<DashboardLayout />> ... </Route> 17
            <Route element={}<StdLayout />> ... </Route> 18
            <Route element={}<SearchLayout />> ... </Route> 19
          </Routes>                  20
        </AuthProvider>              21
      </Notifications />             22
    </NotificationsProvider>         23
  </div>                             24
)                                       25
}                                       26

```

Listing 7.1: High-level component structure of Puckee web application, defined in `apps/web/Puckee.tsx`. `NotificationsProvider` providing context for generating user notifications, `AuthProvider` providing authentication context. `Routes` and `Route` from the `react-router-dom` library for matching URL paths to the corresponding components to be rendered. `RequireAuth` wrapper enforcing user's authenticated status for its child components.

related, often logically independent, projects³. From a high level perspective, Puckee's monorepo is divided into the three parts, all of which are identifiable in Figure 7.3:

- `apps/web` containing the codebase of the React web application.
- `apps/mob` where React Native mobile application's source code is located.
- `packages/puckee-common` containing the modules shared between the web and mobile clients.

While simply keeping the code in the same repository is a prerequisite when striving for module shareability, it does not make the modules reusable from the application engine's perspective. This aspect is covered by *Yarn workspaces*. *Yarn* is a package manager developed by Facebook, Inc. (Meta, Inc.) and *workspaces* are one of its features, leveraging the *monorepo* approach to make the mutual cross-referencing possible. In Puckee's case, the root-level `package.json` configuration file defines the two workspaces as follows:

```
{
  "name": "puckee",
  ...
  "workspaces": [
    "apps/*",
    "packages/*"
  ]
}
```

This setting makes every subdirectory of `apps` and `packages` to be a separate workspace while ensuring that all the modules contained can be cross-referenced. For example, importing a model class to a web application module from the `puckee-common` package is as straightforward as:

```
import { Athlete } from 'puckee-common/types'
```

As analyzed in Section 5.3.2, the target of the React rendering engine is the DOM of the webpage, whereas in case of React Native it is the native mobile view elements. In the code, this is exhibited by a different content of the returned JSX values, where for example `div` elements are substituted by their `View` counterparts. Also, inherently different navigational paradigms of web and mobile applications require different handling of this application feature in both platforms. Apart from that, in an ideal situation, code dealing with the remaining aspects of an application can theoretically be shared. To achieve this level of reusability, however, it is necessary to be familiar with the more advanced React design patterns which were not studied as part of this work – making maximum possible code reuse definitely a goal for future development. As it currently stands, the shared `puckee-common` package contains the following directories:

- `api` containing `athlete.tsx`, `game.tsx`, `icerink.tsx` modules with all the required API calls.
- `auth` defining the authentication context encapsulated into the `AuthProvider` and `RequireAuth` components and the `useAuth()` *hook* described in the previous subsection.

³As a side note, React library itself is a prime example of *monorepo* repository

- `context` introducing the notifications context implemented in the `Notifications-Provider` component and the corresponding `useNotifications()` *hook* also mentioned in the previous subsection.
- `types` defining the model classes representing the key application entities - `Anonym-Athlete`, `Athlete`, `Game` and `IceRink`.

7.2.3 Asynchronous client-server communication

Asynchronous communication between a web client and a server poses multiple challenges, such as data fetching, caching and server state update and its synchronization. Out of the box, In React environment, these functionalities have been traditionally implemented using various combinations of React core features, such as the *state* and the `useEffect()` *hook* or via using general purpose *client* state management libraries, such as `React Redux`, to handle and provide asynchronous data throughout the application. However, as shown by Linsley [25], *server* and *client* states are inherently different, rendering the usage of those state management libraries inappropriate. Namely, *server* state:

- requires asynchronous APIs for fetching and updating,
- implies shared ownership, and finally
- potentially becomes outdated in the application when handled improperly.

Based on author's and his friends' frustrating experience with other event management platforms which suffered from improper handling of *server* vs. *client* state matters, resulting in front end clients displaying clearly outdated data, it was evident that the importance of its correct management cannot be overstated. In the beginning, Puckee web application set out to use `React Redux` in a manner similar as described above. Later, its usage was abandoned completely in favour of the `React Query` library, specialized in working with the *server* state. It is used for asynchronous communication throughout the whole application. In principle, its usage can be summarized as follows:

1. Data is fetched using the `useQuery()` *hook* which accepts, among others, these arguments:
 - **Asynchronous function** to perform the API request.
 - Structured **query key** as a unique identification of the query.
 - Additional parameters, such as a boolean to determine if a given query is active or not.
2. Along with every relevant user action, there is a *mutation* defined using the `useMutation()` *hook*, which creates a request (POST, PUT or DELETE) to be sent and prepares a set of corresponding actions if the request:
 - **successes**: *Invalidate* all the queries affected by this change by enumerating their keys. These will be *refetched lazily*, i.e. once a component referring to them is re-rendered.
 - **fails**: Report a server API error and perform no further action.

While this scheme captures a majority of **React Query** use cases within the Puckee web application, Games and Athlete Overview pages with their theoretically infinite lists of items require different approach⁴. For this purpose, **React Query** provides the `useInfiniteQuery()` *hook* implementing a *pagination* approach towards data fetching. In addition to the parameters provided to the standard `useQuery()`, `useInfiniteQuery()` requires methods to get the identifications of the next/previous pages to be requested, so function callbacks facilitating this have to be provided as well. In essence, the asynchronous communication works as follows:

1. `useInfiniteQuery()` sends a request to the `game` API endpoint, providing the GET parameters `page_id`, identifying the page and `per_page` parameter, indicating the desired length of the list to be returned.
2. The server returns a JSON response in the format:

```
{'next_id': 2, 'previous_id': null, 'data': [...]}
```

3. `useInfiniteQuery()` processes the response data list and stores the `next_id` and `previous_id` values, providing the next and previous page numbers, respectively.
4. If a next page is required based on user's scroll effort, a new API request is sent with the value of `page_id = next_id`. If the previous page is required in the Games Overview based on the corresponding button click and if it actually exists, as ensured by `previous_id != null` condition, a request is sent with `page_id = previous_id`.

7.2.4 Web client layout definition

Apart from the indispensable HTML code and a limited use of the plain vanilla Cascading Style Sheets (CSS), the design of the web application as presented in the next subsection is based on the following four pillars, each of which is described in the following paragraphs:

- **Bootstrap**
- **CSS Grid Layout**
- **CSS Flexbox Layout**

Bootstrap

Originating at Twitter in 2011, Bootstrap is an open-source CSS framework, containing HTML, CSS and JavaScript-based design templates for all sorts of web interface components. It contains predefined CSS classes for styling as well as for web page layout structuring, leveraged in the Puckee web client mainly to prevent reinventing the wheel and to avoid repetitive CSS classes definition. Additionally, as most of the data content within the application is positioned using the Flexbox Layout CSS feature, Bootstrap classes implementing its properties were used extensively.

⁴Actually, in case of Games Overview the list is *bidirectionally infinite* – a user starts from the present day and browses through games both in the future and to the past.

```

<div className="d-flex flex-column justify-content-between"> 1
  <div className="d-flex flex-row justify-content-center flex-1"> 2
    <div>A</div> 3
    <div>B</div> 4
  </div> 5
  <div className="d-flex flex-row justify-content-center flex-7"> 6
    <div>C</div> 7
    <div>D</div> 8
  </div> 9
</div> 10

```

Listing 7.2: Frequently occurring Flexbox layout usage pattern in Puckee web application. Starting from the top level, the parent `div` defines a vertically-oriented Flexbox with its direct child `div` maximizing space that separates them along the vertical axis. Second-level `div` elements define horizontally-oriented Flexbox with their direct child `div` elements centered along the horizontal axis.

CSS Grid Layout

CSS Grid Layout is a two-dimensional grid-based layout system. In the Puckee web application, Grid Layout was used for the high-level partition of page to define the *grid-areas* where the horizontal header, the vertical menubar and the page content box are positioned⁵. The resulting layout can be seen in Section 7.2.5.

CSS Flexbox Layout

Another CSS layout system that can be used to easily define a web page layout is the CSS Flexbox. Given its one-dimensional nature, it is not as powerful as the Grid. In practice, however, a significant portion of positioning problems do not require the management of the second dimension. Therefore, it is reasonable to trade off a theoretically lower layout definition capabilities for the Flexbox's flexibility and its straightforward use, which is why in the Puckee web application it was used to position all the content within all the *grid-areas*, from a basic layout setup to a detailed positioning of displayed icons. Given its ubiquity within the Platform, an illustrative example of a frequently-used pattern using the Bootstrap's implementation of Flexbox layout can be found in Listing 7.2 with a detailed description.

