FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Implementace webové varianty hry Dobyvatel
Student:	Bc. Marek Bajtalon
Vedoucí:	Ing. Viktor Černý
Studijní program:	Informatika
Obor / specializace:	Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Implementujte webovou variantu kvízové hry Dobyvatel, která v současnosti existuje pouze jako aplikace pro mobilní telefony.

Požadavky:
- Analyzujte existující webové technologie a vyberte vhodné kandidáty na implementaci backendu a frontendu aplikace.
- Navrhněte aplikaci od začátku tak, aby umožnila snadné rozšiřování a úpravy uživatelského rozhraní.
- Implementujte aplikaci za použití technologií, které vychází z předchozí analýzy.
- Výsledné řešení bude připraveno na user-management tak, aby se mohli uživatelé registrovat a sledovat své statistiky, nebo se jednoduše spojit s dalšími uživateli.
- Celá aplikace bude implementována jako open-source a zdrojové kódy včetně dokumentace budou k dispozici na veřejně dostupném repozitáři.
- Backend aplikace by měl být robustní tak, aby umožnil dlouhodobý provoz.
- Aplikaci řádně otestujte.
- Při testování se zaměřte především na spolehlivost backendu.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 17. února 2023 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Implementace webové varianty hry Dobyvatel

Bc. Marek Bajtalon

Katedra softwarového inženýrství
Vedoucí práce: Ing. Viktor Černý

4. května 2023

Poděkování

V první řadě bych chtěl poděkovat svému vedoucímu Ing. Viktoru Černému za pomoc a konzultace během psaní této diplomové práce. Dále bych chtěl poděkovat své rodině za čas a motivaci během mého studia a také svým kamarádům, kteří mi pomohli s testováním práce a tvorbou otázek.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. května 2023 . . . . . .

Odkaz na tuto práci

Bajtalon, Marek. *Implementace webové varianty hry Dobyvatel*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Abstrakt

Tato diplomová práce se zabývá reimplementací známé webové hry Dobyvatel, která už není nyní dostupná. V práci se podíváme na analýzu, návrh, implementaci, nasazení a testování. Nejdříve se zaměříme na analýzu dostupných alternativ a vybereme technologie vhodné k implementaci vlastního řešení. To bude nabízet stejné funkcionality, které dělaly Dobyvatele charakteristickým – tj. hra s přáteli, znalostní otázky, zabírání mapy České republiky apod. Dále se podíváme na návrh architektury aplikace, doménový model a wireframe (drátěný model). Aplikace bude postavena jako klient–server řešení a bude napsána v jazyce Typescript, což je nadstavba Javascriptu. V textu se ještě zaměříme na implementaci, otestování výkonu backendu aplikace (tj. kolik aktivních hráčů najednou server zvládne) a aplikaci nasadíme na virtuální privátní stroj za pomocí Github Actions.

Klíčová slova Dobyvatel, trivia, Angular, NgRx, Typescript, Websockets, SocketIO, NestJS

Abstract

This thesis deals with the reimplementation of the well-known web game Dobyvatel, which is no longer available. In the thesis, we will focus on the analysis, design, implementation, deployment, and testing. First, we will analyze the available alternatives and select technologies that are suitable to implement our own solution. It will offer the same functionalities that made Dobyvatel distinctive – i.e., playing with friends, trivia questions, conquering the regions of the Czech Republic map, etc. Next, we will look at the design of the application architecture, domain model, and wireframe. The application will be built as a client–server solution and will be written in Typescript, which is an extension of JavaScript. In the text, we will still focus on the implementation, test the performance of the backend (i.e. how many active players the server can handle at once) and deploy the application to a virtual private server using Github Actions.

Keywords Dobyvatel, trivia, Angular, NgRx, Typescript, Websockets, SocketIO, NestJS

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Oblíbená webová hra Dobyvatel byla bohužel zrušena a byla nahrazena verzemi pro mobilní zařízení. Ty se ale nesetkaly s příliš velkou vlnou úspěchu, spíše naopak. Jedním z důvodů bude pravděpodobně předělaný grafický kabátek, který zbytečně používá 3D grafiku a ta podle recenzí uživatelů vypadá „jako z minulého století“. Další velmi kritizovanou věcí jsou mikrotransakce, které velmi zvýhodňují hráče, kteří platí. Hra jim např. napovídá správné odpovědi anebo si mohou některé témata otázek rovnou úplně zakázat.

S kamarády jsme si chtěli už několikrát zavzpomínat a zahrát si starou dobrou klasiku. To už ale bohužel není možné a to chci v této práci změnit. Navrhnu a naimplementuji si vlastní řešení, které bude obsahovat funkcionalitu, která dělala Dobyvatele tak charakteristickým – tj. hru s přáteli, dobývání krajů České republiky, vědomostní otázky apod. To celé bude snadno rozšiřitelné a bude dostupné jako open–source řešení, aby do něj mohli přispívat i ostatní.

Kapitola 1

Cíl práce

Cílem práce je znovu–implementace webové verze hry Dobyvatel, která už nyní není bohužel dostupná. Podíváme se na dostupné alternativní řešení a poté na technologie, s jejichž pomocí bude možné si napsat vlastní řešení. To bude napsáno tak, aby byla zajištěna co největší kompatibilita zařízení a uživateli stačil pouze webový prohlížeč. Budeme také podporovat přihlašování pomocí sociálních sítí, aby uživatel neztrácel čas s tvorbou účtu a mohl se rovnou pustit do hraní.

Dále navrhneme architekturu naší aplikace, její doménový model a wireframe. Aplikace bude dostupná v repozitáři na platformě Github a budeme ji nasazovat na virtuální privátní server za pomocí Github Actions. Zaměříme se na implementaci klíčových prvků ze hry Dobyvatel tak, aby zážitek byl stejný jako dříve. To znamená, že hra bude podporovat hru více hráčů, přidávání přátel, prohlížení statistik a bude nabízet stejné typy otázek jako původní verze.

Ve finále se zaměříme na otestování výkonu serveru – tj. jaký maximální nápor je schopen server ustát a kolik hráčů může naši hru hrát najednou.

Kapitola 2

Analýza

2.1 Historie Dobyvatele

V první řadě bych chtěl čtenářům více přiblížit, o jaký typ hry se vlastně jedná a proč jsem se rozhodl pro tvorbu této klasiky.

Vraťme se společně do roku 2010, kde TV Nova spouští hru Dobyvatel [8]. Vědomostní hru pro více hráčů, která hned nabývá na popularitě a je hrána většinou žáků nejen během hodin informatiky. Během dvou let od startu si ji vyzkoušelo více než půl milionu lidí [9].

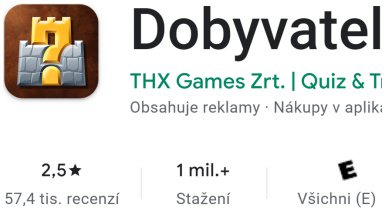
Hra nabízí vědomostní otázky rozdělené do kategorií, jako je například historie, geografie, věda, umění apod. Správnými odpověďmi hráči postupně dobývají území České republiky a rozšiřují tak své impérium.

Za každou vyhranou hru získá hráč body a posune se výše v žebříčku. Dále se zde každý týden konají turnaje, kde je možné vyhrát zajímavé ceny a porovnat své vědomosti a strategické schopnosti s ostatními hráči.

2.2 Zpět do přítomnosti

Tomu je již bohužel konec. V roce 2016 dostala hra nový grafický kabátek, který schytal velkou vlnu kritiky a v roce 2021 byla hra na počítače pohřbena úplně. Zda byl na vině Adobe Flash, ve kterém byla původní verze napsána, nebo něco jiného, zůstává ve hvězdách.

Aktuálně je hra dostupná pouze na mobilních zařízeních v kompletně předělaném designu a s novými prvky, které opět schytaly velkou vlnu kritiky. Hráči dokonce vytvořili petici za starého Dobyvatele [10], ale nic se od té doby nezměnilo. Zatím...



(a) Hodnocení aplikace v Google Play [3]

(b) Hodnocení aplikace v Apple Appstore [11]

Na obrázku můžeme vidět, že aplikace nemají úplně lichotivé hodnocení. S kamarády jsme si už chtěli několikrát zahrát Dobyvatele a nostalgicky zavzpomínat, ale nic tomu podobného jsme úplně nenašli. To v této práci změníme!

Pro ukázku přikládám ještě srovnání vývoje grafického rozhraní jednotlivých verzí.

Obrázek 2.1: „Starý dobrý Dobyvatel“, grafické rozhraní, které si pamatují všichni [1]

Obrázek 2.2: Upravený grafický kabátek vydaný 5 let po spuštění původní verze [2]

Obrázek 2.3: Nové 3D grafické rozhraní pro mobilní zařízení [3]

2.3 Analýza současných řešení

Dobyvatel je typem tzv. trivia hry. Jedná se o typ, kde jsou hráči dotazování na otázky z různých témat a musí na ně odpovídat. Témata mohou být různorodá, např. historie, věda, geografie, sporty, filmy a seriály apod. Dobyvatel není úplně klasická trivia hra, obohacuje tuto vědomostní část o strategické zabírání regionů a útočení na ostatní hráče. To je to, co ho dělá unikátním.

V této části se pokusím vybrat řešení, která jsou co nejvíce podobná Dobyvateli a zanalyzovat je.

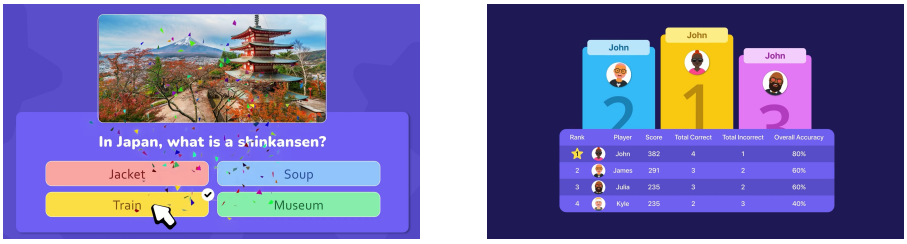
2.3.1 TriviaNerd

Jedna z nejpopulárnějších trivia platforem, nabízí vlastní tvorbu otázek a hostování her [12]. Využívají ji velké společnosti jako jsou Netflix, Meta, Amazon a další a je vhodná i pro učitele, kteří chtějí potrénovat vědomosti svých žáků.

Nejedná se o stejný koncept jako dobyvatel, nedochází k žádnému zabírání regionů. Zde vyhrává pouze ten, kdo má největší počet správných odpovědí.

Hra se dále chlubí tím, že jejich databáze obsahuje přes sto tisíc otázek a přes tisíc různých témat. Je dostupná jako webová aplikace a je responzivní jak pro mobilní zařízení, tak pro počítače.

Velkou nevýhodou je ale její cena, hostování her totiž není zdarma. V levnějším tarifu TriviaNerd Pro, který vychází na 59$ měsíčně (při měsíční fakturaci) hra nabízí až 100 aktivních uživatelů. V dražším, který vychází na slušných 399$ (opět při měsíční fakturaci), nabízí až 1000 aktivních hráčů v naší hře. Pochybuji ale, že tuto sumu parta kamarádů zaplatí jen proto, aby si párkrát zahráli.



(a) Obrazovka s otázkou [12] (b) Výsledné umístění hráčů [12]

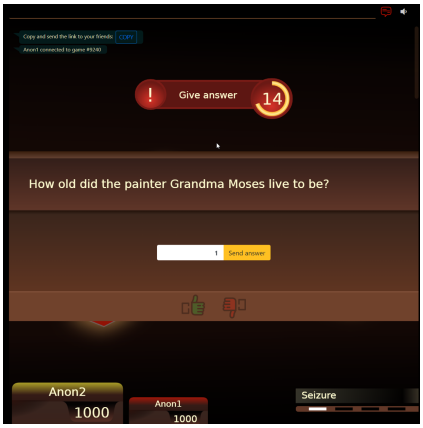
Obrázek 2.4: Ukázky obrazovek z TriviaNerd

2.3.2 ConQUIZtador.online

Hra velmi inspirovaná původní verzí Dobyvatele. Neodehrává se avšak na mapě České republiky ani jiného státu. Jako mapa zde slouží hexagony, které si můžeme před začátkem hry sestavit do námi chtěné podoby. Funkčně je ale hra velmi ořezaná, nabízí pouze základ – hru s přáteli. Turnaje, veřejný chat apod. tu bohužel nenajdeme. Hra je taky vyvíjena primárně v angličtině a ačkoli nabízí možnost přepnutí jazyka i do češtiny, je vidět, že jsou otázky i rozhraní přeložené strojově. Hra je inspirována Dobyvatelem opravdu hodně, „propůjčila“ si i zvukové efekty a samotný vzhled [13].

Škoda je, že je hra pouze v poměru stran 4:3. Nalezneme ji jako webovou aplikaci na adrese https://conquiztador.online.

(a) Mapa [13]



(b) Obrazovka s tipovací otázkou [13]

Obrázek 2.5: Ukázky obrazovek z conquiztador.online

2.3.3 Mobilní verze Dobyvatele

Poslední aplikací, kterou nesmíme opomenout, je mobilní verze Dobyvatele. Tu vyvíjí maďarská společnost THX Games. Hra je lokalizována do více než patnácti jazyků a je dostupná jak pro Android, tak iOS. Původní 2D design byl nahrazen až směšně vypadající 3D grafikou, kterou v recenzích spousta hráčů odsuzuje.

Hra dále nabízí mikrotransakce, to znamená, že ten kdo platí, má oproti neplatícím hráčům výhodu. Může získat správnou odpověď na otázku, kterou by jinak nevěděl nebo zaútočit na území hráče, na které by jinak zaútočit nemohl. Dále si může třeba zakázat témata otázek, která se mu nelíbí.

Hra ale jinak nabízí vše, co nabízel původní Dobyvatel – hru s přáteli, velkou databázi otázek i turnaje. Novinkou je pak zakládání klanů, inventář a změna postavy, se kterou během hry útočíme.

(a) Obrazovka s tipovací otázkou

(b) Hlavní menu hry

Obrázek 2.6: Ukázky rozhraní mobilní verze Dobyvatele

2.4 Analýza požadavků

V této sekci se zaměříme na rozdělení požadavků na funkční (co má být provedeno) a nefunkční (jakým způsobem to má být provedeno).

2.4.1 Funkční požadavky

- **F01 – Registrace/přihlášení uživatele** – Uživatel má možnost si vytvořit svůj účet a přihlásit se.
- **F02 – Přihlášení pomocí sociálních sítí** – Uživatel má možnost se přihlásit pomocí sociálních sítí.
- **F03 – Zobrazení seznamu přátel** – Uživatel si může zobrazit svůj seznam přátel a podívat se, kdo je online.
- **F04 – Historie odehraných her** – Uživatel si může zobrazit historii svých odehraných her a podívat se, jak se mu dařilo.
- **F05 – Odeslání žádosti o přátelství** – Uživatel má možnost odeslat žádost o přátelství.
- **F06 – Přijetí/odmítnutí žádosti o přátelství** – Uživatel může přijmout/odmítnout žádost o přátelství.
- **F07 – Vyhledávání her/matchmaking** – Aplikace bude podporovat matchmaking, pokud mají 3 hráči zapnuté vyhledávání hry, aplikace jim vytvoří hru a připojí je do ní.
- **F08 – Synchronizace** – Stav hry bude synchronizován živě mezi všemi hráči dané hry tak, aby nikdo neměl časovou či jinou výhodu.
- **F09 – Reprezentace pomocí mapy** – Hra bude zobrazovat stav hry na mapě ČR, kde každý region patřící danému hráči bude zabarven jeho barvou.
- **F10 – Základna hráče** – Každý hráč má na mapě svou základnu, která má 3 životy. Pokud je tato základna poražena, útočící hráč získává všechna území poraženého hráče.
- **F11 – Skóre hráče** – Každý hráč získá na začátku hry za základnu 1000 bodů. Za každý další zabraný prázdný region pak 200. Za každý ztracený region pak 200 bodů ztrácí.

  Pokud hráč zabere cizí region, získává 400 bodů. Tento bodový stav se bude hráčům ukazovat v levém dolním rohu během hry.
- **F12 – Tipovací otázky** – Aplikace bude umožňovat pokládat tipovací otázky, na ty je časový limit a odpovídá se na ně pouze číslem.

- **F13 – Otázky s jednou správnou odpovědí** – Aplikace bude umožňovat pokládat otázky, kde je na výběr ze 4 odpovědí a pouze jedna je správná. Na tyto otázky je opět časový limit.

  Pokud oba odpoví špatně, nic se nestane.

  Pokud oba odpoví dobře, o vítězi útoku rozhodne další tipovací otázka.

  Pokud odpoví správně útočník, získává území a pokud odpoví správně obránce, region si ponechá a dostává 100 bodů na své konto.

- **F13 – Výběr regionu** – Hráč bude mít možnost zvolit si region, který chce zabrat, poté co správně odpoví na otázku.

- **F14 – Zabírání regionů** – Pokud hráč zaútočí na území jiného hráče a odpoví správně, zabere mu tento region.

- **F15 – Konec hry a uložení výsledku** – Aplikace na konci hry zobrazí výsledné skóre hráčům a výsledek hry uloží do databáze, aby bylo možné se na něj později podívat.

2.4.2 Nefunkční požadavky

- **N01 – Frontend** – Klientská část aplikace (frontend) bude napsán v Angularu.
- **N02 – Backend** – Serverová část aplikace (backend) bude napsán v NestJS.
- **N03 – Databáze** – Pro ukládání dat bude použita databáze MySQL.
- **N04 – HTTPS** – Aplikace bude dostupná pouze přes protokol HTTPS.
- **N05 – Bezpečnost** – Aplikace bude využívat pro autentizaci a autorizaci uživatelů JWT tokeny.
- **N06 – Integrace a nasazení** – Zdrojový kód aplikace bude umístěn na Github a bude využívat Github Actions pro integraci a nasazení (CI a CD) na VPS.
- **N07 – Určeno pro desktop** – Aplikace bude vytvořena primárně pro desktop a na telefony nebude kvůli již existujícím aplikacím brán nyní ohled.

2.5 Analýza použitých technologií

2.5.1 Javascript

Jedná se o programovací jazyk, který běží v prohlížeči a je jedním ze stavebních kamenů moderních webových aplikací. Je interpretovaný, dynamicky typovaný

a objektově orientovaný. Díky tomu, že je interpretovaný, nemůže samozřejmě nikdy dosahovat takového výkonu jako C nebo C++, ale to ani není jeho cílem.

Ačkoli by shoda jmen Java a Javascript mohla budit dojem, že se jedná o rozšíření Javy, není tomu tak. Autoři ho zvolili pouze z marketingových důvodů.

Dále díky příchodu NodeJS je možné spouštět Javascript i na serveru, čehož využijeme, abychom mohli mít napsané obě části aplikace – frontend i backend v jednom jazyce.

2.5.2 Typescript

Nadstavba Javascriptu od Microsoftu, která měla za úkol usnadnit vývoj komplexních aplikací v Javascriptu. Přidává klíčové vlastnosti jako je statické typování, rozhraní, třídy apod. [14].

Do prohlížečů není potřeba nic instalovat, Typescript se zkompiluje zpět do Javascriptu a ten už podporuje drtivá většina dnes používaných prohlížečů. Během kompilace se ještě zkontrolují datové typy, null checky apod., takže můžeme snadno objevit chyby, které by se jinak projevily až během běhu kódu.

Nespornou výhodou je i to, že každý Javascript kód je i validním Typescript kódem a není potřeba tedy všechny existující knihovny přepisovat.

2.5.3 Výběr frontendového frameworku

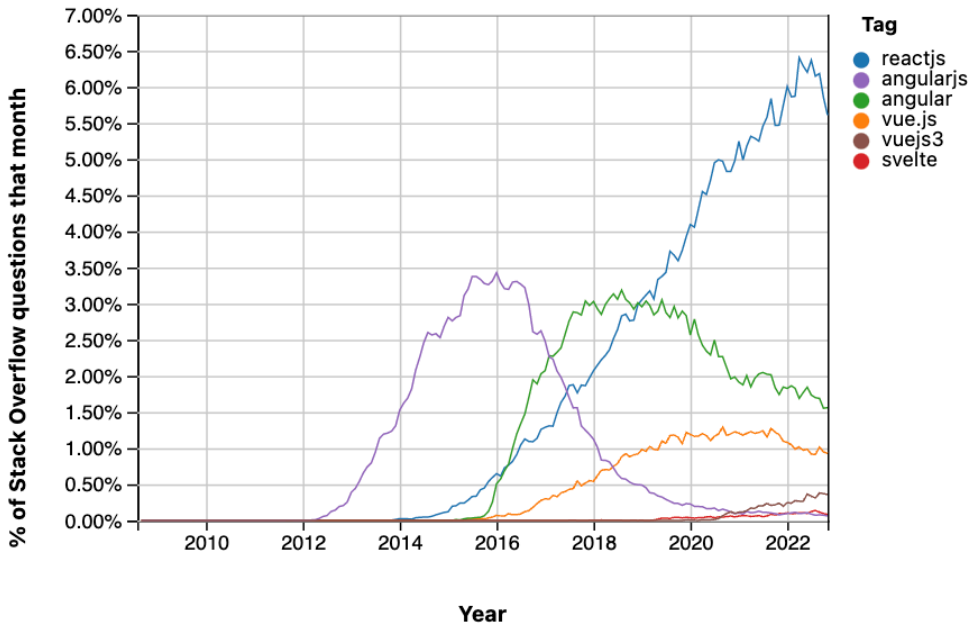
Další otázkou bude volba správného frontendového frameworku, v následující sekci si shrneme tři nejpopulárnější, jejich klady a zápory, a nakonec si ten nejvhodnější, pro naši práci, vybereme.

Na začátek bych ještě přidal zajímavý graf 2.7, který ukazuje popularitu jednotlivých frameworků. Je zřejmé, že na první příčce je React, to ale neznamená, že by ostatní frameworky byly nějak špatné či jim chyběla důležitá dokumentace nebo komunita. Je to pravděpodobně tím, že React má ve srovnání třeba s Angularem poměrně mírnější učící křivku.

2.5.3.1 Angular

Framework vytvořený firmou Google, původní verze AngularJS byla vytvořena již v roce 2010 a později byla přepsána od základu do Typescriptu, kde se i dost zásadně změnila struktura komponent a dalších způsobů používaní [15].

Základním kamenem jsou zde komponenty. Každá obsahuje kód, šablonu, styly a testy. Výhodou komponent je to, že je můžeme vkládat do jiných komponent a použít je tak vícekrát. Komponenty mezi sebou mohou komunikovat pomocí služeb (Services), které v celé aplikaci existují pouze jednou. Ty je pak možné díky vkládání závislostí (Dependcy Injection) zavolat z jiné komponenty nebo služby.



Obrázek 2.7: Oblíbenost jednotlivých frontendových frameworků [4]

Dalším důležitým stavebním kamenem jsou moduly, ty zapouzdřují aplikaci do logických celků (např. sekce před přihlášením a administrace) a tyto moduly je pak možné stahovat až když jsou opravdu potřeba (lazy loading). Je zbytečné, aby nepřihlášený uživatel stahoval moduly, které jsou dostupné až po přihlášení.

Poslední výhodou je Angular CLI, který umožňuje generovat komponenty, služby, moduly apod. z příkazové řádky, správně nastaví importy a my se tak nemusíme o nic starat.

2.5.3.2 React

Aktuálně nejpopulárnější framework, vydán firmou Facebook v roce 2013 pro tvorbu uživatelských rozhraní (UI). Opět zde hrají první housle komponenty, ty jsou složeny z šablony a kódu. Dále pak mají své vlastnosti a svůj vnitřní stav [16].

Na toto navazuje pak knihovna Redux, kde je celý stav aplikace uložen v jednom objektu a pomocí volání akcí a následných redukujících funkcí stav měníme. Tento stavový princip začal být více oblíbený a převzaly si ho postupně i další frameworky, protože se snadněji ladí (debugguje) a snadněji se aplikace rozšiřuje.

Jednou z dalších výhod, co React nabízí, je React Native. Ten umožňuje psát nativní aplikace pro platformy Android a iOS za použití Reactu.

Ačkoliv je React frontendový framework běžící v prohlížeči, je možné ne-

chat ho renderovat šablony i na serveru (server side rendering). Je to především kvůli lepšímu škálování a optimalizaci pro vyhledávače – SEO (search engine optimization).

2.5.3.3 Vue

Poslední z velké trojice frontendových frameworků, tentokrát ale za ním nestojí žádná z velkých firem. Stojí za ním programátor *Evan You*, kterému se velmi líbila práce s AngularemJS, konkrétně to, že vlastně vůbec nemusí pracovat se strukturou HTML (DOM), ale vadily mu jinak jeho dost těžké koncepty [17].

Rozhodl se tedy vyvinout vlastní alternativu, která si převzala manipulaci s DOMem a principy, které se mu nelíbily nechal Angularu.

Základem jsou opět komponenty, které jsou zde pojmenované jako Single-File Components [18]. Ty mají všechen kód, šablonu a styly v jednom souboru.

Vue opět podporuje renderování na serveru (SSR) a také má vlastní knihovnu pro uchovávání stavu inspirovanou Reduxem - Vuex.

2.5.3.4 Shrnutí a výběr

Nakonec jsem se rozhodl jít cestou Angularu, s frameworkem mám zkušenosti ať už z práce, tak i z mé bakalářské práce. Chci použít Typescript, pro což se Angular vyloženě vybízí a dále chci na backendu použít NestJS který, jak si později ukážeme, má velmi podobnou strukturu kódu a rozdělení do modulů, služeb a komponent jako Angular.

2.5.4 NgRx

NgRx je knihovna pro Angular, která přebírá oblíbené principy z knihovny Redux pro React. Slouží k usnadnění tvorby komplexních aplikací a používá reaktivní principy. Reaktivní aplikace je (podle reaktivního manifestu) [19]:

- **Responzivní** – Aplikace odpoví v požadovaném čase, pokud je nějaká odpověď vůbec možná.
- **Odolná** – Aplikace zůstane funkční i v případě výpadku některé z částí.
- **Pružná** – Aplikace zůstane funkční i při proměnlivé zátěži. Umí reagovat na změnu zátěže, změnu množství požadavků pomocí distribuce požadavků nebo replikace.
- **Zprávami–řízená** – Komponenty jsou jasně oddělené a aplikace je založena na asynchronním zasílání zpráv.

Jak jsem zmínil dříve, NgRx se inspiruje u Reduxu, který používá Flux–pattern přístup a je rozdělen na tyto části [20]:

- **Akce:** Jsou vyvolány v případě, že nastane nějaká akce. Každá akce má svůj typ, což je text (string) pomocí něhož je identifikovatelná a svůj payload. To je objekt, který obsahuje proměnné potřebné k vyvolání akce – v ukázce 2.8 je to `login` a `password`.

```
export const login = createAction(
  '[Login Page] Login',
  props<{ username: string; password: string }>()
);
```

Obrázek 2.8: Ukázka kódu – Vytvoření akce pro login

- **Selektory:** Selektory nám pomáhají vybrat nějaké specifické informace ze stavu. Výhodou je, že selektory jsou cachované (tj. ukládají si výsledky). Takže pokud v selektoru provádíme třeba nějaký výpočet, selektor si bude výsledek pamatovat do další změny stavu.

- **Store:** Jediný zdroj pravdy (single source of truth), obsahuje aktuální stav aplikace.

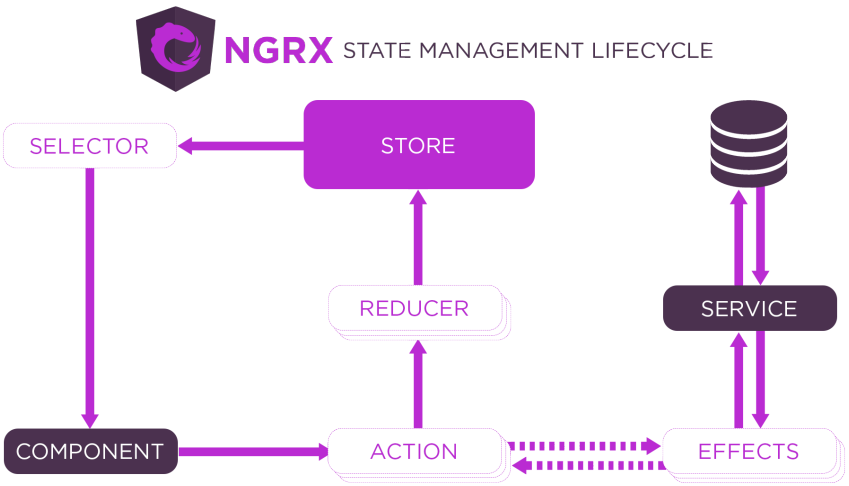
- **Reducery:** Funkce, které reagují na akce a podle toho mění stav aplikace. Funkce dostane na vstupu aktuální stav, který se nesmí modifikovat, protože je immutable (neměnný). Musíme tedy stav zduplikovat, což na ukázce 2.9 dělá spread operátor.

```
export const friendReducer = createReducer(
    initialState,
    on(loadFriends,
        (state) => {
            return {...state, friendsLoading: true};
    })
);
```

Obrázek 2.9: Ukázka kódu – Reducer pro friend state

- **Efekty:** Jedná se o efekty na akce, které se volají po tom, co reducery dokončí svou práci. Například pro volání API nebo pro ukládání do lokálního úložiště. Mohou vyvolat další akce (vedlejší efekty).

Na obrázku 2.10 je ještě vidět, jak funguje celý životní cyklus aplikace používající NgRx.



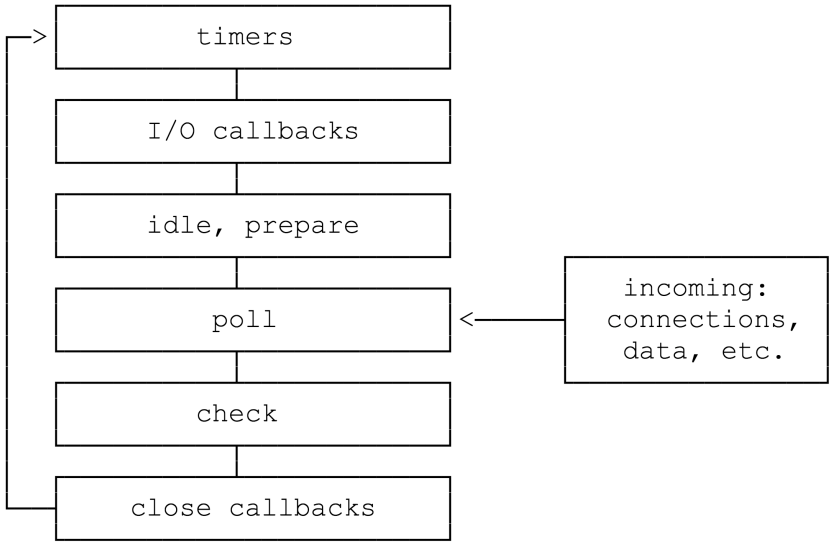
Obrázek 2.10: Životní cyklus NgRx [5]

2.5.5 NodeJS

NodeJS je open-source projekt, který umožňuje spouštění Javascriptu na straně serveru. Výhodou je, že se programátoři umějící Javascript nemusí učit nic nového, pokud chtějí psát full–stack aplikaci.

Je založen na Javascript jádru V8 z prohlížeče Google Chrome. Je napsán v C++ a je multiplatformní. Na rozdíl od ostatních server–side jazyků nevytváří pro každé nové spojení nové vlákno, ale všechno si drží v jednom. Nevzniká tak overhead s přidáváním dalších vláken [21].

Node využívá tzv. smyčku událostí (event loop, obrázek 2.11), která je asynchronní a I/O operace tak nezpůsobují zablokování vlákna [22]. Pokud ale chceme provádět nějaké složitější výpočty, budeme si muset nová vlákna nastartovat sami.



Obrázek 2.11: Smyčka události NodeJS [6]

2.5.6 Npm

Balíčkovací systém pro Javascript, nabízející přes 800 tisíc balíčků, což ho dělá největším na světě. Nabízí 3 hlavní komponenty [23]:

1. **CLI** – rozhraní pro příkazovou řádku, které se stará o instalaci a aktualizaci balíčků

2. **Registry** – centrální úložiště pro balíčky, které je sdíleno mezi vývojáři a komunitou

3. **Npm web** – obsahuje dokumentaci k balíčkům, návody jak je nainstalovat, apod.

V každém projektu pak najdeme soubor `package.json`, který obsahuje informace o nainstalovaných balíčcích a jejich verzích. Stačí tak verzovat pouze tento soubor a při instalaci na novém zařízení si npm natáhne správné verze.

2.5.7 Express

Express je nejpopulárnější Node framework pro vývoj server–side aplikací a tvorbu API [24]. Je velmi jednoduchý, a proto je základem pro další, více robustnější webové frameworky. Má i vlastní šablonovací systém a systém pro routování, nic z toho ale nepoužijeme, protože je až moc jednoduchý pro naše potřeby.

2.5.8 NestJS

Framework založený na Expressu a je aktuálně nejrychleji rostoucím NodeJS frameworkem pro tvorbu webových aplikací. Je napsán v Typescriptu a kombinuje prvky OOP (objektově orientovaného programování), FP (funkcionálního programování) a FRP (funkcionálně reaktivního programování) [25].

Je velmi silně inspirován architekturou a strukturou Angularu, což byl jeden z hlavních důvodů, proč jsem si ho vybral. Stejně jako Angular nabízí vlastní CLI, kde je možné generovat komponenty.

Jeho další výhodou je nativní podpora pro WebSockety, kterou rozhodně využijeme pro real–time komunikaci s klienty. Hlavní nevýhodou bude stejně jako u Angularu jeho strmá učící křivka, která může spoustu nováčků odradit.

2.5.9 Socket.IO

Knihovna pro real–time obousměrnou komunikaci mezi klientem a serverem. Uvnitř je založena na WebSocketech, které dnes podporuje přes 98 % [26] prohlížečů. I kdyby nastala výjimka a prohlížeč či spojení to nepodporovalo, umí Socket.IO přepnout na long–polling, ale to není tak efektivní jako WebSockety.