7.2.5 Application tour

With the long term goal of creating the Platform's web interface featuring all the functionality described in Section 4.3, the focus of the web application implementation phase within this work was to create the working prototypes of the Games Overview, Athlete Overview and Dashboard pages. The following text contains a descriptive summary of the implemented pages and a description of selected aspects of their implementation. Web application screen captures of the discussed pages are available in Appendix B.

⁵For the specific definition of the Grid used please refer to `apps/web/index.css`

Games Overview

The key feature of the Games Overview page whose layout is captured in Figure B.1 is the `useInfiniteQuery()` hook discussed in Section 7.2.3, facilitating the smooth loading of the *bidirectional infinite list* of games. To be able to manage broad palette of games and simultaneously to enable a user to create its tailored version of this page for himself/herself, a plan into the future is to implement a complex set of filters with a possibility to save them for a later use. At the time of finishing this thesis, the page enables a user to do the following:

- attend/leave games in various roles using the Attendance Shortcut Selector – examples of some of its possible states are depicted in Figure B.2,
- open a Game Detail page of a specific game,
- create a new game, i.e. navigate to the New Game page, and, finally,
- modify a game, provided the logged in user is present in the list of game’s organizers

Athlete Overview

Analogically to the Games Overview page, the core of this page is the *infinite list* of Players, though this time it is only unidirectional – no sufficiently relevant time criterion has been identified to justify the need for the time dimension. As depicted in Figure B.3, the Athlete Overview page covers the following use cases:

- Finding an arbitrary athlete to initiate the follow relationship, for example based on a favourable game experience in past games.
- Finding the athlete’s skill rating in his various attendance roles.

Additionally, these features are planned for the future:

- A complex set of filters - by name, attendance roles, follow status, preferred playing location and skill rating.
- As indicated in the previously presented Athlete Overview mock-ups, played games statistics over the previous month, quarter, and a year are to be added.

Dashboard

The purpose of the Puckee Dashboard is to extract the most relevant game data to the user and present it in a cohesive manner. Specifically, as depicted in Figure B.4, Puckee’s Dashboard is divided in the three parts:

- **My Upcoming Games** section always reports three nearest games of the currently logged in current user. Compared to the Games Overview page and the Might Interest You component, games in this list are rendered slightly differently, with the Game Attendance Shortcut Selector displaying also the financial implications to the user.
- **Games Calendar** section aims to stress the planning dimension of the user’s games. At the time of this thesis submission, the Games Calendar user interface has not been connected to the server back end yet.
- **Might Interest You** shows a maximum of 10 games *not* currently attended by the logged in user and attended by followed athletes and/or organizers, sorted by the count of the participating followed people.

Game detail

Game Detail page is divided in the three parts, one of which (Game Chat) has not been implemented yet, as depicted in Figure B.5. In general, it aims to convey the following information:

- **Basic Game Data** – when and where does the game take place, who is organizing it, what is the target count of players, goaltenders and referees.
- **Game Attendance** – what players, goaltenders and referees are attending the game. Highlight the followed ones to make the game more attractive and show what additional unregistered participants were added by the organizers.
- **Game Chat** – is there any game specific information that needs to be discussed?

New game

The New Game page, depicted in Figure B.6, offers the organizer an interface to create a game and, if sensible for his/her use case, to directly *add* athletes in different roles to a game, provided that they follow him in the *opt-out* mode. From the React component tree perspective, the page content is represented by the `GameAdminForm` component which has the following children:

- Three instances of the `GameAdminParticipants` component for the Player, Goal-tender and Referee roles. This component encapsulates the athlete search bar and updates the `GameAdminForm state` with the relevant participants added in their specific roles, ensuring their addition to the game right at the moment it is created on the server.
- Three instances of three different helper components, discussed in Section 4.3.4, all of which will be subject to future development:
 - `GameAdminAvailableGroups` suggesting groups of followers to be added,
 - `GameAdminFinancialEstimate` providing a basic game budgeting overview, and,
 - `GameAdminRecentlyOrganized` serving as templates for auto-filling of basic game properties.

7.3 React Native mobile application front end

Even though the mobile client implementation was of lower priority, author's enrollment to the Application Development for Mobile Devices course at FIT BUT seemed to present a good opportunity to implement a small feature subset of Puckee's mobile application client, despite being limited by the length of one semester. This constraint proved to be unfavourable, as the development had to proceed even without a comprehensive definition of the Platform concept, which was still in the definition phase, continuously altered based on discussions with various stakeholders to maximize the value it brings to the market.

Because of the above, it was becoming increasingly evident that the mobile application was going to be rendered partially obsolete by the final version of the application concept. That ended up being the case, indeed. That is not to say, though, that in the big picture this endeavour did not bring any benefits – quite the opposite. Apart from the more tangible benefits, such as having a basic application structure along with the more advanced

navigational features already laid out for the future, this parallel implementation was constantly reminding the author about the necessity to make some of the modules *shareable* to avoid code duplication and the related unproductive double maintenance effort. This led to a research of potential solutions, resulted in adopting the *monorepo* approach for the whole application front end, essentially influencing the path forward for the mobile client, which was determined as follows:

1. Incorporation of mobile client's code into the Platform's *monorepo*.
2. Mobile client's architecture refactoring to reflect the newly introduced `puckee-common` package, introducing the shared modules, as discussed in Section 7.2.2.
3. A temporary freeze of mobile client development in favour of the web application.

For the sake of completeness, Appendix C contains the screen captures of some of the mobile screens implemented, matching in principle the mobile mock-ups introduced in Section 4.2: Games Overview, Game Detail: Basic Info and Game Detail: Attendance.

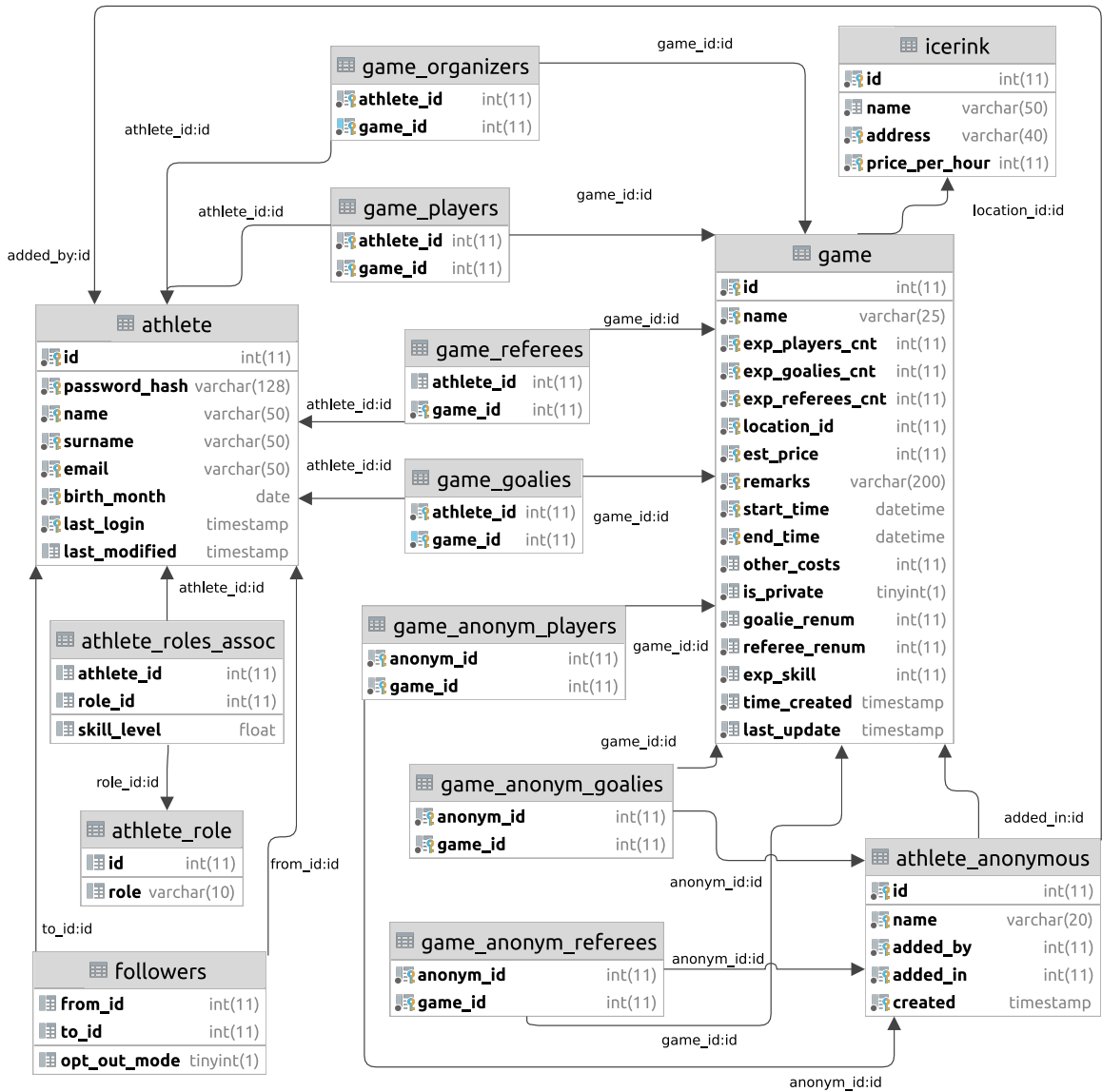


Figure 7.2: Server API database diagram showing the schema structure generated by the SQLAlchemy engine based on ORM representation of the entities and their relationships defined in app/core/model/models.py.