2.5.10 MySQL

Relační databáze, kterou není potřeba ani představovat. Je dobře škálovatelná, snadno nastavitelná a spolehlivá. Existuje už přes více než čtvrt století a pořád se v žebříčcích řadí mezi nejpoužívanější databázové systémy [27].

Pro dotazování je používán deklarativní jazyk SQL, ve kterém popisujeme, co chceme s daty udělat než jakým způsobem. V moderních aplikacích už se ale tolik samotné dotazování přes SQL nepoužívá, nahradilo ho ORM (objektově relační mapování).

2.5.11 ORM & TypeORM

ORM (objektově relační mapování) má za úkol propojit tabulky z relačních databází a objekty z objektově orientovaných jazyků. Práce s objekty je pro programátory přirozenější a není potřeba zdlouhavě psát SQL dotazy.

Další výhodou je, že pokud se změní struktura aplikace, ORM zajistí to, aby se struktura změnila i v databázi a naopak.

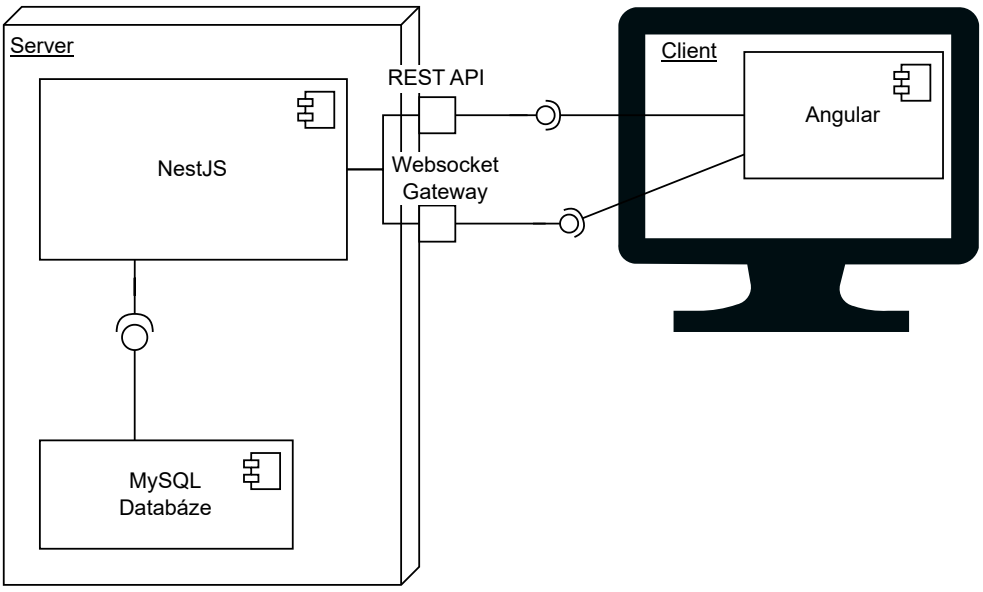
Poslední výhodou, kterou tu zmíním, je nezávislost na konkrétním databázovém stroji. ORM podporuje různé typy databází, nemusíme tedy opět řešit, co se děje uvnitř a pamatovat na specifické dotazy dané platformy.

TypeORM je knihovna, kterou využívá framework NestJS pro ORM a podporuje jak SQL, tak i NoSQL databáze.

Kapitola 3

Návrh

Další důležitou částí vývojového cyklu je návrh aplikace. Zpracovává výstup z analýzy, popisuje aplikaci z technického hlediska a může sloužit jako její dokumentace.



Obrázek 3.1: Diagram architektury aplikace

3.1 Architektura aplikace

Aplikace bude rozdělena na frontend (klientskou část) a backend (serverovou část). Klientská část bude spouštěná v prohlížeči a serverovová na serveru, společně budou komunikovat přes vystavené REST API a WebSockety, viz obrázek 3.1.

Výhodou rozdělení aplikace na klient–server je její snazší rozšiřování. Pokud bychom v budoucnu chtěli vyvinout třeba nativní aplikaci, budeme už nuceni vyvíjet pouze klientskou část.

Autentizaci a následnou autorizaci klienta budeme řešit pomocí JWT tokenů, na které se zaměříme v další části.

3.2 Autentizace a autorizace pomocí JWT tokenů

JSON Web Token [7] slouží k bezpečné výměně dat mezi dvěma stranami. Skládá se z:

(a) **hlavičky**, obsahující informace o způsobu výpočtu podpisu

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

(b) **payloadu (dat)**, typicky obsahující informace o uživateli

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

(c) **podpisu**, kterým ověříme, že zpráva nebyla na cestě změněna, vypočítáme ho následujícím způsobem:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Výsledky těchto tří částí zahashujeme pomocí Base64 a oddělíme tečkou. Výsledkem pro předchozí příklady bude JWT vypadat takto:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Obrázek 3.2: Výsledek výpočtu JWT tokenu [7]

Proces přihlášení bude tedy vypadat následujícím způsobem:

1. Uživatel se přihlásí (autentizuje) pomocí přihlašovacího jména a hesla nebo pomocí přihlášení přes sociální síť
2. Server vygeneruje na základě správných údajů JWT token a předá ho klientovi v cookie
3. Klient se již nemusí přihlašovat a vždy jenom v požadavku předá JWT token, dokud nevyprší jeho platnost

Hlavní výhodou oproti ukládání sezení (sessions) přímo na serveru, je jeho bezestavovost. Znamená to, že pokud bychom chtěli škálovat, nemusíme vůbec řešit u jakého uzlu se uživatel přihlašoval – je to totiž jedno, požadavek může vyřídit kterýkoliv uzel.

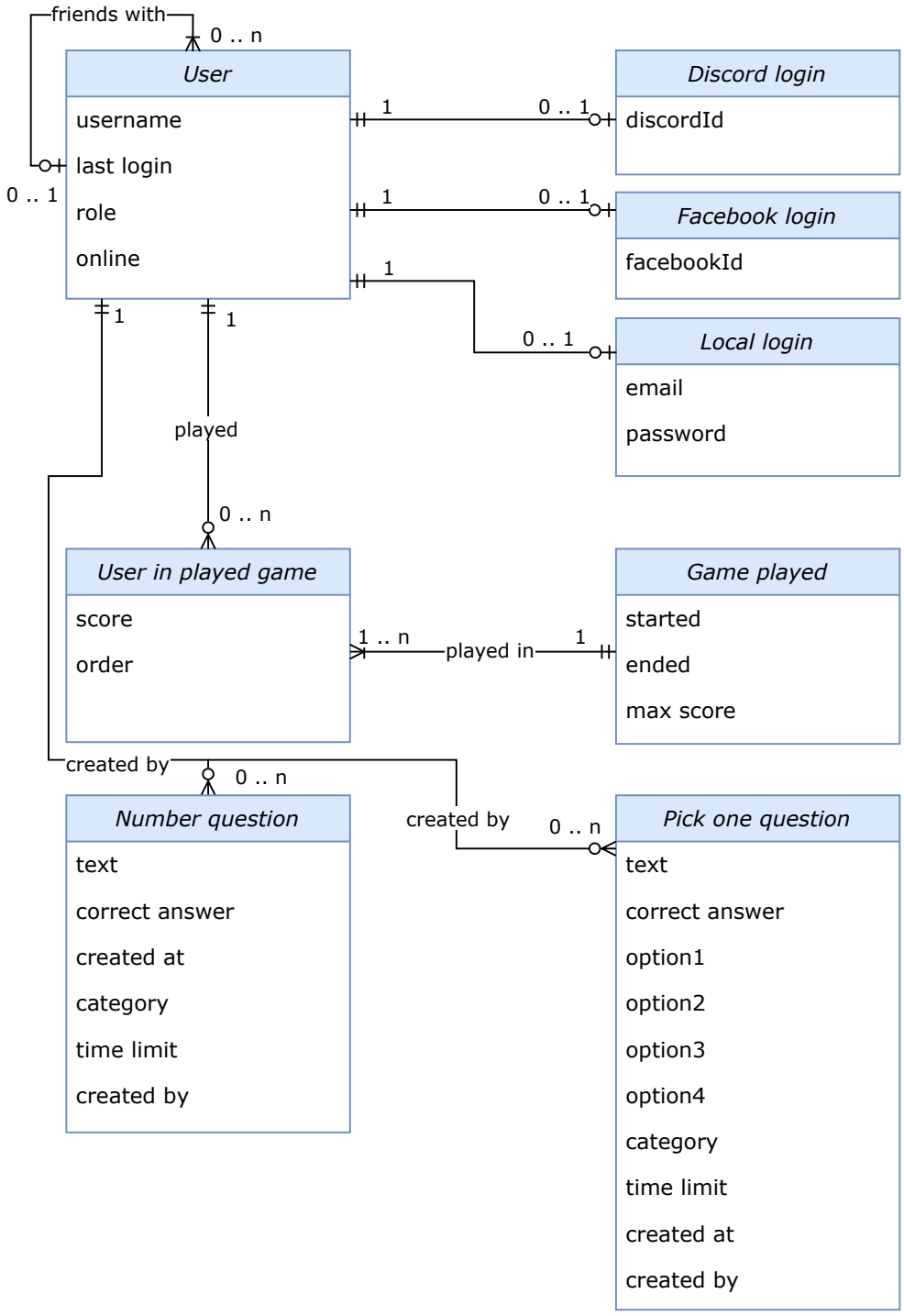
Další výhodou je nastavení jeho platnosti, většinou je doba platnosti nastavena na krátkou dobu a klient poté žádá o prodloužení pomocí obnovovacího (refresh) tokenu.

Nevýhodou je nemožnost zrušit nebo pozměnit token, dokud nevyprší jeho platnost. Problém tedy nastává, pokud je token odcizen nebo jinak kompromitován, řešením může být ukládat si tyto „zablokované“ tokeny někde na server, ale tím už bychom porušili jeho bezestavovost.

3.3 Doménový model

Klíčovou částí návrhu je doménový model, ten definuje jednotlivé entity, jejich atributy a vztahy mezi nimi. Může být zachycen například pomocí UML diagramu, diagramu tříd anebo entitně vztahového diagramu. Výhodou je, že se neváže na konkrétní programovací jazyk nebo platformu. V naší aplikaci ho využijeme k tvorbě entitních tříd, ze kterých pak ORM vytvoří databázový model.

Na obrázku 3.3 popisujeme doménový model pomocí UML diagramu.



Obrázek 3.3: Doménový model aplikace v UML notaci

3.4 Backend

Poté, co jsme přihlášeni a je navázáno spojení pomocí WebSocketů, potřebujeme vytvořit nějaký systém zpráv. V následující části si navrhneme události, jejich typ a data, které bude posílat ze serveru na klienty. WebSockety nabízejí dvě možnosti přenosu zprávy:

1. `socket.emit(zpráva, data)` – odešle se zpráva a neočekává se žádná odpověď

2. `socket.emitWithAck(zpráva, data)` – více podobné klasické request–response API, bude se čekat na odpověď druhé strany

První variantu využijeme, pokud hráče třeba jenom o něčem informujeme (např. někdo zabral území), druhou naopak pokud od hráče čekáme nějaký výstup, například odpověď na otázku. Výhodou je, že jí můžeme zkombinovat i s `timeout()` funkcí, tj. pokud hráč neodpoví do časového limitu na otázku, vyhodí se výjimka.

Události si rozdělíme do dvou tabulek podle toho, o kterou variantu se jedná.

Tabulka 3.1: Události s čekáním na odpověď

Typ zprávy	**Parametry**	**Parametry odpovědi**	**Popis**
selectRegion	socketid, seznam možných regionů	vybraný region	Vybrání regionu hráčem.
question	zadání otázky, čas. limit, autor, (informace o útoku)	odpověď	Tipovací otázka
pickOne	zadání otázky, možnosti, čas, informace o útoku	vybraná možnost	Otázka s jednou správnou odpovědí.

Tabulka 3.2: Události bez čekání

Typ zprávy	**Parametry**	**Popis**
players	pole hráčů	Pošle informace o hráčích na začátku hry.
base	id socketu, region key	Informace o bázi hráče.
updatePlayerHealth	id socketu, počet životů	Informace o změně počtu životů hráče.
question	zadání otázky, čas. limit, autor, (informace o útoku)	Informace o nové otázce.
answer	odpovědi ostatních hráčů, správná odpověď	Pošle odpovědi na otázky ostatních hráčů a správnou odpověď.
numberQuestionEnded		Konec otázky.
pickOne	zadání otázky, možnosti, čas, informace o útoku	Otázka s jednou správnou odpovědí.
endGame		Oznamuje konec hry.
selectRegionEnded		Oznamuje konec výběru regionu.
regionPlayerChanged	id socketu, region key, cena regionu	Hráč vlastnící region se změnil.
score	id socketu, skóre	Hráčovo skóre se změnilo.

3.5 Frontend

Stejně jako u backendu si v tabulce 3.3 ukážeme zprávy, které budeme odesílat. V další části se pak podíváme na návrh uživatelského rozhraní.

Tabulka 3.3: Události s čekáním na odpověď

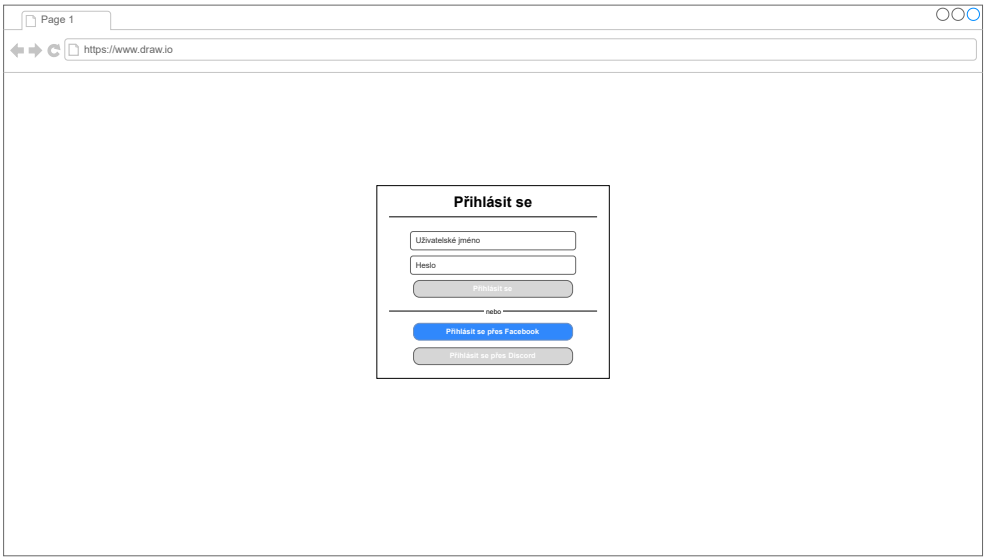
Typ zprávy	**Parametry**	**Parametry odpovědi**	**Popis**
getFriends		seznam přátel	Vrátí seznam přátel.
getFriendRequests		seznam žádostí	Vrátí seznam žádostí o přátelství.
sendFriendRequest	přezdívka hráče	úspěch/selhání	Odešle žádost o přátelství.
acceptFriendRequest	id uživatele	úspěch/selhání	Přijme žádost o přátelství.
declineFriendRequest	id uživatele	úspěch/selhání	Odmítne žádost o přátelství.
matchmaking	vypnout/zapnout		Vypne/zapne vyhledávání hry.

Wireframe

Wireframe nebo také drátěný model je typ návrhu uživatelského rozhraní a definuje základní kostru webu. Ukazuje, jak budou jednotlivé obrazovky/stránky vypadat a jak na nich budou komponenty rozloženy. Nejedná se o grafický návrh, a proto není nutné řešit konkrétní podobu prvků na stránce.

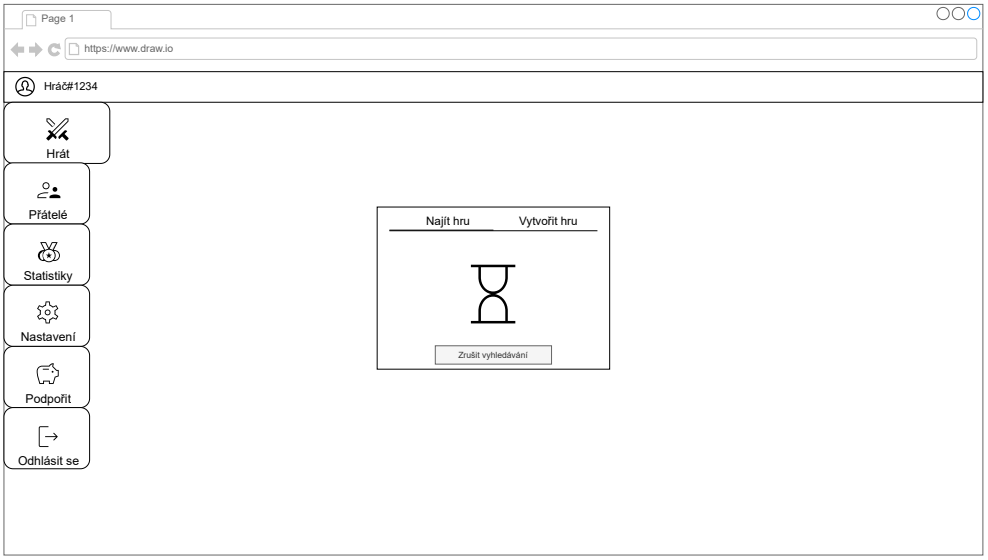
Návrh budeme provádět pouze pro desktopová zařízení, protože jak již bylo zmíněno dříve, pro mobilní zařízení to nedává vzhledem k existujícím aplikacím smysl.

Obrazovek je celkem 10, pokusil jsem se o minimalistické provedení s jednoduchými barvami, uvidíme, jak se nám to podaří zrealizovat v kapitole Implementace.



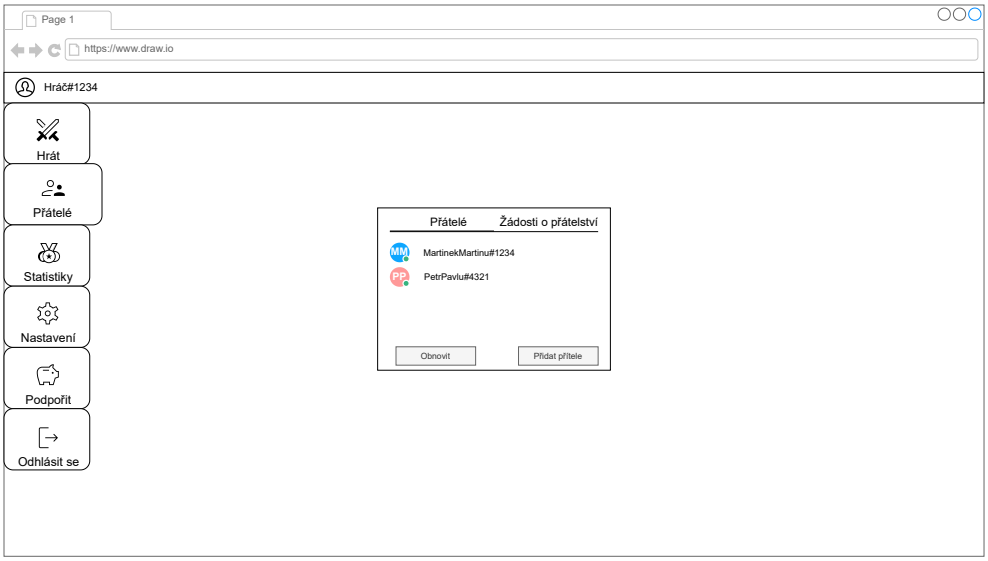
Obrázek 3.4: Přihlášení

Obrázek 3.5: Registrace

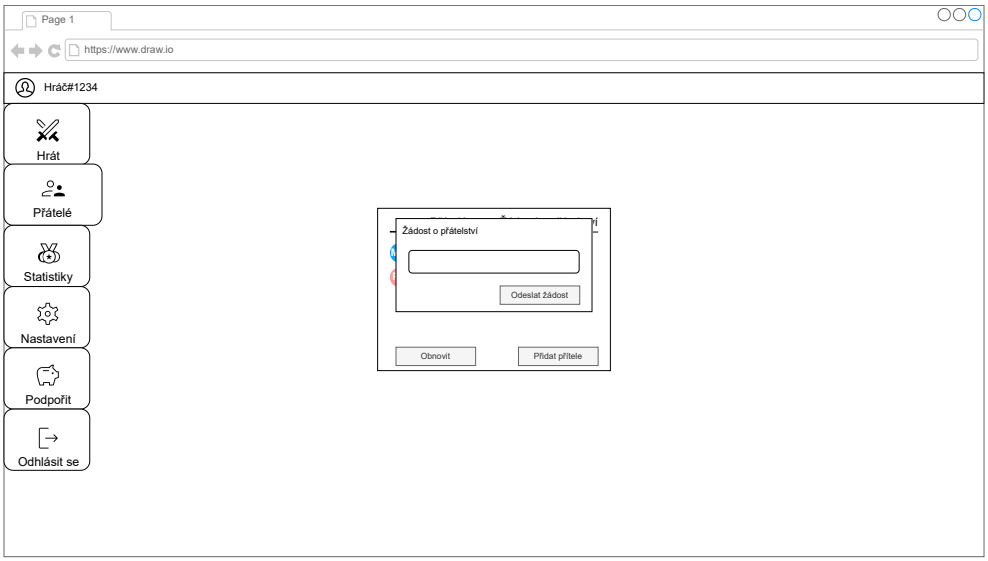


Obrázek 3.6: Vyhledávání hry

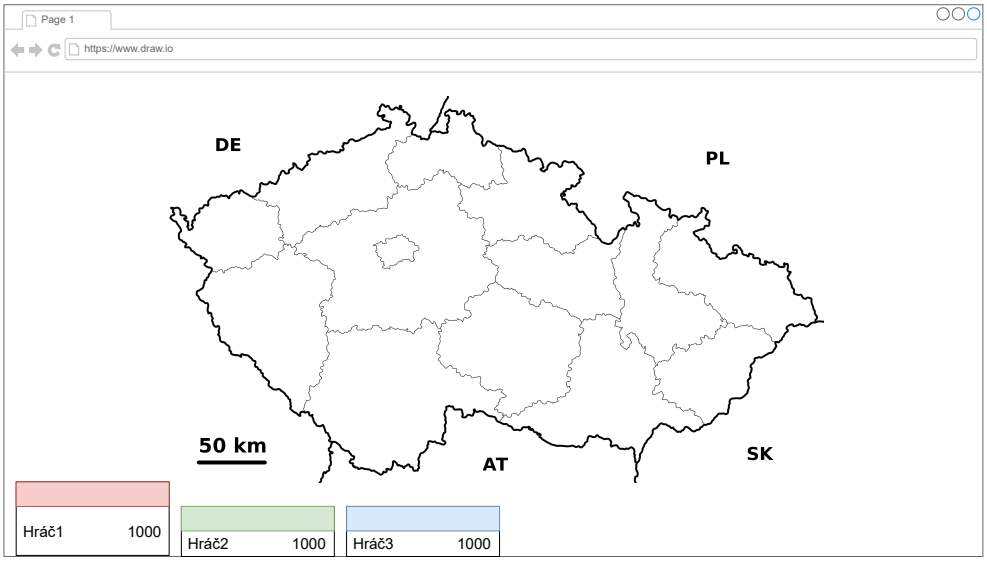
Obrázek 3.7: Odehrané hry



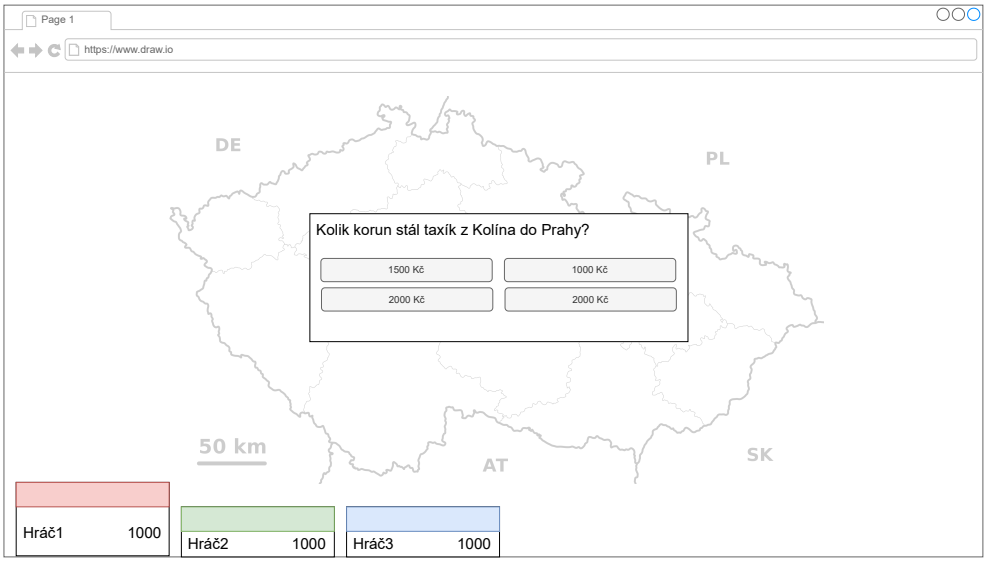
Obrázek 3.8: Seznam přátel



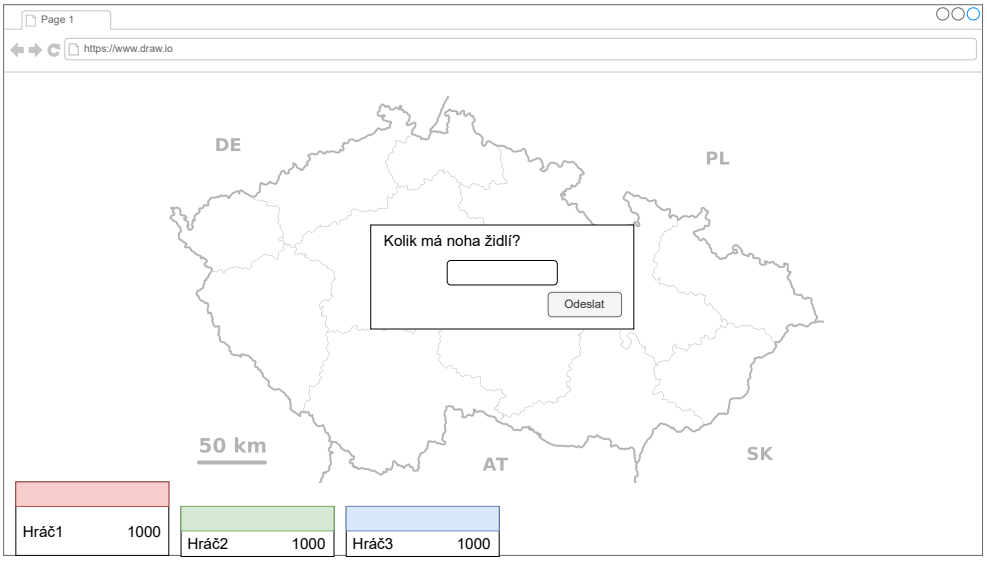
Obrázek 3.9: Odeslání žádosti o přátelství



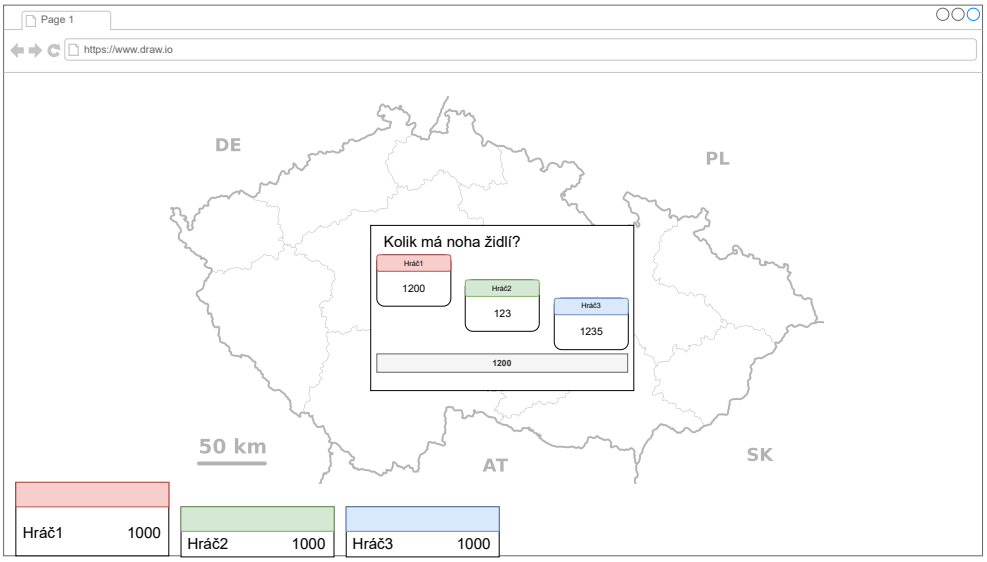
Obrázek 3.10: Herní mapa



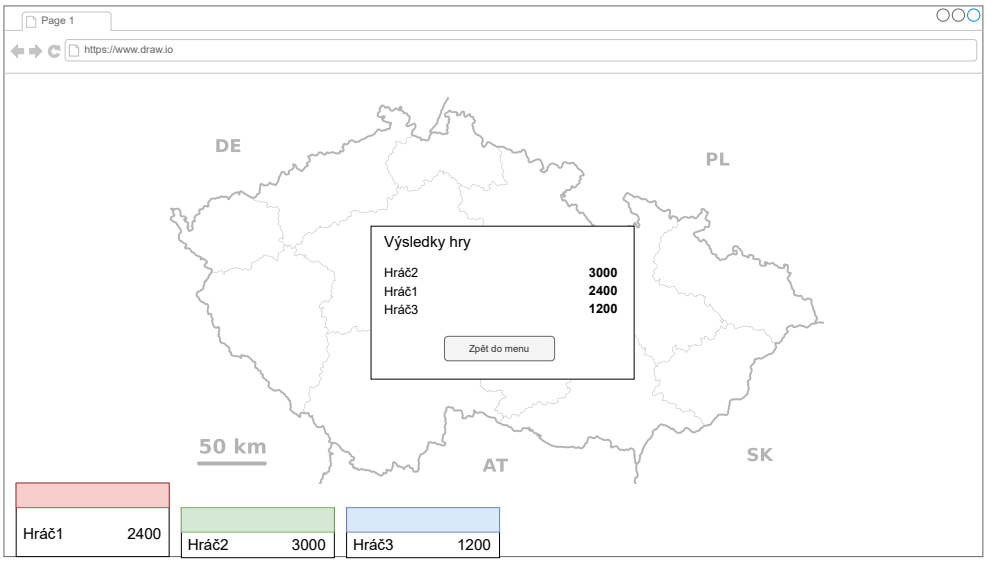
Obrázek 3.11: Otázka s jednou správnou odpovědí



Obrázek 3.12: Tipovací otázka



Obrázek 3.13: Výsledky tipovací otázky



Obrázek 3.14: Výsledky hry

Kapitola 4

Implementace

Další částí softwarového vývoje je implementace, zaměříme se zde na implementaci frontendu (klientské části) a backendu (serverové části). Výsledek bude dostupný v oddělených repozitářích na platformě Github (frontend, backend).

4.1 Backend

4.1.1 Tvorba entit

Prvním krokem bude vytvořit entity podle doménového modelu (viz 3.3). Protože používáme TypeORM, nebudeme tvořit klasické tabulky v SQL, ale pouze entitní třídy v Typescriptu. Ukážeme si to na třídě uživatele v ukázce 4.1.

Každý kód entity začíná anotací `@Entity`, začneme s proměnnou `id`, která má anotaci `@PrimaryGeneratedColumn`. To znamená, že se jedná o primární klíč a bude se automaticky inkrementovat při přidání nového záznamu.

Na dalších řádcích pak najdeme obyčejné proměnné. Na řádcích **18** a **22** najdeme 1:N vztahy pro odehrané hry hráče (`games`) a žádosti o přátelství (`friendRequests`).

Jak je vidět, anotacemi se opravdu nešetří. Pomáhají zpřehlednit kód, udělat ho čitelnějším a používají je hojně i ostatní backendové frameworky, NestJS není výjimkou. K anotacím si ještě přičichneme později, protože se používají opravdu všude, ale pro teď to bude stačit.

```typescript
@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    username: string;

    @CreateDateColumn()
    lastLogin: Date;

    @Column({default: 0})
    role: number;

    @Column('boolean', {default: false})
    online: boolean;

    @OneToMany(type => UserInGame,
               userInGame => userInGame.user)
    games: UserInGame[];

    @OneToMany(type => FriendRequest,
               friendRequest => friendRequest.toUser)
    friendRequests: FriendRequest[];
}
```

Obrázek 4.1: Ukázka kódu – Implementace `user.entity.ts`

4.1.2 Přihlášení a JWT tokeny

Další klíčovou věcí, bez níž by aplikace nemohla fungovat, je přihlášení. S tím nám částečně pomůže knihovna `@nestjs/passport`, která rozšiřuje knihovnu Passport.js. Jedná se o jednu z nejpopulárnějších knihoven pro přihlašování, podporuje spoustu poskytovatelů a sjednocuje volání do jednotného formátu.

V NestJS jsou dvě klíčové věci, které souvisí s implementací přihlašování, jsou to:

1. **Strategie** – pro každého poskytovatele máme jednu strategii, musí implementovat metodu `validate()`

2. **AuthGuard** – volají strategie a pomocí nich nastavujeme do kterých URL adres (route) může uživatel přistupovat

Ukážeme si to na implementaci přihlašování přes Facebook. Budeme si muset nainstalovat balíček `passport-facebook` a zaregistrovat se jako vývojář na Facebook Developers. V administraci si necháme vygenerovat **App ID** a **App Secret** a vytvoříme si soubor `facebook.strategy.ts`. V něm vytvoříme třídu FacebookStrategy, která rozšiřuje PassportStrategy, viz ukázka 4.2.