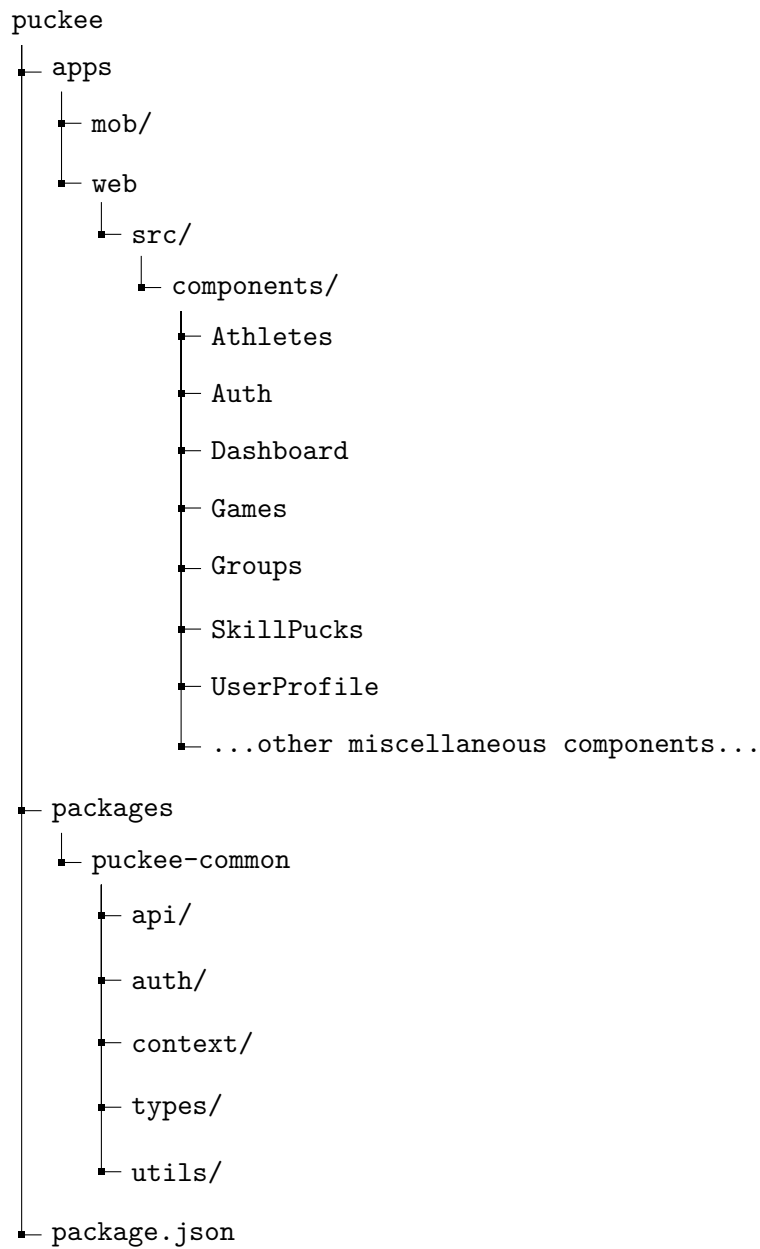


Figure 7.3: Platform’s front end React and React Native clients’ directory structure. The root-level `package.json` defines two Yarn workspaces: `apps` and `packages`. The `apps` package contains the platform-specific source code for the web and mobile clients in their appropriately-named directories, while both of them import shared modules code from the `puckee-common` package

Chapter 8

Conclusion

Multiple aims were outlined for this thesis, all of which were the necessary preliminary steps towards achieving the main goal – the development of a Platform facilitating the organization of collective amateur sports, namely the game of ice hockey.

Firstly, a thorough analysis of the key aspects of amateur ice hockey organization had to be performed to support the definition of appropriate Platform’s use cases, solving the most pressing of issues. Those were identified as, first, the lack of available amateur goaltenders, caused by an inefficient or rather non-existent market, and, second, the inability to keep the player attendance counts consistent in an environment of high fixed costs.

With these challenges to be solved, the user interface design phase followed, generating user interface mock-ups for both of the target platforms determined by the Personas’ analysis – a web application client and a mobile application. After the initial struggle with the Figma user interface designer tool, its component structuring capabilities later greatly facilitated the generation of multiple mock-ups iterations based on feedback from potential users.

Emphasizing the web client, while simultaneously planning to further develop the mobile application in the future, the REST API server written in Python programming language was chosen as a suitable back end technology, whose foundations were deeply analyzed along with those of React JavaScript/TypeScript web front end library and its React Native variant for creating cross-platform mobile applications.

In the implementation phase, bearing in mind the Platform’s complexity, the successfully fulfilled aim was to develop the core features of the web application, supported by the necessary REST API back end endpoints. To maximize the user experience, advanced use cases of asynchronous communication, such as the data *pagination*, were studied and employed. Additionally, the modern principles of *lazy registration* guided the design of the Platform’s registration process. Next, the groundwork was laid for further development of the mobile client, originally and partly implemented in the Application Development for Mobile Devices course at FIT BUT. This naturally led the *monorepo* repository scheme adoption, providing the necessary basis for the code sharing mechanisms available by the selection of React and React Native as the two front end frameworks.

In general, the goals of this thesis were fulfilled, while many development tasks and ideas are still outstanding – most importantly, in the short term, it is to implement the interactive part of the skill rating feature, the post game player rating. Next step is to implement the User Profile page, enabling the user to see his statistics, further strengthening the user engagement. A long term plan is to add an e-mail subscription mechanism to regularly report the games attended by the followed athletes.

Bibliography

- [1] Model-View-Controller Pattern. In: *Learn Objective-C for Java Developers*. Berkeley, CA: Apress, 2009, p. 353–402. ISBN 978-1-4302-2370-2.
- [2] AHL.CZ. *Soutěže a turnaje registrované na AHL.cz* [<https://www.ahl.cz/registrovane-souteze>]. January 2022. Accessed: 2022-01-04.
- [3] BROWN, T. *An Introduction to the Python Web Server Gateway Interface (WSGI)*.
- [4] CANADA, S. *Canada's population clock (real-time model)* [<https://www150.statcan.gc.ca/n1/pub/71-607-x/71-607-x2018005-eng.htm>]. January 2022. Accessed: 2022-01-04.
- [5] CHERNY, B. *Programming TypeScript: Making Your JavaScript Applications Scale*. O'Reilly Media, 2019. ISBN 9781492037606. Available at: <https://books.google.cz/books?id=YemUDwAAQBAJ>.
- [6] CONTRIBUTORS, M. *An overview of HTTP* [<https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>]. December 2021. Accessed: 2022-01-05.
- [7] CONTRIBUTORS, M. *Responsive design* [https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design]. April 2022. Accessed: 2022-04-20.
- [8] COOK, J. *How the React Native bridge works and how it will change in the near future* [<https://dev.to/wjimmycook/how-the-react-native-bridge-works-and-how-it-will-change-in-the-near-future-4ekc>]. August 2020. Accessed: 2021-12-17.
- [9] COPELAND, R. *Essential SQLAlchemy*. Firstth ed. O'Reilly, 2008. ISBN 9780596516147.
- [10] COS, D. *Overview of Docker Compose* [<https://docs.docker.com/compose/>]. August 2021. Accessed: 2021-01-13.
- [11] COUGH, C. *Countries by number of ice hockey rinks in 2020/21* [<https://www.statista.com/statistics/282353/countries-by-number-of-ice-hockey-rinks/>]. December 2021. Accessed: 2022-01-04.
- [12] COUGH, C. *Countries by number of registered ice hockey players in 2020/21* [<https://www.statista.com/statistics/282349/number-of-registered-ice-hockey-by-country/>]. December 2021. Accessed: 2022-01-04.
- [13] DATA.AI. *Who Uses Apps? A Mobile App Demographics Primer* [<https://www.data.ai/en/academy/uses-apps-mobile-app-demographics-primer/>]. January 2022. Accessed: 2022-04-15.

- [14] EBY, P. J. *Python Web Server Gateway Interface v1.0*. PEP 333. 2003. Available at: <https://www.python.org/dev/peps/pep-0333/>.
- [15] EISENMAN, B. *Learning React Native*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2016.