V konstruktoru zavoláme konstruktor rodičovské třídy pomocí `super()` a předáme mu objekt obsahující **App ID**, **App Secret**, **callbackURL** a **profileFields**, což jsou data uživatele, která chceme získat. **CallbackURL** je pak, jak název napovídá, URL adresa na kterou nás Facebook, po úspěšném přihlášení, přesměruje.

```
@Injectable()
export class FacebookStrategy
    extends PassportStrategy(Strategy, 'facebook') {
  constructor() {
    super({
      clientID: process.env.FB_ID,
      clientSecret: process.env.FB_SECRET,
      callbackURL: process.env.FB_CALLBACK,
      profileFields: ["email", "id", "displayName"]
    });
  }

  async validate(
     accessToken: string,
     refreshToken: string,
     profile: Profile,
     done: (err: any, user: any, info?: any) => void
     ) {
    const payload: PayloadUser = {
        name: profile.displayName,
        id: profile.id
    };
    if (profile.emails != null)
        payload.email = profile.emails[0];

    done(null, payload);
  }
}
```

Obrázek 4.2: Ukázka kódu – Implementace Facebook strategie

Na řádcích **13** až **27** pak z objektu, který dostaneme, vyfiltrujeme to, co nás zajímá a voláním funkce `done()` předáme výsledný objekt. Tento objekt pak bude součástí každého požadavku, kde strategii používáme.

Dále si vytvoříme kontrolér pomocí příkazu `nest g co facebook-auth` a přidáme do něj metodu `facebookLoginRedirect()`, která vypadá následovně:

```
1  @Get("/facebook/redirect")
2  @UseGuards(AuthGuard("facebook"))
3  async facebookLoginRedirect(
4          @Req() req: Request, @Res() res: Response
5      ): Promise<any> {
6      if (req["user"] == null) {
7          res.status(401).send();
8          return;
9      }
10
11     const reqUser: PayloadUser = (req.user as PayloadUser);
12     const u = await this.userService.getUserByFacebook(reqUser)
13
14     const jwt = this.userService.getJwt({id: u.id});
15     res.cookie("jwt", jwt.access_token);
16     res.redirect(`${process.env.BASE_URL}/fblogin`);
17     res.status(200).send();
18 }
```

Obrázek 4.3: Ukázka kódu – Implementace přihlášení pomocí Facebooku

Na prvním řádku máme anotaci `@Get()`, ta specifikuje která HTTP metoda se má použít pro volání této funkce a na jaké adrese je dostupná. Na druhém řádku máme pak použití AuthGuardu, zde logicky použijeme AuthGuard pro Facebook, protože chceme aby se nám zavolala `FacebookStrategy`.

Na vstupu funkce dostaneme objekty Request – požadavek a Response – odpověď, požadavek už by měl obsahovat proměnnou `user`, pokud došlo k úspěšnému přihlášení. Na řádce **6** si to ověříme, jinak vrátíme HTTP status kód `401 Unauthorized`, což znamená, že autorizace neproběhla úspěšně.

Na řádce **12** pak saháme do databáze a hledáme uživatele, který má tento Facebook identifikátor. Tato funkce je poměrně dlouhá, protože pokud není nalezen, musíme vytvořit nový účet a nový Facebook login, ale tím nebudu čtenáře dále zatěžovat.

Dále na řádce **15** si necháme vygenerovat nový JWT token, který předáme uživateli v cookie `jwt` a přesměrujeme ho na `/fblogin`, kde se už na frontendu postaráme o zpracování cookie a její uložení. Odhlášení už na straně serveru

řešit nebudeme, v kapitole Návrh jsem vysvětloval, že není možné JWT token zneplatnit. Budeme to řešit tak, že ho na frontendu zapomeneme.

4.1.3 Přátelé a žádosti o přátelství

Na backendu si připravíme čtyři metody – `getFriends()`, `getFriendRequests()`, `acceptFriendRequest()` a `declineFriendRequest()`.

Přátelství budeme reprezentovat pomocnou entitou `friends`, která bude mít dvě proměnné `fromUserId` a `toUserId`. Abychom se vyhnuli dvojnásobnému počtu záznamů, budeme brát že i jednostranné přátelství je přátelství.

Bohužel jsem narazil na limity TypeORM, protože nepodporuje SQL operaci UNION, abych mohl spojit výsledky dvou dotazů dohromady. Dotaz v metodě `getFriends()` budu muset tedy napsat ručně. Zároveň si ale musíme dát pozor na SQL Injection, to uděláme tak, že použijeme `prepared statements`. To jsou ty otazníky na řádcích **4** a **6**, které se pak nahradí hodnotami z pole na řádce **7**.

```
1  async getFriends(userId: number) {
2      const res = await this.dataSource.manager.query(`
3      SELECT id,username,online FROM user WHERE id in (
4      SELECT fromUserId as f FROM friend where toUserId = "?"
5      UNION
6      SELECT toUserId  as f FROM friend where fromUserId = "?")`
7      , [userId, userId]);
8
9      return res;
10 }
```

Obrázek 4.4: Ukázka kódu – `getFriends()`

Se žádostmi o přátelství už takový problém nebude, protože se jedná už z principu o jednostranný vztah. K dotazu už použijeme TypeORM a jeho `QueryBuilder`.

```
async getFriendRequests(userId: number) {
    const res = await this.dataSource.getRepository(User)
    .createQueryBuilder("user")
    .leftJoinAndSelect(FriendRequest, "friendrequest",
        "user.id = friendrequest.fromUserId")
    .select(["user.id", "user.username"])
    .where("friendrequest.toUserId = :userId",{userId: userId})
    .getMany();
    return res;
}
```

Obrázek 4.5: Ukázka kódu – `getFriendRequests()`

Tímto máme implementaci pro fungování přátel hotovou, metody pro přijetí/odmítnutí žádosti už jsou pak snadné na implementaci. Pouze vložíme dané záznamy do databáze.

4.1.4 Matchmaking

Předtím než můžeme přejít k samotné herní smyčce, musíme ještě uživatele nějak spojit. Do `game-manager.gateway.ts` přidáme metodu `changeMatchmakingStatus()`.

```
@SubscribeMessage('matchmaking')
async changeMatchmakingStatus(
        @MessageBody() status: boolean,
        @ConnectedSocket() socket: Socket
    ) {
    if (status)
        await this.gameManagerService.addToMatchmaking(socket);
    else
        this.gameManagerService.removeFromMatchmaking(socket);
}
```

Obrázek 4.6: Ukázka kódu – Matchmaking

Vytvoříme si set hráčů, kteří právě vyhledávají hru, abychom měli přidávání a mazání v konstantním čase. Funkce `addToMatchmaking()` pak bude vypadat následovně (viz 4.7).

Na začátku zkontrolujeme, že už hráč není v jiné hře, poté ho přidáme do setu. Pokud aktuálně 3 hráči vyhledávají hru, vygenerujeme náhodný identifikátor a hru vložíme do mapy.

Hráče pak ze setu matchmakingu odstraníme, nastavíme jim aktivní hru na tuto a na řádce **24** počkáme, dokud hra neskončí. Výsledky hry pak uložíme do databáze.

```
1  async addToMatchmaking(socket: Socket) {
2      const pl = this.getPlayer(socket.id);
3      if (pl.activeGame)
4          throw new Error("Hráč už je v jiné hře");
5
6      this.matchmakingPlayers.add(pl);
7
8      if (this.matchmakingPlayers.size != 3)
9          return;
10
11     let gameId = this.generateRandomString();
12     while (this.games.has(gameId)) {
13         gameId = this.generateRandomString();
14     }
15     const g = new Game(gameId, null, this.q);
16     this.games.set(gameId, g);
17
18     for (let p of this.matchmakingPlayers) {
19         this.matchmakingPlayers.delete(p);
20         p.activeGame = this.games.get(gameId);
21         g.players.set(p.socket.id,p);
22     }
23
24     const resp: GameFinished = await g.start();
25     this.g.addPlayedGame(resp);
26     for (let p of g.players) {
27         p[1].activeGame = null;
28     }
29     this.games.delete(gameId);
30 }
```

Obrázek 4.7: Ukázka kódu – `addToMatchmaking()`

4.1.5 Herní smyčka

Posledním krokem na backendu a vlastně tím nejdůležitějším je samotný průběh hry a její smyčka. Pro tu bude sloužit třída `game.ts`, která bude reprezentovat jednu instanci hry.

Celá hra je obsažena v metodě `start()` a volá se po vytvoření hry, jak jsme si ukazovali v ukázce 4.7.

Nejprve je hráčům odeslána informace o ostatních hráčích a poté vygenerujeme každému hráči náhodný region a přiřadíme ho jako jejich základnu.

Hra je rozdělena do dvou fází – dobývání a zabírání.

4.1.5.1 První fáze – dobývání

V této fázi budeme hráčům pokládat tipovací otázky do té doby, než budou rozebrány všechny regiony na mapě. Po zodpovězení otázky se porovná, kdo byl svým tipem nejblíže ke správné odpovědi, a pokud má více hráčů stejnou odpověď, lepší je ten, kdo má lepší čas. První hráč si vybere dva regiony na mapě, druhý jeden region a poslední žádný. Hráč si může vybrat pouze region sousedící s regiony, které už má. Pokud by nebyl žádný sousedící volný, může si vybrat jakýkoliv.

```
for (let i = 0; i < 4; i++) {
    const a: NumberQuestionAnswer[] = await this.question();
    const reg1 = await this.getPlayerToSelectRegion(
        this.players.get(a[0].key), RegionSelectType.Nearby);
    this.changePlayerRegion(this.players.get(a[0].key),reg1);

    await this.wait(1);

    const reg1_2 = await this.getPlayerToSelectRegion(
        this.players.get(a[0].key), RegionSelectType.Nearby);
    this.changePlayerRegion(this.players.get(a[0].key),reg1_2);

    await this.wait(1);

    const reg2 = await this.getPlayerToSelectRegion(
        this.players.get(a[1].key), RegionSelectType.Nearby);
    this.changePlayerRegion(this.players.get(a[1].key),reg2);

    await this.wait(2);
}
```

Obrázek 4.8: Ukázka kódu – první fáze (dobývání)

Regionů je celkem 15, aby to bylo dobře dělitelné třemi. Po rozdělení regionů už jich je volných pouze 12. Bude nám tedy stačit provést volání čtyřikrát, jak je vidět na prvním řádku v ukázce 4.8.

V každé iteraci nejdříve zadáme otázku, to dělá funkce `question()`, z výsledků pak vybereme prvního hráče a necháme ho vybrat dva regiony, druhého hráče pak necháme vybrat jeden region. Ještě si můžeme všimnout častého volání funkce `wait()`, používáme ji k tomu, aby se stihly přehrát animace na klientech a rychlost hry byla konzistentní. Nejedná se o klasické blokující volání, kde se uspí celé vlákno, které můžeme znát z jiných jazyků. Díky tomu, jak NodeJS funguje, bude mezitím obsluhovat další požadavky.

4.1.5.2 Druhá fáze – zabírání

Druhá fáze spočívá v útočení hráčů na sebe, bude opět opakována čtyřikrát a v každé fázi bude iterováno přes všechny hráče. Hráče, který je na řadě, necháme vybrat region protihráče. Využijeme pro to volání funkce `getPlayerToSelectRegion`, tentokrát s parametrem `EnemyNearby`.

Funkce `pickOne()` už se postará o zadání otázky a pokud oba dva hráči odpoví správně, rozhodne o vítězi útoku ještě tipovací otázka.

```
for (let i = 0; i < 4; i++) {
    for (let p of this.players) {
        // zkontrolujeme, že hráč není mrtev
        if (!p.isAlive())
            continue;
        const reg = await this.getPlayerToSelectRegion(
            p, RegionSelectType.EnemyNearby);
        await this.pickOneQuestion(
            p, this.map.getRegion(reg).player, reg);
        await this.wait(1.5);
    }
}
```

Obrázek 4.9: Ukázka kódu – druhá fáze (zabírání)

4.2 Frontend

4.2.1 JWT token

První věcí, na kterou se zaměříme, je přihlášení, většinu už jsme si naštěstí odbyli na backendu. Naším úkolem už bude jenom získaný JWT token z cookie uložit do lokálního úložiště, abychom ho neztratili po znovu–načtení stránky.

Budeme tedy potřebovat vytvořit novou komponentu, na kterou se bude přesměrovávat z backendu po úspěšném přihlášení. Pojmenujeme ji `AfterLoginComponent` a na ukázce 4.10 je vidět její konstruktor.

```
constructor(private store: Store<AuthState>) {
    const cookie = this.getCookieValue("jwt");
    this.deleteCookie("jwt");
    if (localStorage.getItem("jwt") == null && cookie != "")
        this.store.dispatch(setJwt({jwt: cookie}))
}
```

Obrázek 4.10: Ukázka kódu – uložení JWT tokenu

Token si také uložíme do `Storu`, abychom ho nemuseli pořád vytahovat z lokálního úložiště. Pak už ho budeme předávat v hlavičce `Authorization` (viz ukázka 4.11) a vše bude fungovat tak jak má.

```
const socketOptions = {
    transportOptions: {
        polling: {
            extraHeaders: {
                Authorization: jwt
            }
        }
    }, secure: true
};
this.socket = io(environment.ioUri + "/game",socketOptions);
```

Obrázek 4.11: Ukázka kódu – předání JWT tokenu

4.2.2 Směrování

Směrování je proces, který podle daných pravidel a URL adres zobrazuje různé komponenty v aplikaci. Angular má pro to tyto případy knihovnu `AppRoutingModule`, kterou využijeme. Směrování v Angularu má tu výhodu, že se po přechodu na jinou adresu nemusí stránka celá přenačítat. Celý průchod aplikací je tak pro uživatele mnohem rychlejší, plynulejší a nevytváří tak velkou zátěž na server.

Seznam pravidel a URL adres definujeme datovým typem `Routes` a výsledný seznam je v ukázce 4.12.

V ukázce si můžeme všimnou použití `canActivate`, tím říkáme, že na danou URL adresu je možné se dostat pouze po splnění nějaké podmínky. V našem případě je tato podmínka být přihlášen. Pravidla je možné také dále zanořovat do sebe, toho využijeme abychom měli sjednocenou grafiku v sekci po přihlášení.

```
const routes: Routes = [
    { path: 'hra', component: EmptyComponent,
        canActivate: [AuthguardService] },
    { path: 'fblogin', component: FbloginComponent},
    { path: '', component: MenuComponent,
        canActivate: [AuthguardService], children: [
        { path: '', redirectTo: "hrat", pathMatch: "full"},
        { path: 'hrat', component: PlayComponent },
        { path: 'hrat/:id', component: LobbyComponent },
        { path: 'pratele', component: FriendsComponent },
        { path: 'statistiky', component: StatsComponent },
        { path: 'nastaveni', component: SettingsComponent },
        { path: 'podporit', component: DonateComponent },
    ] },
    { path: "", component: UnauthComponent, children: [
        { path: '', redirectTo: "login", pathMatch: "full"},
        { path: "login", component: LoginComponent},
        { path: 'registrace', component: RegisterComponent }
    ]},
    { path: "**", redirectTo: "hrat"},
];
```

Obrázek 4.12: Ukázka kódu – seznam pravidel pro směrování

Na posledním řádku je pak vidět použití `"**"` (tzv. wildcardu), ten je zavolán v případě, že žádné pravidlo nebylo splněno. V našem případě přesměrujeme uživatele na hlavní obrazovku pro hledání hry. Popřípadě bychom mohli zobrazit i stránku 404 – Stránka nenalezena.

4.2.3 NgRx a Store

V analýze použitých technologií jsem dost vyzdvihoval knihovnu NgRx pro Angular, která dělá kód aplikace přehlednější a snadnější na rozšíření, tak se na to pojďme podívat.

V této části se zaměříme na tvorbu storu pro funkcionalitu autentizace, projdeme si celým procesem tvorby akcí, reducerů a storu, který jsme zmiňovali v kapitole Návrh.

Prvním krokem bude vytvořit si **akce**, které budou reprezentovat změnu stavu aplikace. V ukázce 4.13 je obsah souboru `auth.actions.ts`.

Na posledním řádku máme přidaný i **selektor**, kterým pak budeme z celého storu vybírat právě `AuthState`.

Následně si vytvoříme soubor `auth.state.ts` v ukázce 4.14, což bude rozhraní definující strukturu tohoto stavu.

```
export const facebookLogin = createAction(
    '[Login component] Facebook login initiated');
export const discordLogin = createAction(
    '[Login component] Discord login initiated');

export const localLoginInitiated = createAction(
    '[Login component] Local login initiated',
    props<{username: string, password: string}>());
export const localLoginResponseError = createAction(
    '[Auth effects] Local login response error',
    props<{message: string, statusCode: number}>());
export const localLoginSuccess = createAction(
    '[Auth effects] Local login success');
export const logout = createAction(
    '[Login component] Logout initiated');
export const setJwt = createAction(
    '[Login Effect] JWT token set',
    props<{jwt: string}>());

export const selectAuthState =
    createFeatureSelector<AuthState>("auth");
```

Obrázek 4.13: Ukázka kódu – `auth.actions.ts`

```
export interface AuthState {
    user?: User;
    isLoading: boolean;
    jwt?: string;
    msg?: {
        message: string;
        statusCode: number;
    };
}
```

Obrázek 4.14: Ukázka kódu – `auth.state.ts`

Otazníky za názvy proměnných znamenají, že proměnná není povinná. To dává smysl, protože `user`, `jwt` a `msg` nebude před přihlášením ještě existovat a kompilátor Typescriptu by nás navíc nenechal pokračovat, pokud by tam otazníky nebyly.