- [16] FEDERATION, I. I. H. *Season Summary 2020 edition* [<https://blob.iihf.com/iihf-media/iihfmvc/media/downloads/annual%20report/seasonsummary2020b.pdf>]. January 2021. Accessed: 2022-04-15.
- [17] FIELDING, R. T. and RESCHKE, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [RFC 7231]. RFC Editor, june 2014. DOI: 10.17487/RFC7231. Available at: <https://rfc-editor.org/rfc/rfc7231.txt>.
- [18] FIELDING, R. T. *REST: Architectural Styles and the Design of Network-based Software Architectures*. 2000. Doctoral dissertation. University of California, Irvine. Available at: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [19] GARRETT, J. J. Ajax: A New Approach to Web Applications. In:. 2007.
- [20] GRINBERG, M. *Flask Web Development: Developing Web Applications with Python*. 1stth ed. O'Reilly Media, Inc., 2014. ISBN 1449372627.
- [21] GUARDIA, C. de la. *Python Web Frameworks*. USA: O'Reilly Media, Inc., 2016. ISBN 9781491938096.
- [22] JONES, M., BRADLEY, J. and SAKIMURA, N. *JSON Web Token (JWT)* [RFC 7519]. RFC Editor, may 2015. DOI: 10.17487/RFC7519. Available at: <https://www.rfc-editor.org/info/rfc7519>.
- [23] KEVIN BURKE, R. H. F. S. G. B. *Flask-RESTful* [<https://flask-restful.readthedocs.io/en/latest>]. May 2020. Accessed: 2022-01-11.
- [24] LEE, Y.-H. and YUAN, C. W. The Privacy Calculus of “Friending” Across Multiple Social Media Platforms. *Social Media + Society*. 2020, vol. 6, no. 2, p. 2056305120928478. DOI: 10.1177/2056305120928478.
- [25] LINSLEY, T. *React Query: Overview* [<https://react-query.tanstack.com/overview>]. August 2020. Accessed: 2021-05-05.
- [26] MARCOTTE, E. *Responsive Web Design*. A Book Apart, 2011. Book Apart. ISBN 9780984442577. Available at: <https://books.google.cz/books?id=vhe4XwAACAAJ>.
- [27] MASINTER, L. M. *Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)* [RFC 2324]. RFC Editor, 1. april 1998. DOI: 10.17487/RFC2324. Available at: <https://rfc-editor.org/rfc/rfc2324.txt>.
- [28] RED HAT, I. *What is a REST API?* [<https://www.redhat.com/en/topics/api/what-is-a-rest-api>]. May 2020. Accessed: 2022-01-09.
- [29] RICHARDSON, L., AMUNDSEN, M. and RUBY, S. *RESTful Web APIs*. O'Reilly Media, Inc., 2013. ISBN 1449358063.

- [30] ROUTER, O. *What is REST? Use in industry* [<https://www.opc-router.com/what-is-rest>]. December 2021. Accessed: 2022-01-06.
- [31] SAINT EXUPÉRY, A. de. *Airman's Odyssey*. Harcourt Brace Jovanovich, 1984. ISBN 9780156037334.
- [32] SAURO, J. *Customer analytics for dummies*. New Delhi: Wiley, ©2015.
- [33] TOM, N. *Should You Give Users Access Before They Register* [<https://auth0.com/blog/should-you-give-users-access-before-they-register/>]. October 2021. Accessed: 2022-04-26.
- [34] TOTTY, B., GOURLEY, D., SAYER, M., AGGARWAL, A. and REDDY, S. *Http: The Definitive Guide*. USA: O'Reilly Associates, Inc., 2002. ISBN 1565925092.
- [35] TURNBULL, J. *The Docker book*. October 2014.
- [36] WIERUCH, R. *Road to React: Your journey to master plain yet pragmatic React.js*. CreateSpace Independent Publishing Platform, 2017. ISBN 1979807078. Available at: https://books.google.cz/books/about/The_Road_to_Learn_React.html?id=NFD6swEACAAJ&redir_esc=y.
- [37] WROBLEWSKI, L. *Gradual Engagement Boosts Twitter Sign-Ups by 29%* [<https://www.lukew.com/ff/entry.asp?1128>]. June 2010. Accessed: 2022-04-26.
- [38] [ČSÚ], C. S. O. *Population* [https://www.czso.cz/csu/czso/obyvatelstvo_lide]. October 2021. Accessed: 2022-01-04.

Appendix A

Full web application mock-ups

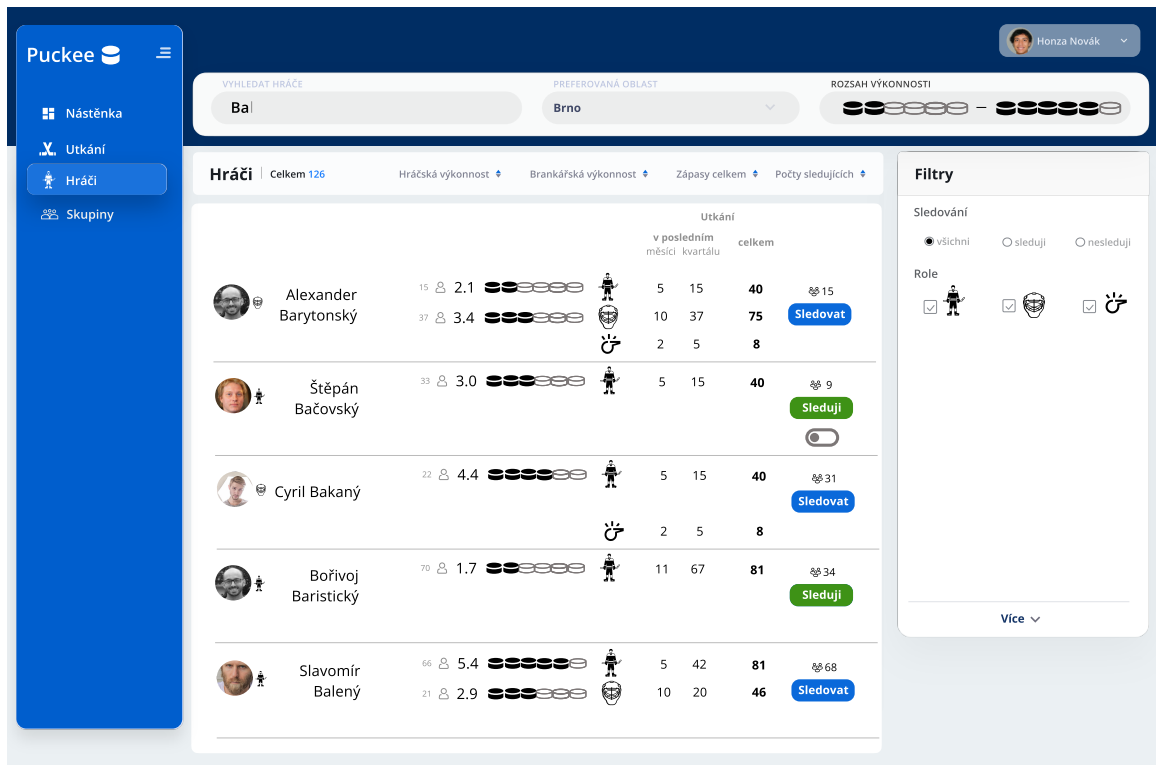


Figure A.1: **Athlete Overview page**: A filterable infinite list of players to find the teammates from the past games to either follow them or to find their current skill rating.

Puckee

- Nástěnka
- Utkání
- Hráči
- Skupiny

Honza Novák

Moje nejbližší utkání

Honza a kamarádi

2

🕒 07:15 - 08:15, St 14/11

📍 winninggroup Arena

👤 19/20 🏆 1/2 🇨🇪 0/1

250 Kč

Uskupení démonů

🕒 19:30 - 20:45, St 14/11

📍 Sportcentrum Lužánky, NHL plocha

👤 12/20 🏆 1/2 🇨🇪 0/1

+200 Kč

Rádoby hokejisti

🕒 08:15 - 09:30, Pá 16/11

📍 Hokejová hala Úvoz

👤 19/20 🏆 2/2 🇨🇪 0/1

300 Kč

Listopad 2022

Po	Út	St	Čt	Pá	So	Ne
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

07:15 - 08:15 🏆
Honza a kamarádi, winninggroup Arena

19:30 - 20:45 🏆
Uskupení démonů, Sportcentrum Lužánky, NHL plocha

Mohlo by Tě zajímat

Jan Řízek, Jaroslav Novotný, Břetislav Honzík, Petr Prokop, Roman Kulich se účastní.
Břetislav Honzík organizuje.