Další částí je **reducer**, ten mění stav na základě akcí, které přijdou.

V ukázce 4.15 si musíme nejdříve nadefinovat výchozí stav – `initialState`, což je první argument funkce `createReducer()`. Zde je to velmi jednoduché, povinná hodnota je pouze `isLoading`, kterou nastavíme na `false`.

```
const initialState = {isLoading: false}

export const authReducer = createReducer(
    initialState,
    on(setJwt,
        (state, {jwt}) => {
            return {...state,
                jwt: jwt, isLoading: false, msg: null};
    }),
    on(localLoginInitiated,
        (state) => {
            return {...state, isLoading: true};
    }),
    on(logout, // při odhlášení vrátíme stav na původní
        (state)=>{
            return initialState;
    })
...
```

Obrázek 4.15: Ukázka kódu – `auth.reducer.ts`

Dalšími argumenty jsou pak listenery na akce, které vrací změnu stavu. Jak jsme zmiňovali dříve, není možné měnit stav napřímo, protože je `immutable`. Musíme si vytvořit kopii, tu upravit a předat.

Posledními jsou **efekty**, ty naslouchají na akce a volají se potom, co doběhnou reducery. Ukážeme si to na dvou efektech z `auth.effects.ts` v ukázce 4.16.

```
...
 onLogout$ = createEffect(()=> this.actions$.pipe(
    ofType(logout),
    tap(()=>{
        localStorage.removeItem("jwt");
        this.game.disconnect();
        this.router.navigate(['/login']);
    })
), {dispatch: false});

onJwtTokenSet$ = createEffect(() => this.actions$.pipe(
    ofType(setJwt),
    tap(action => {
        if (localStorage.getItem("jwt") == null) {
            localStorage.setItem("jwt", action.jwt);
            this.router.navigate(['/']);
        }
        this.game.connect(action.jwt);
    })
), {dispatch: false});
...
```

Obrázek 4.16: Ukázka kódu – `auth.effects.ts`

První efekt reaguje na akci `logout`, ta značí, že se uživatel odhlásil. Jako první odstraníme JWT token z lokálního úložiště, pak se odpojíme ze serveru a uživatele přesměruje zpátky na přihlašovací obrazovku.

Druhým efektem je reakce na nastavení JWT tokenu. Nejdříve zkontrolujeme, že JWT token není nastaven, pak ho uložíme, přesměrujeme uživatele na správnou obrazovku a připojíme ho k serveru.

Možnost `{dispatch: false}` pak znamená to, že se nebude akce volat znovu.

Tím máme store pro autentizaci hotov, ostatními story tu nebudu čtenáře už zatěžovat.

4.2.4 Herní mapa

Jsou dva způsoby, jak reprezentovat herní mapu, prvním je použít HTML element `<canvas>`, druhým je použít `<svg>`. S canvasem je problém ten, že pokud chceme detekovat na jaký region uživatel klikl, museli bychom si napsat vlastní detekci kolize polygonů. Problém by taky nastal, pokud bychom chtěli použít nějaké efekty nebo animace, opět bychom si museli všechno napsat sami. V případě SVG je výhoda to, že se jedná o běžný HTML element

a fungují zde všechny CSS vlastnosti, které známe z ostatních prvků. Menší nevýhodou SVG může být slabší výkon oproti canvasu, ale jelikož je naše hra nenáročná, nemusí nás to vůbec trápit.

4.2.4.1 Vybírání regionu

Důležitou funkcí, kterou musíme naimplementovat, je volení regionu. Budeme to potřebovat hned v první fázi hry, kdy si hráči volí regiony na mapě, které chtějí zabrat. Dále pak ještě ve fázi hry, kdy útočí jeden na druhého.

V návrhu jsme si pro to zavedli zprávu `selectRegion`, ve které nám server pošle seznam regionů, které můžeme aktuálně zabrat. Naším úkolem bude to na frontendu zobrazit a vybraný region poslat zpátky.

Začneme tedy se zobrazením na mapě, ideální by bylo regiony, které hráč nemůže zabrat, přeškrtat. Bohužel HTML elementy `<path>`, které používáme k zobrazení regionů, nemohou mít na sobě více vrstev.

Nikdo nám ale nezakazuje mít těchto regionů, které vypadají stejně, více. Toho využijeme, abychom přes region zobrazili přeškrtání. Nejprve si v našem SVG s mapou definujeme novou šablonu, která bude přeškrtaná (viz ukázka 4.17).

```
<svg>
...
<pattern id="diagonalHatch" width="10" height="10"
    patternTransform="rotate(45 0 0)"
    patternUnits="userSpaceOnUse">
    <line x1="0" y1="0" x2="0"
    y2="10" style="stroke:black; stroke-width:1" />
</pattern>
...
</svg>
```

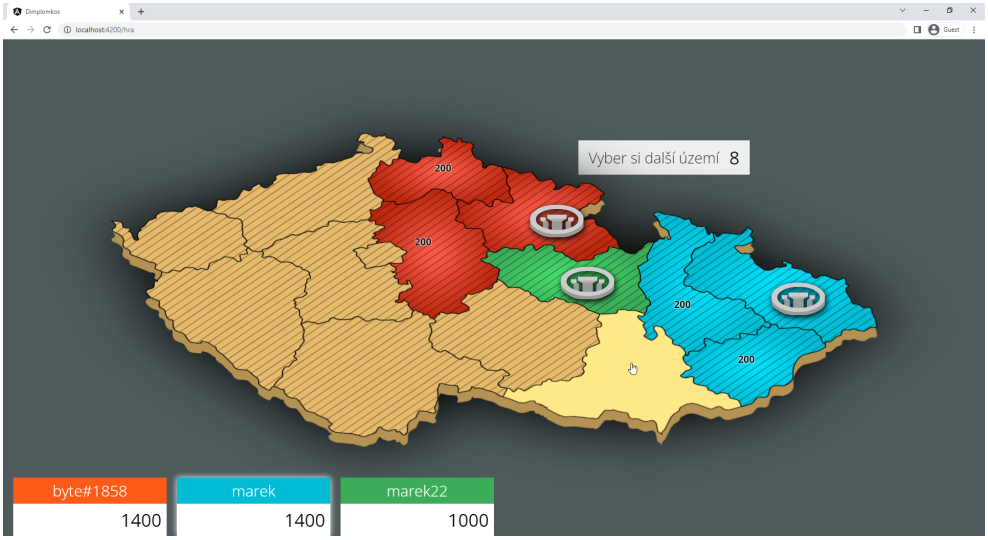
Obrázek 4.17: Ukázka kódu – přeškrtaná šablona

V kódu na ukázce 4.18 máme iteraci přes všechny regiony. K normálnímu vykreslení regionu přidáme ještě další `<path>` element – ten se bude zobrazovat v případě, že má být region zaškrtán a nedostupný k vybírání.

```
<ng-container *ngFor="let reg of regions | keyvalue:asIsOrder">
    ...
    <path *ngIf="((pickRegion.player.me
    && !pickRegion?.regions?.has(reg.value.name))
    || !pickRegion.player.me)" [attr.name]="reg.value.name"
    class="map--hatched" ></path>
</ng-container>
```

Obrázek 4.18: Ukázka kódu – zobrazení přeškrtaného regionu

Výsledné vybírání regionu vidíme na obrázku 4.19.



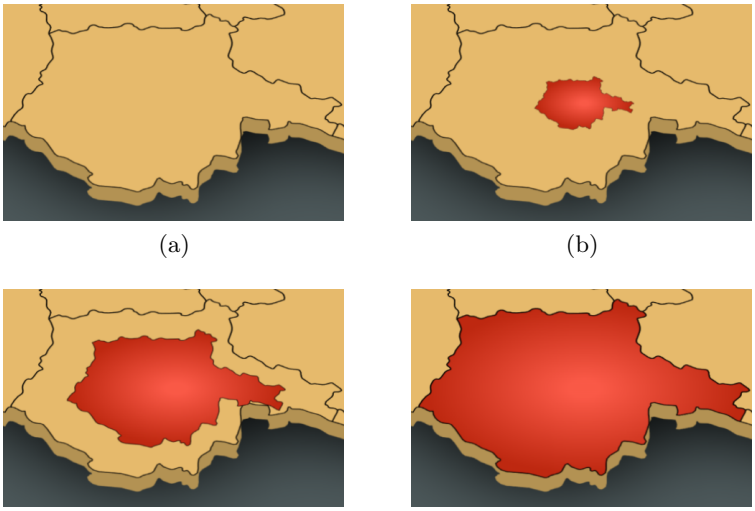
Obrázek 4.19: Zabírání regionu

4.2.4.2 Animace

Region bude na mapě zabarven podle toho, jaký hráč ho vlastní. Cílem je udělat to stejně jako v původním Dobyvateli, pro ukázku jsem si připravil čtyři obrázky (ukázka 4.20), které ukazují to, jak to má vypadat.

Použijeme k tomu balíček `@angular/animation`, který obaluje klasické CSS animace a umožňuje nám mnohem lehčí práci s nimi. Animace můžeme snadno ovládat pomocí změn proměnných nebo na základě uživatelského vstupu.

Ve hře budou aktuálně 3 barvy – červená, zelená a modrá. Nejdříve si v ukázce 4.21 zadefinujeme základní stavy pro každou barvu.



(a) (b)

(c) (d)

Obrázek 4.20: Animace zabírání regionu

```
state('red',
    style({ fill: "url(#red-color)"})
),
state('blue',
    style({ fill: "url(#blue-color)"})
),
state('green',
    style({ fill: "url(#green-color)"})
),
```

Obrázek 4.21: Ukázka kódu – definice stavů v `map.animations.ts`

Dále musíme zadefinovat přechody pro základní stav, kdy region ještě není obsazen žádným hráčem. Ke každému přechodu definujeme ještě `keyframes` (klíčové snímky), kde nastavíme, jak se budou jednotlivé styly měnit (ukázka 4.22).

```
...
transition('void => red', [
    animation([
        style({ fill: "url(#red-color)" }),
        animate('1s ease-in-out', keyframes([
            style({ transform: 'scale(0)', offset: 0 }),
            style({ transform: 'scale(1)',
                    fill: "url(#red-color)",
                    offset: 1 })
        ])),
        style({ fill: "url(#red-color)" })
    ]),
]),
...
```

Obrázek 4.22: Ukázka kódu – přechod z „prázdné“ barvy na červenou

Posledním krokem bude v ukázce 4.23 nastavit přechody mezi barvami hráčů a máme hotovo.

```
...
transition('red => blue', [
    animation([
        animate('2s ease-in-out', keyframes([
            style({ transform: 'scale(1)',
                    fill: "url(#red-color)",
                    offset: 0 }),
            style({ transform: 'scale(0)',
                    offset: 0.49 }),
            style({ transform: 'scale(0)',
                    fill: "url(#blue-color)",
                    offset: 0.5 }),
            style({ transform: 'scale(1)',
                    fill: "url(#blue-color)",
                    offset: 1 })
        ])),
    ]),
]),
...
```

Obrázek 4.23: Ukázka kódu – přechod z červené barvy na modrou

4.2.5 Tipovací otázka

Podíváme se dále na tipovací otázku, budeme ji ve stavu se hrou reprezentovat tímto objektem:

```
export interface NumberQuestion  {
    state: "none"|"answering"|"waiting_for_answer"|"results";
    answers: QuestionAnswer[];
    text: string;
    sentBy: string;
    targetDate: Date;
    callback: Function;
    correctAnswer?: number;
    from?: Player;
    to?: Player;
    isBaseAttack?: boolean;
    isSpectating: boolean;
    type: "phase1" | "phase2";
}
```

Obrázek 4.24: Ukázka kódu – reprezentace tipovací otázky v `app.state.ts`

Tento objekt pokryje jak variantu první, kdy se na začátku hry tipuje o území, tak druhou variantu, kdy na sebe hráči útočí a otázku s jednou odpovědí měli oba správně.

Celý proces bude (zjednodušeně) vypadat takto:

1. Server pošle zadání otázky
2. Zobrazíme ji hráči
3. Hráčovu odpověď odešleme na server
4. Potom, co mají všichni zodpovězeno nebo vypršel časový limit, pošle server odpovědi ostatních hráčů společně se správnou odpovědí
5. Výsledky zobrazíme

V této části se zaměříme na zobrazení výsledků. Cílem je zobrazit výsledky hráčů a zanimovat je tak, jako na stupni vítězů (viz 4.29).

Zobrazení výsledků šablony `question-dialog.component.html` si ukážeme v ukázce 4.25.

```
1  <div class="eval__pl" *ngFor="let p of question.answers"
2  [class.first]="p.order == 0"
3  [class.last]="p.order == question.answers.length - 1">
4      <div class="eval__name"
5          [ngClass]="'eval__pl--' + p.player.color">
6              {{p.player.name}}
7      </div>
8      <div class="eval__player-box">
9          <div class="eval__answer">
10             {{p.val != null ? p.val : '-'}}
11         </div>
12         <div class="eval__time">
13             {{p.time != null ?
14             (p.time/1000).toFixed(2) + 's' : "-"}}
15         </div>
16     </div>
17 </div>
```

Obrázek 4.25: Ukázka kódu – zobrazení výsledků hráčů tipovací otázky

Na prvním řádku můžeme vidět direktivu `*ngFor`, která slouží k iteraci přes kolekci dat, v našem případě přes odpovědi hráčů.

Na pátém řádku používáme direktivu `[ngClass]`, která dynamicky přidává třídu k HTML objektu. My ji použijeme k zabarvení podle barvy hráče.

Na řádku **4**, **8** a **11** si pak můžete všimnout dvojitých složených závorek, jedná se o tzv. **interpolaci**. Slouží k zobrazení dat a ke zpracování kódu uvnitř HTML šablony.

Na čtrnáctém řádku převedeme hráčův čas z milisekund na sekundy a zaokrouhlíme ho na dvě desetinná místa.

Odpočet

Na začátku této sekce jsme si ukazovali objekt, kterým je otázka reprezentována. Důležitá bude pro nás hodnota proměnné `targetDate`, která obsahuje časovou značku, do kdy je možné na otázku odpovědět.

Budeme potřebovat vytvořit odpočet, který bude hráči zobrazovat počet sekund, které mu ještě zbývají.

Použijeme pro to Javascript funkci `setInterval()`. Ta bere jako první parametr funkci, kterou bude volat a jako druhý počet milisekund, po kolika má volání opakovat.

V konstruktoru dialogu s otázkou – `question-dialog.component.ts` budeme naslouchat na zadání nové otázky. Jakmile přijde, vytvoříme časovač a jeho výstup budeme vypisovat v šabloně (viz ukázka 4.26).

```
this.question$.subscribe((q: NumberQuestion) => {
    clearInterval(this.timer);
    this.date2 = new Date(q.targetDate);

    this.timer = setInterval(()=>{
        this.updateTimer();
    },1000)
    this.updateTimer();
});

updateTimer() {
    this.timerText = Math.floor(
        (this.date2.getTime() - Date.now()
    ) / 1000);
}
```

Obrázek 4.26: Ukázka kódu – odpočet

Funkce `updateTimer()` vezme čas do kdy má být otázka zodpovězena, odečte od něj aktuální čas, převede ho na sekundy a zaokrouhlí směrem dolu.

Animace stupně vítězů

V ukázce kódu 4.25 jsme si na řádcích **2** a **3** připravili třídy `first` a `last`. Pomocí nich jim přidáme animace a nastavíme jim `animation-delay`, aby se přehrály až po tom, co se zobrazí správná odpověď.

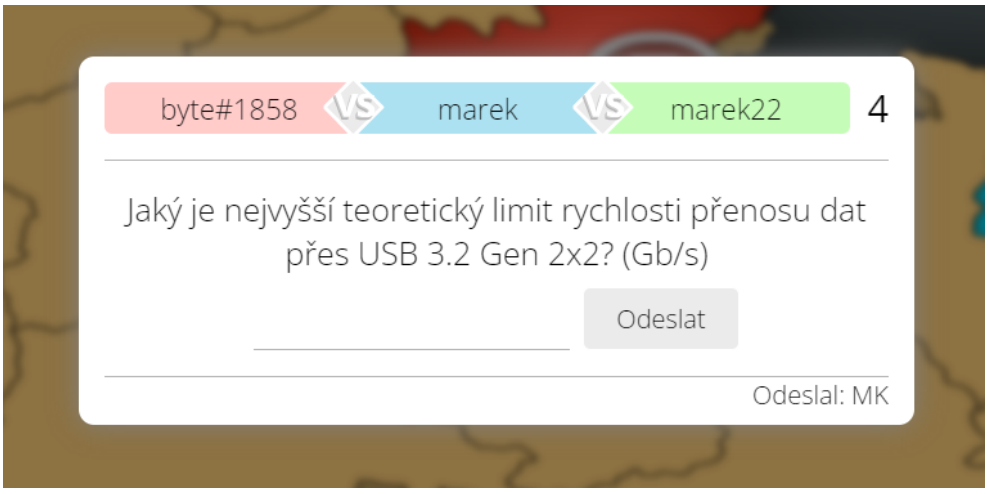
V ukázce 4.27 nejprve zadefinujeme to, jak bude vypadat animace a pak jí v třídě `first` použijeme. Klíčové slovo `forwards` zajistí to, že objekt zůstane v posledním klíčovém snímku po dokončení animace. Pokud bychom ho tam nedali, objekt by se vrátil do původní pozice před animací.

```
@keyframes first { /* vytvoříme klíčové snímky */
    from {
        transform: translateY(0px);
    }
    to {
        transform: translateY(-30px);
    }
}

.first { /* animace se spustí až po 5s od přidání třídy */
    animation: first 5s forwards;
    animation-delay: 3.5s;
}
```

Obrázek 4.27: Ukázka kódu – animace zobrazení výsledků, styly

Výsledná podoba je na obrázcích 4.28 a 4.29.



Obrázek 4.28: Tipovací otázka

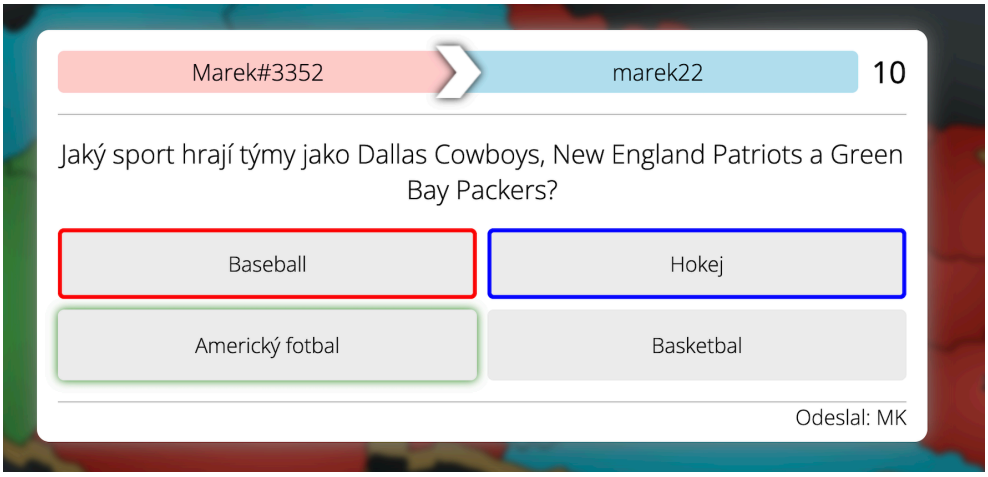
Obrázek 4.29: Výsledky tipovací otázky

4.2.6 Otázka s jednou správnou odpovědí

Implementace otázky s jednou správnou odpovědí je velmi podobná tipovací otázce. Jediným rozdílem bude proměnná `answers`, která obsahuje možné odpovědi na otázku.

Výsledky už nebudeme zobrazovat pomocí stupně vítězů, odpovědi pouze zabarvíme podle barvy hráče, který na ně odpověděl.

Na obrázku 4.30 je barevně vidět, na jakou odpověď hráči odpověděli. Správná zde byla ale třetí možnost, která je zobrazena světle zeleným poblikáváním.



Obrázek 4.30: Otázka s jednou správnou odpovědí

Kapitola 5

Nasazení

Rozhodl jsem se nasazení věnovat vlastní kapitolu, protože to pro mě byla novinka a není to úplně tak jednoduchý proces, jak by se mohlo zdát. Jedná se o multiplayerovou hru a chceme, aby si ji mohli snadno zahrát i ostatní hráči. K tomu budeme potřebovat nějaký server, kde budeme naše řešení hostovat.

Pro nasazení jsem vybral VPS (virtuální privátní server), je levnější než služby typu AWS nebo Azure, ale vyžaduje vyšší úsilí na konfiguraci. Ideálním výstupem by bylo, aby se po každém commitu vytvořil .env soubor, nejnovější verze se nahrála na server a server se zrestartoval. To zní dobře, ale jak na to?

Použijeme novinku od Githubu jménem Actions. Jedná se o formu CI/CD (continous integration/continous delivery), která má za účel usnadnit vývojáři práci s nasazováním nového kódu. Výhodou je, že je zdarma a nabízí uživateli až 2000 minut procesorového času na kompilaci či jiné úkoly, které si nadefinujeme.

5.1 Backend

Nejprve začneme se serverovou částí, protože zde bude práce více. V první řadě je důležité si vytvořit soubor `cd.yml` ve složce `.github/workflows`.

V ukázce 5.1 máme na prvním řádku název a na řádcích 3–6 máme klíčové slovo `on`, to definuje, že se budou joby spouštět vždy při pushi do `master` větve.

```yaml
name: Node.js CD

on:
  push:
    branches:
      - master
```

Obrázek 5.1: Ukázka kódu – `cd.yml`

Dalším klíčovým slovem je slovo `jobs` (viz ukázka 5.2). Job je v podstatě proces, který má nějaký vstup a výstup. Jobů můžeme definovat více a taky můžeme specifikovat, v jakém pořadí se spustí. Každý job má ještě své kroky (`steps`), kterými popisujeme dílčí operace.

Github Actions už má připraveno spoustu předpřipravených balíčků, které nám usnadní život, abychom nemuseli vypisovat všechny příkazy ručně. Na řádku **13** zklonujeme repozitář a na řádku **15** použijeme balíček `create-env-file`, ten slouží v vytvoření `.env` souboru, který obsahuje přístupové údaje k databázi, šifrovací klíč apod. Tyto věci je nedoporučeno verzovat, a proto je máme uloženy jako tajné proměnné (`SECRETS`) přímo v Actions. Není možné si je zobrazit a mají k nim přístup jenom joby.

Posledním krokem tohoto jobu je přenesení celého výstupu na náš server pomocí utility `rsync`. Ta je oproti klasickému `scp` mnohem rychlejší a posílá jenom změny, ne celé soubory znovu.

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest      # job bude běžet v ubuntu
    environment: production     # v prostředí produkce
    steps:
      - uses: actions/checkout@v3 # zklonujeme
      - name: Make envfile
        uses: ozaytsev86/create-env-file@v1
        with:    # vytvoříme .env soubor
          ENV_DB_PORT: '${{ secrets.DB_PORT }}'
          ...
          ENV_BASE_URL: '${{ secrets.BASE_URL }}'
      - name: rsync deployments
        uses: burnett01/rsync-deployments@5.2.1
        with:
          switches: -avzr --delete --exclude="node_modules"
          path: /
          remote_path: diplomka/server
          remote_host: '${{ secrets.HOST }}'
          remote_user: '${{ secrets.USERNAME }}'
          remote_key: '${{ secrets.PRIVATE_KEY }}'
```

Obrázek 5.2: Ukázka kódu – `cd.yml`, deploy job

V ukázce 5.3 vidíme další job – `build`, ten má za úkol připojit se na server pomocí `ssh` a dostahovat potřebné balíčky a zrestartovat server.

Na řádku **31** vidíme, že se job spustí až tehdy, co je hotový job `deploy`. To je chtěné chování, server chceme restartovat až bude nahrán všechen obsah z předchozí joby.

```yaml
build:
  runs-on: ubuntu-latest
  needs: deploy # job se provede až po deployi
  environment: production
  steps:
    - name: Deploy using ssh
      uses: appleboy/ssh-action@master
      with:
        host: '${{ secrets.HOST }}'
        username: '${{ secrets.USERNAME }}'
        key: '${{ secrets.PRIVATE_KEY }}'
        port: '${{ secrets.PORT }}'
        script: | # příkazy se provedou na serveru
          cd diplomka/server
          npm install --omit=dev
          nest build
          pm2 restart server
```

Obrázek 5.3: Ukázka kódu – `cd.yml`, build job

Pozorný čtenář si určitě všiml, že to, co jsme si tu vyrobili, je CD, ne CI. Pokud se během kompilace vyskytne nějaká chyba, dozvíme se to až na serveru.

To není úplně best–practice, správně bychom měli přidat ještě jeden job na začátek našeho seznamu, kde se pokusíme kód zkompilovat. Pokud nenastanou žádné komplikace, až pak pokračovat. Tím nebudu čtenáře dále zatěžovat a přesuneme se na frontend.

5.2 Frontend

Frontend bude opět reagovat na push do větve `master`, jediným rozdílem bude to, že budeme kompilovat už v jobu. Pak už se bude opět pomocí `rsync` nahrávat na server, tentokrát do složky `client`. Zkompilovaný Angular už nemusí běžet jako samostatný proces, můžeme ho servírovat jako statické soubory a toho také využijeme.

Na serveru použijeme package `@nestjs/serve-static` a nastavíme ho způsobem jako v ukázce 5.4.

```
ServeStaticModule.forRoot({
    rootPath: join(__dirname, '..', "..","client","dist"),
    exclude: ['/api/(.*)', '/auth/(.*)']
}),
```

Obrázek 5.4: Ukázka kódu – nastavení servírování statických souborů

Protože máme vytvořeno routování v Angularu, budeme všechny cesty mimo `/api` a `/auth` směrovat přímo do složky `dist` se zkompilovaným výstupem.

Tímto máme vše hotovo, teď už stačí jen nasměrovat doménu a můžeme hrát!

Kapitola 6

Testování

Testování aplikace je nedílnou součástí softwarového vývoje aplikace. V zadání diplomové práce jsme dostali za úkol se zaměřit na testování backendu a jeho výkonu, tak se na to pojďme podívat.

Naše aplikace běží na VPS (virtuálním privátním serveru) s Debianem. Jedná se o poměrně slabý stroj, obsahuje jen jedno jádro `Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz` a 2GB operační paměti. Měřítkem výkonu budeme brát počet aktivních hráčů ve hře.

Budeme tedy potřebovat napsat skript (ukázka 6.1), který bude vytvářet nové klienty, co se budou chovat jako normální hráči.

```
1 (async () => {
2     for (let i = 0; i < 9999; i++) {
3         await delay(200);
4         spawnSocket(i);
5     }
6 })();
```

Obrázek 6.1: Ukázka kódu – skript pro testování

Rozhodl jsem se nastavit čekání mezi každým nově vytvořeným klientem na 200ms. Kratší interval už zahlcoval linku a spousta klientů se ani nedokázala připojit. Dále jsem ještě vypnul přihlašování a ověřování JWT tokenu, abychom to nemuseli pro testování řešit.

Funkce `spawnSocket()` je popsána v ukázce 6.2.