Hokejová tlupa

🕒 07:15 - 08:15, Pá 28/11

📍 TJ Stadion Brno

👤 19/20 🏆 1/2 🇨🇪 +250 Kč

0/1 🇨🇪 +200 Kč

[Přihlásit se](#)

Jaroslav Novotný, Břetislav Honzík, Petr Prokop se účastní

Uskupení démonů

🕒 10:15 - 11:15, Pá 30/11

📍 Sportcentrum Lužánky, NHL plocha

👤 15/20 🏆 1/2 🇨🇪 +250 Kč

0/1 🇨🇪 +200 Kč

[Přihlásit se](#)

Petr Prokop, Salazar Březina se účastní

Příslib hokejovna

🕒 06:15 - 07:15, Po 2/12

📍 Hokejová hala Úvoz

👤 11/20 🏆 2/2 🇨🇪 +250 Kč

0/1 🇨🇪 +200 Kč

[Přihlásit se](#)

Jan Bořivojský, Petr Prokop, Roman Kulich se účastní.
Květoslav Novák organizuje.

Honza a kamarádi

🕒 07:15 - 08:15, Pá 28/11

📍 winninggroup Arena

👤 19/20 🏆 1/2 🇨🇪 +250 Kč

0/1 🇨🇪 +200 Kč

JDU

Jan Řízek, Jaroslav Novotný, Břetislav Honzík, Petr Prokop, Roman Kulich se účastní.

Uskupení démonů

🕒 07:15 - 08:15, Pá 28/11

📍 winninggroup Arena

👤 19/20 🏆 1/2 🇨🇪 +250 Kč

0/1 🇨🇪 +200 Kč

JDU

Figure A.2: **Dashboard page:** My Upcoming Games overview of the three nearest games, Games Calendar simplifying game planning by providing timeline perspective and Might Interest You overview of the selected games attended/organized by the largest amount of user's followed athletes.

The screenshot displays the Puckee Games Overview page. The main content area shows a list of games with the following details:

Game Name	Time	Location	Score	Price	Team
Honza a kamarádi	07:15 - 08:15, Pá 28/11	winninggroup Arena	17/20	250 Kč	JDU
Nováčci	18:00 - 19:15, Pá 28/11	Sportcentrum Lužánky	16/20	250 Kč	JDU
Honza a kamarádi	10:30 - 11:45, So 29/11	Hokejová Hala Úvoz	12/20	250 Kč	Přihlásit se
Byznysmeni	07:15 - 08:15, Ne 30/11	TJ Stadion Brno	16/20	250 Kč	Přihlásit se
HC Bezejmenní	07:15 - 08:15, Po 1/12	Sportcentrum Lužánky	20/20	250 Kč	JDU
Giorgio et al	08:00 - 07:15, Po 1/12	Hokejová Hala Úvoz	15/20	300 Kč	Přihlásit se
Honza a kamarádi	10:30 - 11:45, St 3/12	Sportcentrum Lužánky	13/20	250 Kč	JDU
Nováčci	08:00 - 09:15, Čt 4/12	Hokejová Hala Úvoz	9/20	200 Kč	JDU

The right sidebar contains a 'Filtry' section with the following options:

- Organizátor: libovolný
- Den v týdnu: libovolný, po - pá, so - ne, zvolit
- Časové preference: 06:30 - 23:00
- Volná místa: 0 - 11

Figure A.3: Games Overview page: An infinite list of games along with a complex set of filters enabling the user to create his own Games Dashboard.

Puckee ☰ ← Honza a jeho kamarádi - pátek Přihlásit se Honza Novák

Základní informace

5. 12. 2022 08:45 - 10:00 Sportcentrum Lužánky, Brno Honza Novák Nezapomeňte s sebou vzít dostatek puků!
 Počítejte s tím, že budeme možná hrát o něco déle.

Očekávaná úroveň 12/20 250 Kč 2/2 +250 Kč 1/1 +100 Kč Reálná úroveň

Účastníci 17/20

Registrování: 13 Radomír Čestný Adam Datelný
Alexander Barytonský Josef Huk Přemysl Kališnický
Martin Garilický Jiří Dechytka Novomír Eklektický
Zikmund Zalehováci Jakub Oravský Šimon Fügner
Radomír Novotný Salazar Přichozi

Neregistrovaní: 4 Lukáš N Miloš B
Andres Jimenez

Diskuse

Kdo se mnou půjde po hokeji na pivo? Martin Garilický
 pondělí 4. 12. 2022, 18:52

Šel bych, kam máš v plánu jít? Adam Datelný
 úterý 4. 12. 2022, 07:07

Honza Novák Já asi nakonec nemůžu, pardon. Budu unavený po celém týdnu a asi zvolím spíše klidový režim.
 úterý 4. 12. 2022, 18:52

Domluvíme se na místě. Představoval bych si něco poblíž haly, nemám moc času. Martin Garilický
 úterý 4. 12. 2022, 18:52

Hodnocení

Ohodnot, prosím, hru vybraných účastníků:

Alexander Barytonský Martin Garilický Šimon Fügner Alexandr Barytonský Josef Huk Jiří Dechytka Adam Datelný

Uložit hodnocení

Figure A.4: **Game Detail page**: Renders basic game data, detailed list of both registered and unregistered attendees as well as Game Chat for game-related discussions. Evaluation section is opened to access after a game to provide an interface for evaluation of selected athletes. This input serves for athlete's skill rating calculation.

Puckee Nové utkání Honza Novák

Základní údaje

Titulek utkání: Honza a jeho kamarádi - pátek

Poznámky: Nezapomeňte s sebou vzít dostatek puků!
Počítejte s tím, že budeme možná hrát o něco déle.

Organizátoři: Honza Novák Přidat organizátora

Soukromé utkání:

Místo: Sportcentrum Lužánky, Brno

Datum: 5. 12. 2022

Opakovat utkání: týdně

Cena pronájmu: 3960 Kč/hod

Ostatní náklady: 150 Kč

Začátek: 08:45

Konec: 10:00

Očekávané počty účastníků: 20 2 1

Odhadovaná cena pro hráče: 300 Kč

Očekávaná úroveň:

Hráči v poli

Přidat hráče + Registrovaní: 13

Alexander Barytonský - Radomír Čestný - Adam Datelný - Jiří Dechytka -

Novomír Eklektický - Šimon Fugner - Martin Garlický - Josef Huk -

Přemysl Kališnický - Radomír Novotný - Jakub Oravský - Salazar Příchozí -

Zikmund Zalehovací -

17/20

Neregistrovaní: 4

Lukáš N - Miloš B - Andres - Jimenez -

Brankáři

Přidat brankáře + Registrovaní: 1

Alexander Barytonský -

1/2

200 Kč

Neregistrovaní: 0

Nedávno organizovaná

Honza a jeho kamarádi - pátek
pá 30.11. 07:00 - 08:15 📄

Honza a jeho kamarádi - pátek
pá 30.11. 07:00 - 08:15 📄

Honza a jeho kamarádi - pátek
pá 30.11. 07:00 - 08:15 📄

Honza a jeho kamarádi - pátek
pá 30.11. 07:00 - 08:15 📄

Orientační kalkulace

x 1.25h -4950 Kč

x 2 -400 Kč

x 1 -200 Kč

ostatní náklady -150 Kč

Skupiny

Nejčastější úterní 📄
Úterní hokeje - Adam, Petr, Karel a zbytek

Zelení 📄
Tým, proti kterému hrajeme nejčastěji

Středechní mohykáni 📄
Původní skupina středechních

Vytvořit utkání

Figure A.5: **Game Admin** page: Provides interface for determining basic game features and offers helper components such as Recently Organized Games and estimated budgeting to facilitate the process. Also provides an interface to add both registered and unregistered players into the game. Note: Referee role was intentionally skipped to fit the screen capture on one page.

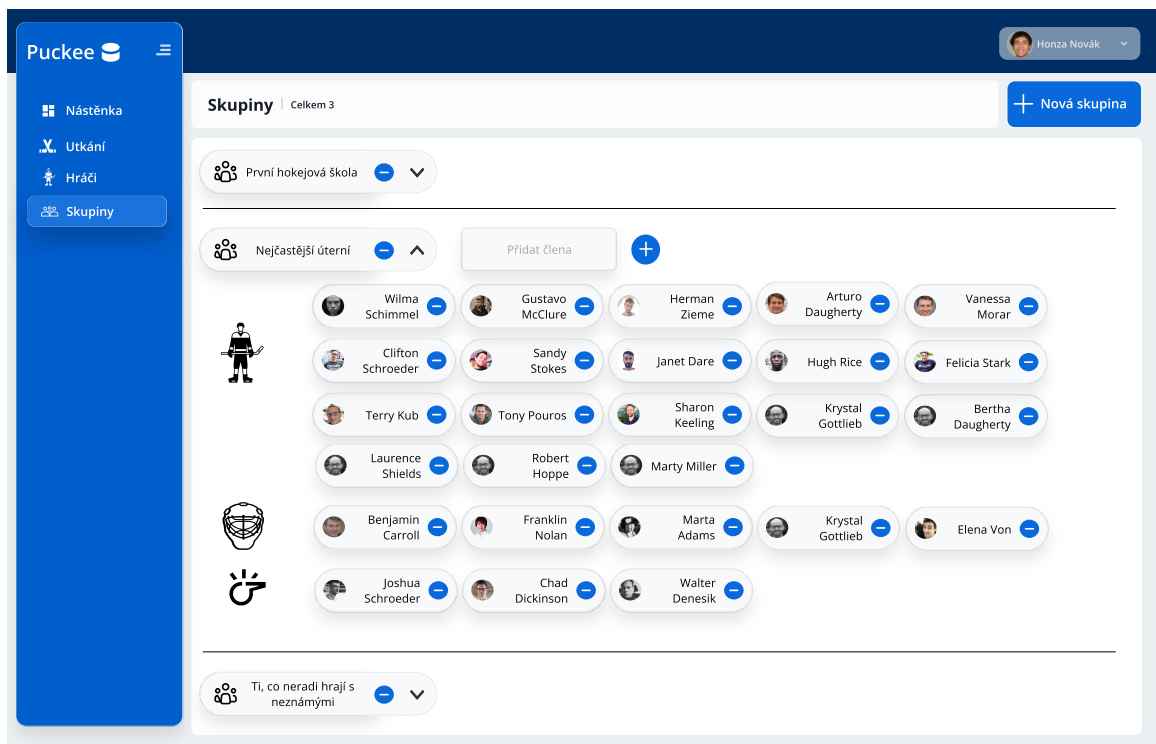


Figure A.6: **Groups page:** An interface for game organizers to predefine groups of athletes, following them in the *opt-out* mode, i.e. enabling him/her to add them to a game without requiring their confirmation. Groups can then be added to a game by a single click of an organizer.

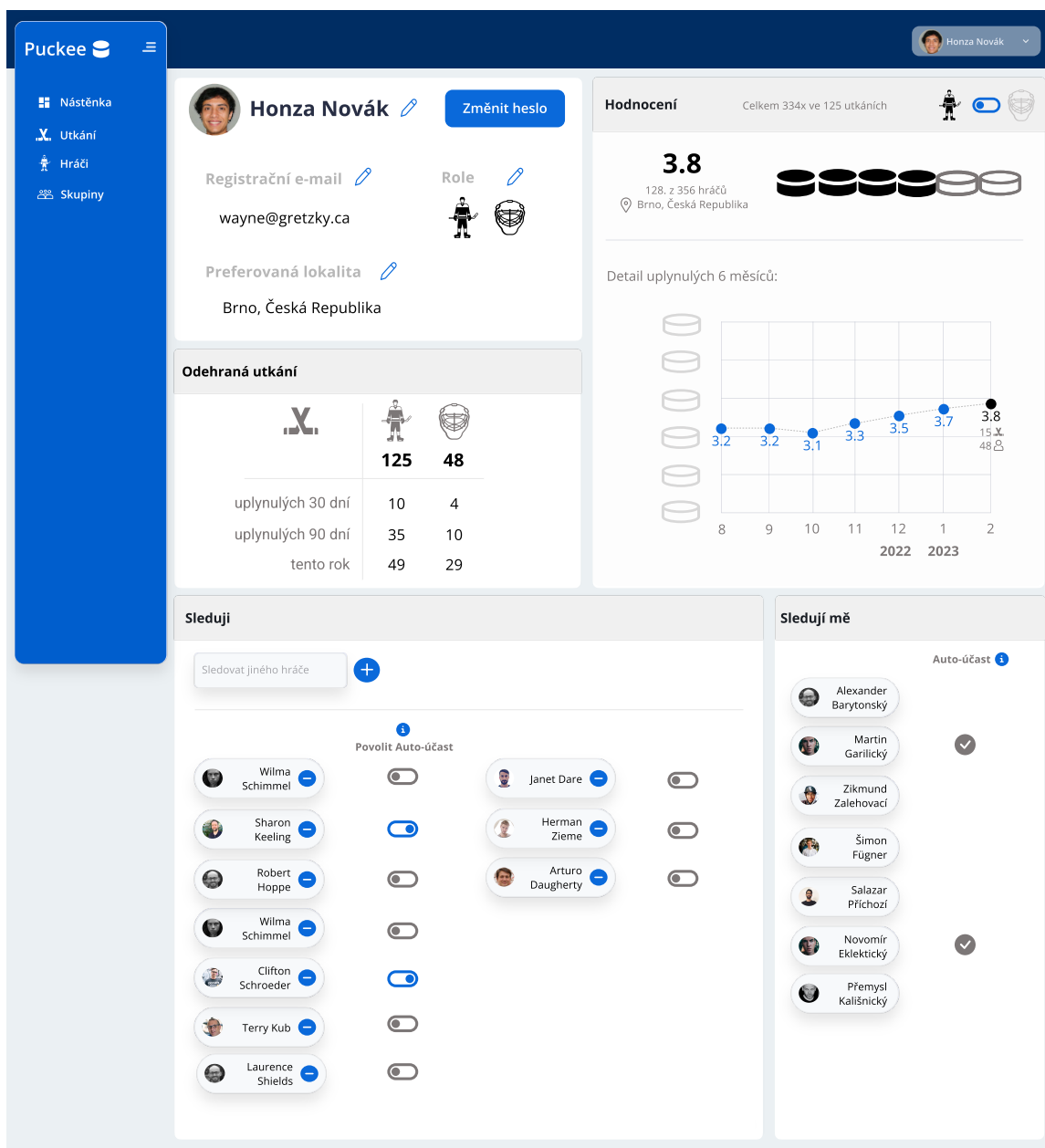


Figure A.7: **User Profile page**: Provides interface for user profile management, statistics of the played games in different roles, as well as the development of his/her skill rating based on post-game evaluation by his/her teammates (as noted in Figure A.4). Also an interface to manage the followed athletes is provided and all user's followers are listed in the last section.

Appendix B

Web application screen captures

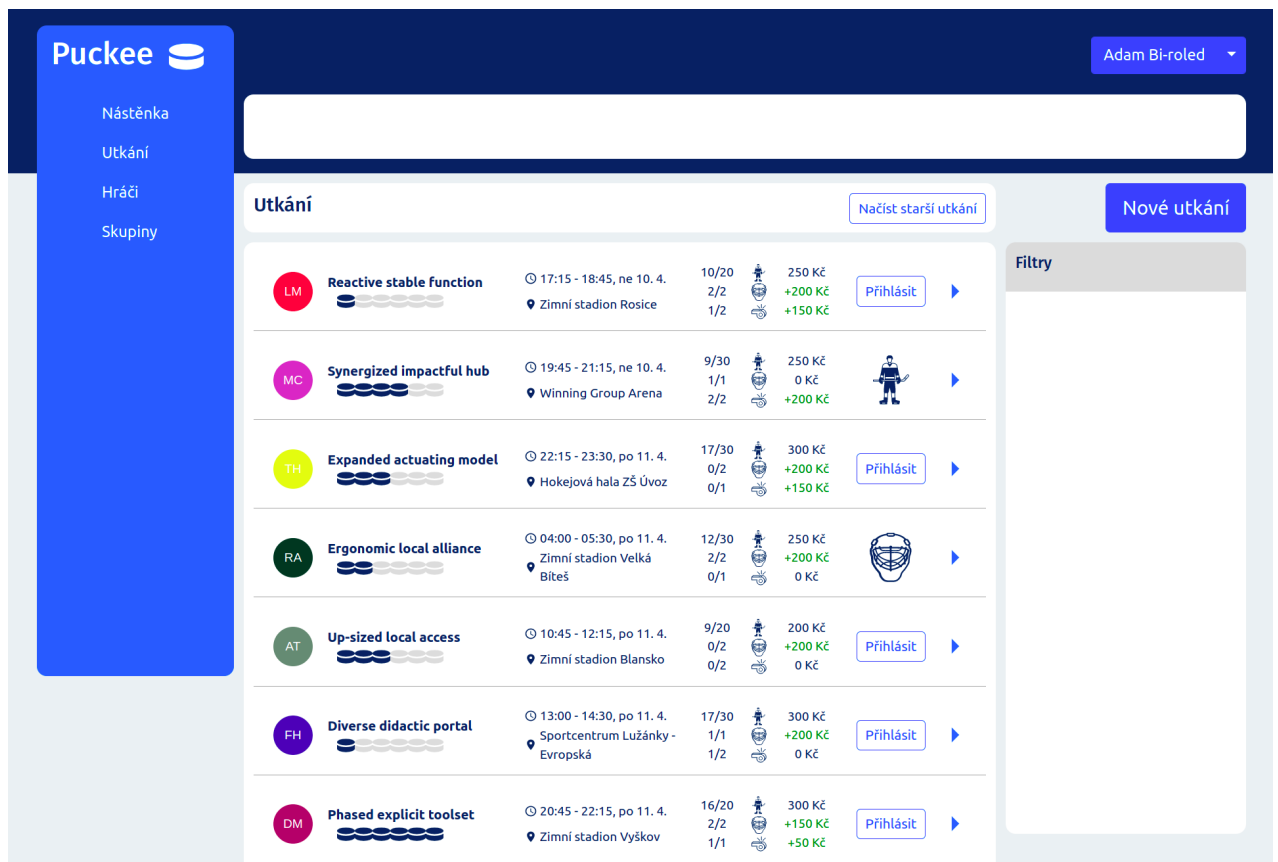


Figure B.1: Games Overview: A *paginated* infinite list of games displayed to a user with all their key properties. To be extended with complex filtering capabilities in the future, enabling the user to predefine and save a set of filters to effectively create his own Games Dashboard (in addition to the Dashboard provided by Puckee).



MC	Synergized impactful hub	🕒 19:45 - 21:15, ne 10. 4. 📍 Winning Group Arena	9/30 1/1 2/2	👤 250 Kč 🧤 0 Kč 🔄 +200 Kč		▶
TH	Expanded actuating model	🕒 22:15 - 23:30, po 11. 4. 📍 Hokejová hala ZŠ Úvoz	17/30 0/2 0/1	👤 300 Kč 🧤 +200 Kč 🔄 +150 Kč	Přihlásit	▶
RA	Ergonomic local alliance	🕒 04:00 - 05:30, po 11. 4. 📍 Zimní stadion Velká Bíteš	12/30 2/2 0/1	👤 250 Kč 🧤 +200 Kč 🔄 0 Kč		▶

Figure B.2: Games as Games Overview items with different states of Attendance Shortcut Selector in case of user with the Player and the Goaltender attendance roles.

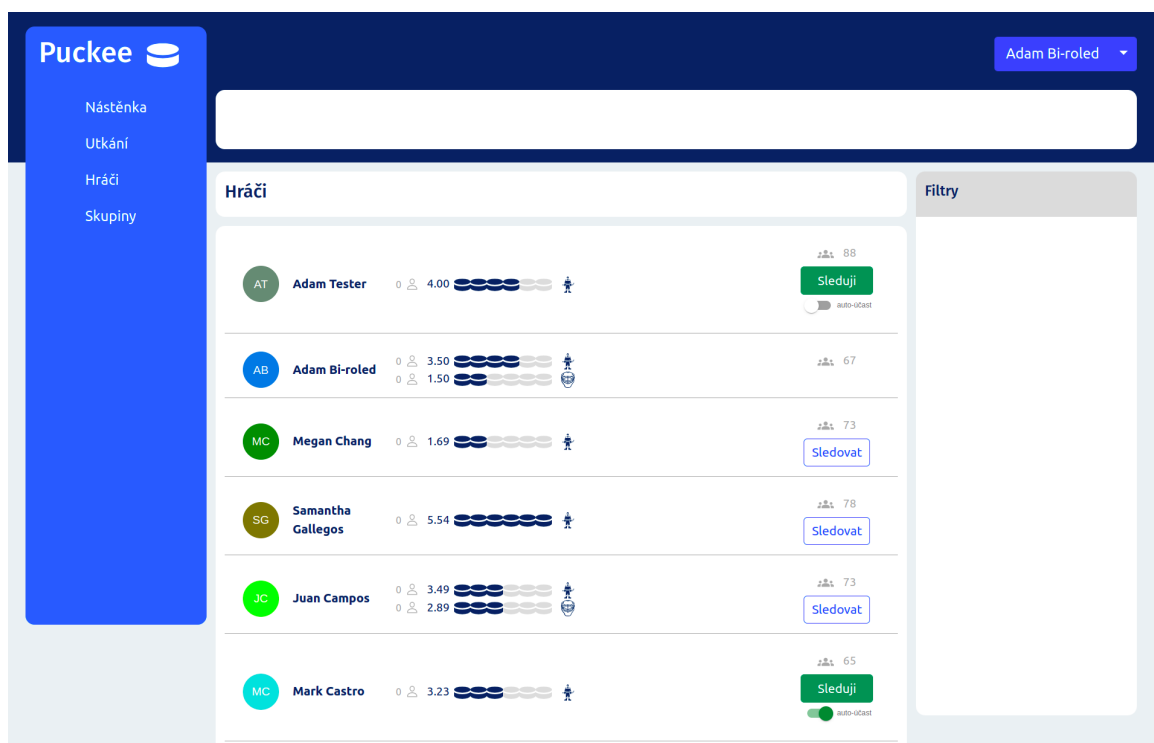


Figure B.3: Athlete Overview as a page to filter the athletes worth following and find out their basic characteristics, such as their current skill level, as evaluated by their playmates, as well as the count of games played during the last month, quarter and a year. Game Statistics and the Athlete Filters features have not been implemented by the time of this thesis's submission.

Puckee

- Nástěnka
- Utkání
- Hráči
- Skupiny

Adam Bi-roled ▾

Tvoje nejbližší utkání

Secured tertiary success

🕒 11:30 - 13:00, pá 1. 4.

📍 Zimní stadion Vyškov

👤 10/20 🏆 1/1 📈 0/2

250 Kč

Seamless local adapter

🕒 19:30 - 20:30, so 9. 4.

📍 Zimní stadion Kuřim

👤 17/20 🏆 1/1 📈 2/1

200 Kč

Ergonomic local alliance

🕒 04:00 - 05:30, po 11. 4.

📍 Zimní stadion Velká Bíteš

👤 12/30 🏆 2/2 📈 0/1

+200 Kč

Kalendář

« < **květen 2022** > »

	PO	ÚT	ST	ČT	PÁ	SO	NE
17	25.	26.	27.	28.	29.	30.	1.
18	2.	3.	4.	5.	6.	7.	8.
19	9.	10.	11.	12.	13.	14.	15.
20	16.	17.	18.	19.	20.	21.	22.
21	23.	24.	25.	26.	27.	28.	29.
22	30.	31.	1.	2.	3.	4.	5.

Mohlo by Tě zajímat

Chelsea Robinson, Linda Stewart, Christopher Wallace, Kathy Harrington 🏆 **Emma Smith** 🏆 **Rachel Lane** 🏆 se účastní.

Virtual mobile website

🕒 15:15 - 16:45, pá 1. 4.

📍 Hokejová hala ZŠ Úvoz

12/30

1/1

2/2

300 Kč

+150 Kč

0 Kč

[Přihlásit](#)

Logan Allen, Hannah Ellis, Audrey Gregory, Samuel Todd 🏆 se účastní.
Andrew Hendricks organizuje.

Focused radical extranet

🕒 02:00 - 03:00, pá 1. 4.

📍 Zimní stadion Velká Bíteš

18/30

1/1

1/1

200 Kč

+200 Kč

0 Kč

[Přihlásit](#)

Patrick Frank, Justin Fitzgerald 🏆 se účastní.

Focused local project

🕒 13:45 - 15:15, pá 1. 4.

📍 Zimní stadion Vyškov

17/20

0/2

2/2

200 Kč

+150 Kč

+150 Kč

[Přihlásit](#)

Linda Stewart 🏆 se účastní.

Focused systemic adapter

🕒 02:30 - 04:00, pá 1. 4.

📍 Zimní stadion Kuřim

9/30

0/2

0/2

250 Kč

+150 Kč

+150 Kč

[Přihlásit](#)

Figure B.4: The Dashboard page rendering My Upcoming Games, Games Calendar and Might Interest You components. My Upcoming Games displays three nearest games attended by the user, Might Interest You component recommends games attended and/or organized by followed athletes. Games Calendar component has not been connected to the application back end yet by the time of this thesis's submission.

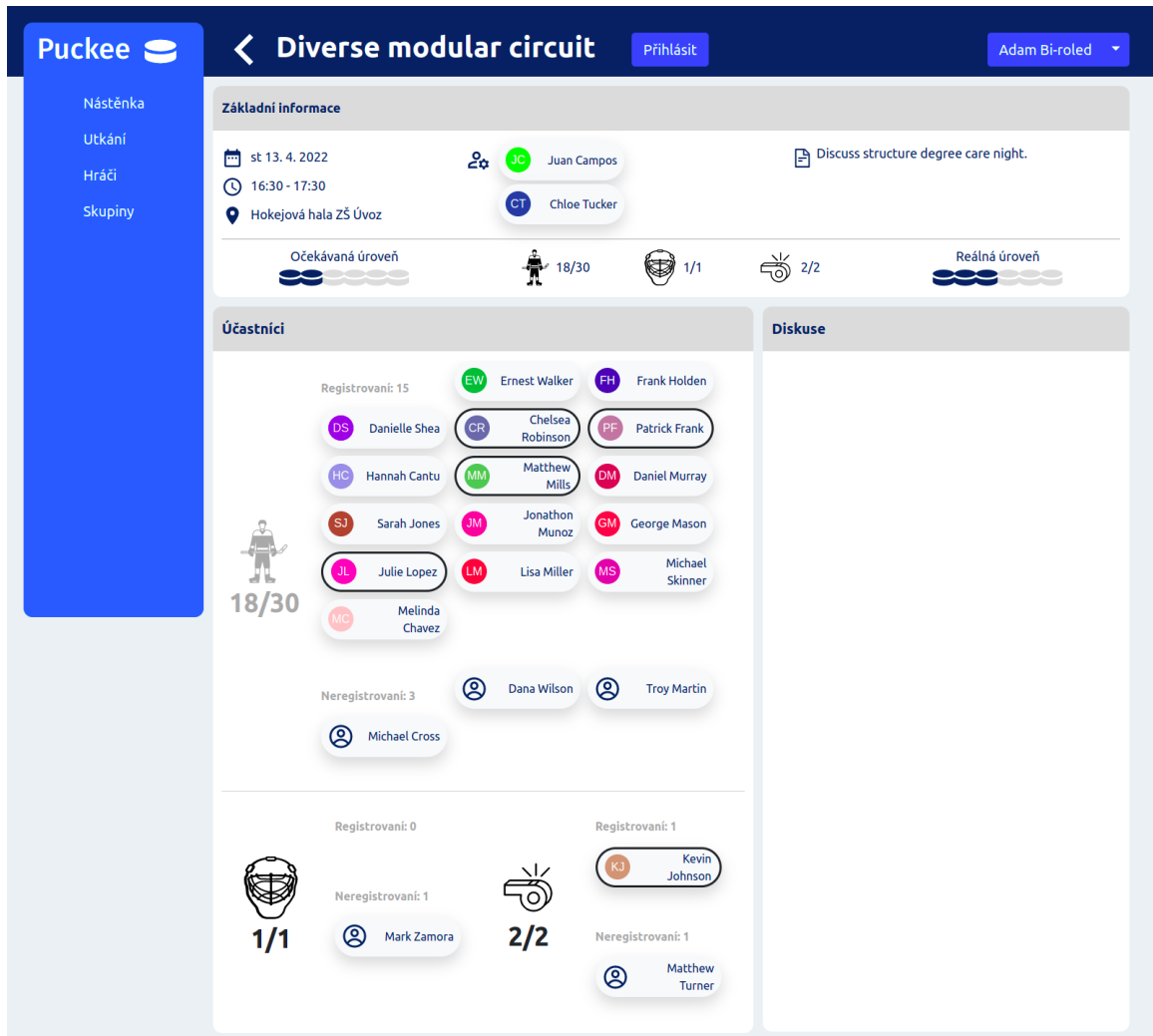


Figure B.5: Game Detail page divided into three sections: Basic Game Data, Game Attendance and a Game Chat which has not been implemented by the time of this thesis submission. Game Attendance showing the comprehensive overview of all game participants while highlighting the followed ones to make the game more attendance-worthy.

Puckee ← Moje nová hra Adam Bi-roled

Základní údaje

Titulek utkání: Poznámky:

Organizátoři: AB Adam Bi-roled

Místo konání: Datum:

Cena pronájmu ledu: Kč/h Ostatní náklady: Kč Čas začátku: Čas konce:

Požadované speciální role a jejich odměny: ks Kč ks Kč

Požadované počty hráčů a odhadovaná cena: ks Kč

Očekávaná úroveň:

Hráči v poli

Hledej hráče: Registrovaní hráči: 1 Megan Compton

2/20 NoName Player Neregistrovaní hráči: 1

Skupiny

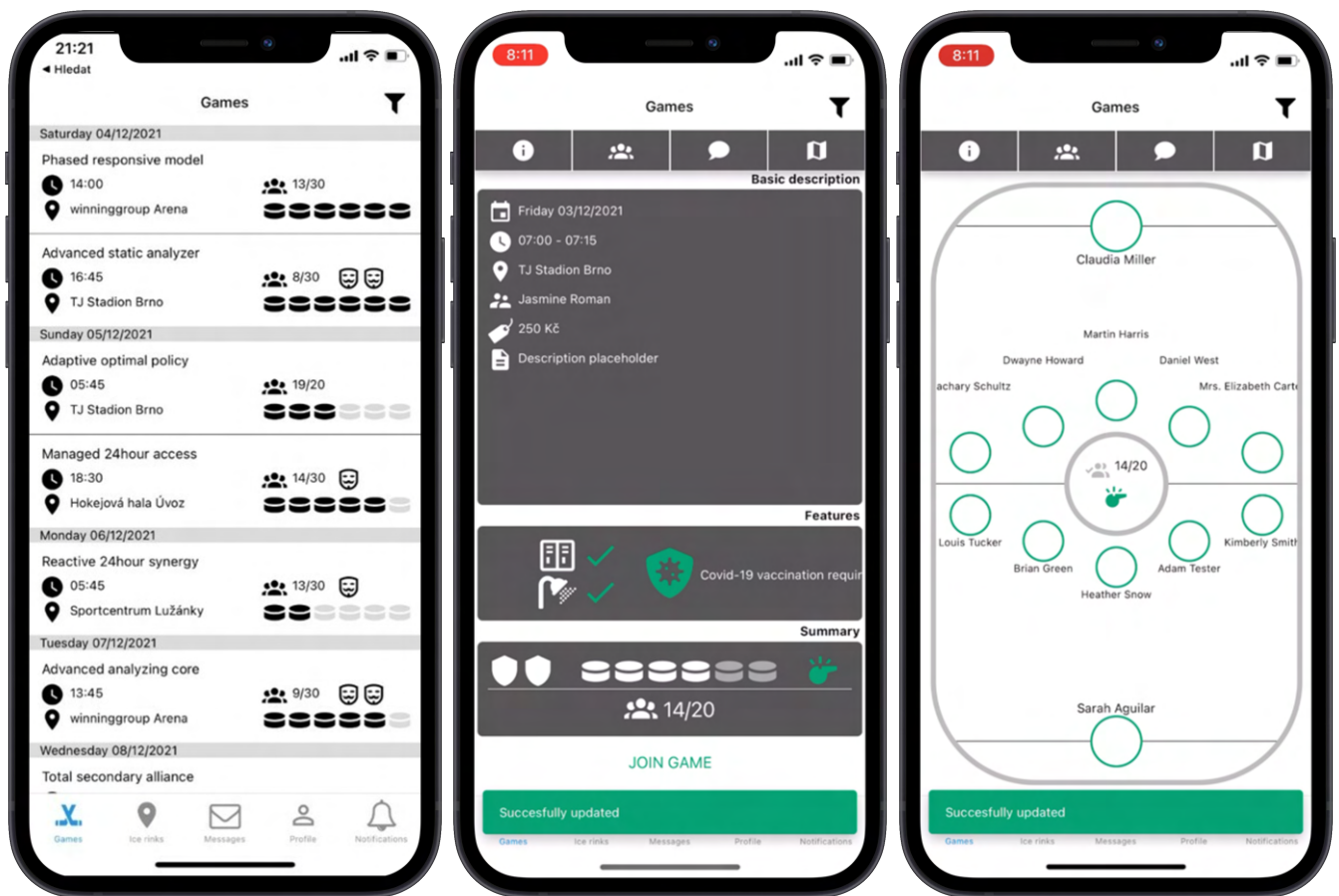
Nedávno organizované

Orientační kalkulace

Figure B.6: The New Game page represented by the `GameAdminForm` component, instantiating `GameAdminParticipants` for each of the attendance roles. Only its `Player` role instance is captured in the screenshot. Additionally, three helper components `GameAdminAvailableGroups` to suggest groups of followed people to be added, `GameAdminFinancialEstimate` to assist with game's budgeting and `GameAdminRecentlyOrganized` to provide an interface to duplicate previous events' details are instantiated, all of which are subject to future development.

Appendix C

Mobile application screen captures



(a) Games Overview

(b) Game Detail: Basic Info

(c) Game Detail: Attendance

Figure C.1: Selected screens from the mobile client: Games Overview screen structured as a *section list* grouped by date. The two subscreens of the Game Detail screen — Basic Info and Attendance.

Appendix D

Included storage medium contents

- `puckee-api`: a repository containing the REST API server implementation
- `puckee`: a repository containing the front-end code for both web and mobile applications
 - `apps/web`: web-specific implementation
 - `apps/mob`: mobile-specific implementation
 - `packages/puckee-common`: shared modules used by both clients
 - `mockups`: a directory containing all the mock-ups presented in this work in the SVG format
- `poster.svg`: a poster presenting the implemented Platform
- `presentation.mp4`: a short video presentation showing the Platform's features