```
1  function spawnSocket(i) {
2      const socket = io(SERVER_URL,{forceNew: true});
3      socket.on("connect", ()=>{
4          socket.emit("matchmaking",true);
5      })
6
7      socket.on("disconnect", ()=>{
8          console.log("disconnect " + i);
9      })
10
11     socket.on("question", (val, callback)=>{
12         if (callback)
13             callback(Math.floor(Math.random()*1000));
14     })
15
16     socket.on("pickOne", (val, callback)=>{
17         if (callback)
18             callback(0);
19     })
20
21     socket.on("selectRegion", (val, callback)=>{
22         if (callback)
23             callback(val.regs[0]);
24     })
25
26     socket.on("endgame",()=>{
27         socket.emit("matchmaking", true);
28     })
29
30     sockets.push(socket);
31 }
```

Obrázek 6.2: Ukázka kódu – funkce `spawnSocket()`

Opravdu velmi jednoduchý kód, hned jakmile se podaří klientovi připojit se, oznámí serverů, že vyhledává hru. Na řádcích **11**, **16** a **21** pak klient odpovídá na otázky a vybírá region.

Na řádku **26** pak hned jakmile skončí hra, začne vyhledávat novou a takto pořád dokola.

Bohužel při prvním testování jsem narazil na limit už po 500 aktivních připojeních. V unixových systémech je každé nové spojení reprezentováno souborovým deskriptorem, každý proces jich ale může vytvořit pouze omezené množství, které je nyní pro naše potřeby nastaveno na příliš nízkou hodnotu

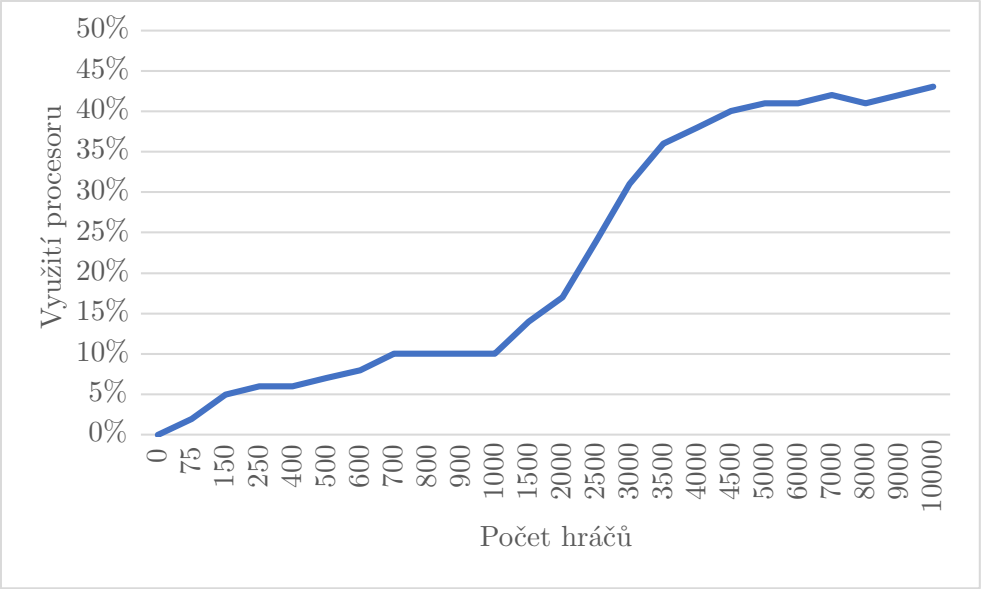
[28].

Toto omezení můžeme změnit v souboru `/etc/security/limits.conf`, kam přidáme na konec následující řádky:

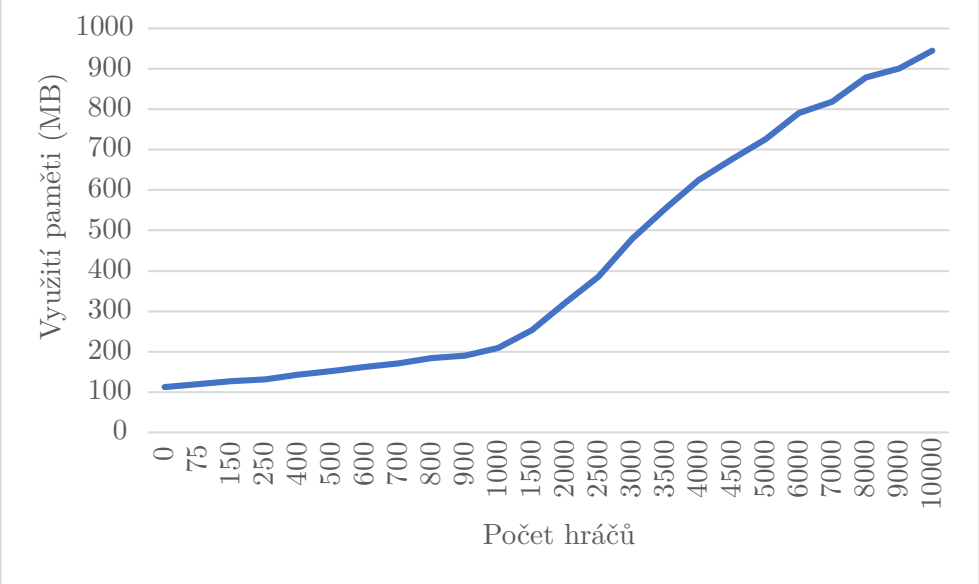
```
*               hard    nofile      100000
*               soft    nofile      100000
```

Nastavíme tak maximální počet souborových deskriptorů na 100000. Po restartu serveru už jsem nebyl odpojován a mohl jsem začít s testováním.

Abych mohl vzdáleně monitorovat využití procesoru a operační paměti, použil jsem program pm2. Ten v terminálu přehledně ukazuje standardní výstup aplikace, počet připojení, využití haldy, CPU, RAM apod.



Obrázek 6.3: Závislost počtu hráčů na využití procesoru



Obrázek 6.4: Závislost počtu hráčů na využití operační paměti

Na grafech 6.3 a 6.4 jsem zaznamenal vývoje závislosti počtu hráčů na CPU a operační paměti.

Zastavil jsem se na 10000 hráčích, tam server využíval 43 % procesoru a 944 megabajtů operační paměti. Odezvy serveru už byly dost pomalé a dlouho se čekalo na zadání otázky. Bohužel kolem 2000 hráčů už začalo zlobit přihlašování a začal jsem dostávat HTTP chybu 504 – Gateway timeout. Což znamená, že server nestihl odpovědět do časového limitu.

Je ale docela zajímavé, že hrát hry šlo bez problému. Nejsem si tedy úplně jist, kde je chyba. Pravděpodobně jsou klasické požadavky na REST API dražší, než WebSockety anebo jsem měl něco špatně nastaveno.

I tak ale server vydržel mnohem větší nápor, než jsem čekal, zas tak velkou popularitu neočekávám a tohle bude aktuálně více než stačit.

Závěr

Cílem práce bylo vytvořit webovou verzi oblíbené hry Dobyvatel, která byla bohužel zrušena a její vývojáři se zaměřili na mobilní zařízení. Povedlo se nám to více než dobře, všechny body zadání jsme splnili. Nejprve jsme prošli alternativní řešení, zhodnotili jejich výhody a nevýhody. Dále jsme si předvedli vhodné technologie pro implementaci naší aplikace.

Navrhli jsme architekturu aplikace, doménový model a wireframe (drátěný model aplikace). Na klientskou i serverovou část jsme použili programovací jazyk Javascript, konkrétně jeho nadstavbu Typescript. V implementaci jsme se zaměřili na to, aby hra obsahovala klíčové prvky, které dělají Dobyvatele charakteristickým – tj. hra s přáteli, tipovací otázky, mapa České republiky apod. V závěru jsme aplikaci nasadili za pomocí Github Actions na náš virtuální privátní server a otestovali ji na výkon.

Aplikace ale ještě určitě není ve finální podobě, během testování s přáteli jsme narazili na pár nedostatků. Konkrétně:

- **Znovupřipojení do hry** – Pokud se uživatel během hry odpojí, chtělo by to přidat funkcionalitu pro připojení zpátky do hry.
- **Umělá inteligence** – Přidat umělou inteligenci, aby si člověk mohl zahrát i pokud nemá právě s kým.
- **Uživatelský profil** – Přidat úpravu svého uživatelské profilu, aby si člověk mohl změnit svůj username, avatar apod.
- **Chat** – Přidat chat, aby se spolu mohli hráči seznamovat a hrát proti sobě.
- **Přihlášení jako host** – Pokud hráč nechce využít přihlášení pomocí sociálních sítí, hodila by se možnost přihlásit se jako host.

- **Tvorba vlastní místnosti** – Možnost vytvořit si vlastní místnost a pozvat do ní své přátele, popř. vygenerovat link na místnost, aby hraní s přáteli bylo co nejsnazší.
- **Zvuky** – Původní hra obsahovala zvuky, které lépe hráče vtáhly do hry. To samé by se hodilo přidat i do naší implementace.

Aplikace je ale jinak plně funkční a výše zmíněné nedostatky nejsou nic, co by narušovalo funkčnost hry.

Literatura

[1] FlyGunCZ: Český GamePlay | Dobyvatel (Vědomostní Hra) | FlyGun + SoNy | Divné Otázky | HD - 720p. únor 2013, [cit. 2023-04-08]. Dostupné z: https://www.youtube.com/watch?v=h8BYQqyHfbA

[2] netbox - Facebook. listopad 2020, [cit. 2023-04-08]. Dostupné z: https://www.facebook.com/photo/?fbid=10158608326553567&set=pcb.10158608326933567

[3] Dobyvatel - Aplikace na Google Play. duben 2023, [cit. 2023-04-07]. Dostupné z: https://play.google.com/store/apps/details?id=com.thxgames.triviador

[4] Krotoff, T.: Front-end frameworks popularity (React, Vue, Angular and Svelte). únor 2022, [cit. 2023-04-03]. Dostupné z: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190

[5] NgRx - @ngrx/store. duben 2023, [cit. 2023-04-03]. Dostupné z: https://ngrx.io/guide/store

[6] Kuchař, J.: 1. Introduction, Node.js - callbacks, promise, async/await. *NI-AM2 Courses*, únor 2021, [cit. 2023-04-05]. Dostupné z: https://courses.fit.cvut.cz/NI-AM2/tutorials/01/index.html

[7] JSON Web Token Introduction - jwt.io. *jwt.io introduction*, duben 2023, [cit. 2023-04-07]. Dostupné z: https://jwt.io/introduction

[8] TV Nova přináší nový herní server dobyvatel.cz. *lupa.cz*, duben 2010, [cit. 2023-04-02]. Dostupné z: https://www.lupa.cz/tiskove-zpravy/tv-nova-prinasi-novy-herni-server-dobyvatel-cz/

[9] Nova, T.: Dobyvatel.cz slaví dva roky od svého spuštění! duben 2012, [cit. 2023-04-02]. Dostupné z: https://tn.nova.cz/zpravodajstvi/clanek/251791-dobyvatel-cz-slavi-dva-roky-od-sveho-spusteni

[10] Dombrovský, J.: Petice za obnovení staré verze hry Dobyvatel. červenec 2016, [cit. 2023-04-06]. Dostupné z: https://e-petice.cz/petitions/petice-za-obnoveni-hry-dobyvatel.html

[11] Triviador on the App Store. duben 2023, [cit. 2023-04-07]. Dostupné z: https://apps.apple.com/cz/app/dobyvatel/id1520397866

[12] TriviaNerd: Play, Host & Create Live Multiplayer Trivia Games. duben 2023, [cit. 2023-04-09]. Dostupné z: https://www.trivianerd.com/

[13] conQUIZtador. duben 2023, [cit. 2023-04-09]. Dostupné z: https://conquiztador.online/

[14] Kvapil, J.: Lekce 1 - Úvod do TypeScriptu. květen 2018, [cit. 2023-04-05]. Dostupné z: https://www.itnetwork.cz/javascript/typescript/uvod-do-typescriptu

[15] Angular (web framework) - Wikipedia. duben 2023, [cit. 2023-04-12]. Dostupné z: https://en.wikipedia.org/wiki/Angular_(web_framework)

[16] Máca, J.: Lekce 1 - Úvod do React. leden 2023, [cit. 2023-04-09]. Dostupné z: https://www.itnetwork.cz/javascript/react/zaklady/uvod-do-react/

[17] Cromwell, V.: Between the Wires: An interview with Vue.js creator Evan You. květen 2017, [cit. 2023-04-04]. Dostupné z: https://www.freecodecamp.org/news/between-the-wires-an-interview-with-vue-js-creator-evan-you-e383cbf57cc4/

[18] Single File Components | Vue.js. duben 2023, [cit. 2023-04-10]. Dostupné z: https://vuejs.org/guide/scaling-up/sfc.html

[19] Jonas Bonér, R. K. a. M. T., Dave Farley: Reaktivní Manifest. září 2014, [cit. 2023-04-03]. Dostupné z: https://www.reactivemanifesto.org/cz

[20] Roos, P.: What is NgRx and why is it used in Angular apps? *workingsoftware.dev*, červen 2022, [cit. 2023-04-03]. Dostupné z: https://www.workingsoftware.dev/what-is-ngrx-and-why-is-it-used-in-angular/

[21] Sheldon, R.: What is the Node.js (Node) runtime environment?— TechTarget Definition. *TechTarget*, listopad 2022, [cit. 2023-04-05]. Dostupné z: https://www.techtarget.com/whatis/definition/Nodejs

[22] The Node.js Event Loop, Timers, and process.nextTick() | Node.js. *Node.js Docs*, duben 2023, [cit. 2023-04-16]. Dostupné z: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick

[23] What is the Node.js (Node) runtime environment?—TechTarget Definition. *npm Docs*, duben 2023, [cit. 2023-04-05]. Dostupné z: https://docs.npmjs.com/about-npm

[24] Sahu, A.: Top 10 NPM Packages for Node JS Developers. *Turing.com*, červen 2022, [cit. 2023-04-16]. Dostupné z: https://www.turing.com/blog/top-npm-packages-for-node-js-developers/

[25] Documentation | NestJS - A progressive Node.js framework. duben 2023, [cit. 2023-04-06]. Dostupné z: https://docs.nestjs.com/

[26] websockets | Can I use... Support tables for HTML5, CSS3, etc. duben 2023, [cit. 2023-04-06]. Dostupné z: https://caniuse.com/?search=websockets

[27] Chand, M.: Most Popular Databases In The World (2023). *C-Sharp Corner*, prosinec 2022, [cit. 2023-04-12]. Dostupné z: https://www.c-sharpcorner.com/article/what-is-the-most-popular-database-in-the-world/

[28] Performance tuning | Socket.IO. duben 2023, [cit. 2023-04-20]. Dostupné z: https://socket.io/docs/v4/performance-tuning/

Příloha A

Seznam použitých zkratek

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

HTML Hypertext Markup Language

SSR Server Side Rendering

SEO Search Engine Optimization

UI User Interface

API Application Programming Interface

SSL Secure Sockets Layer

CLI Command Line Interface

SQL Structured Query Language

ORM Object-Relational Mapping

JSON JavaScript Object Notation

JWT JSON Web Token

UML Unified Modeling Language

VPS Virtual Private Server

CI Continous Integration

CD Continous Delivery

REST Representational State Transfer

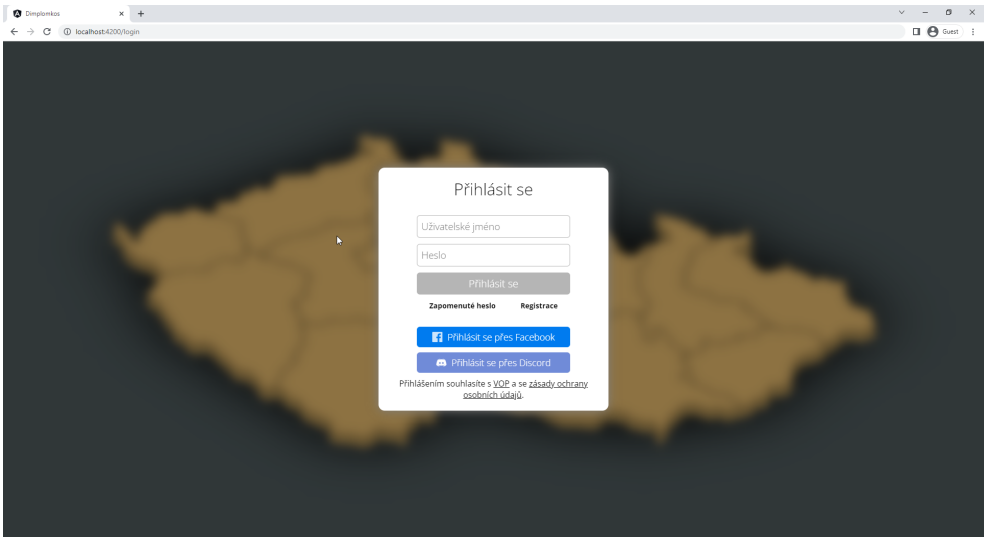
URL Uniform Resource Locator

CSS Cascading Style Sheets

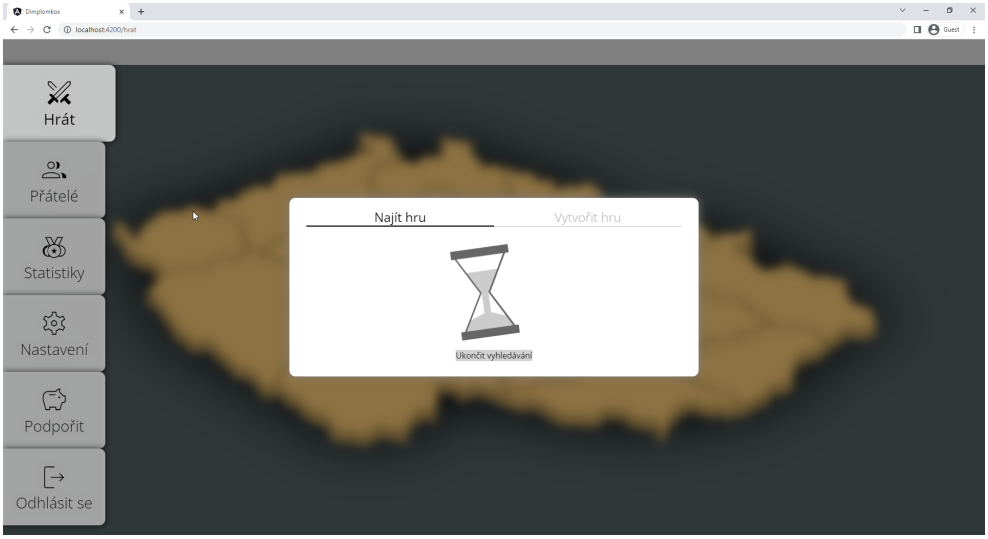
SVG Scalable Vector Graphics

Příloha B

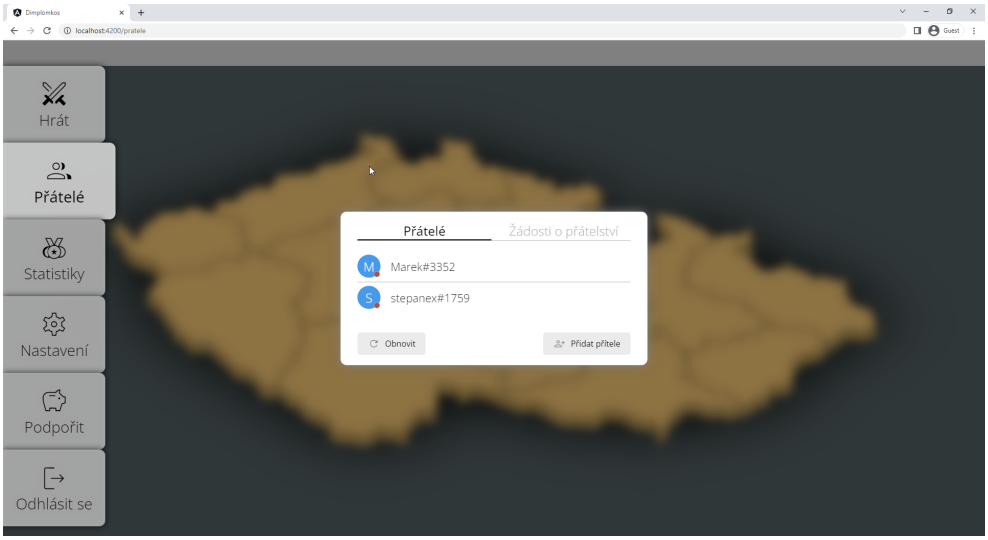
Výsledná podoba aplikace



Obrázek B.1: Přihlášení

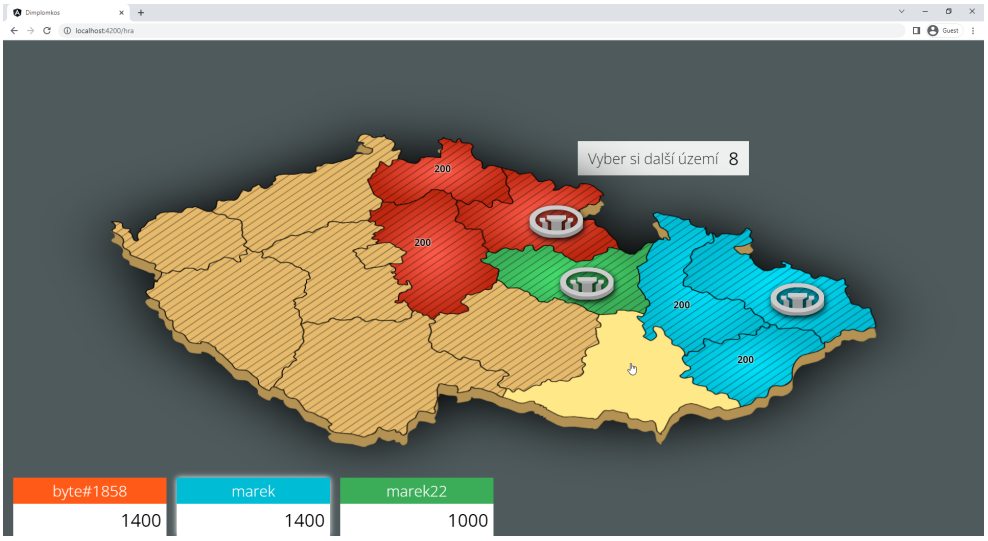


Obrázek B.2: Vyhledávání hry

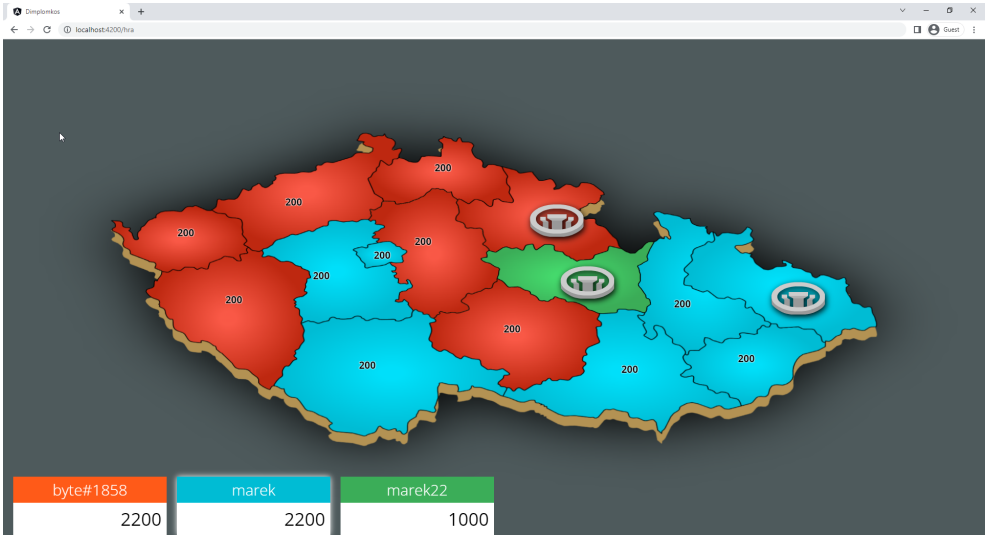


Obrázek B.3: Přátelé

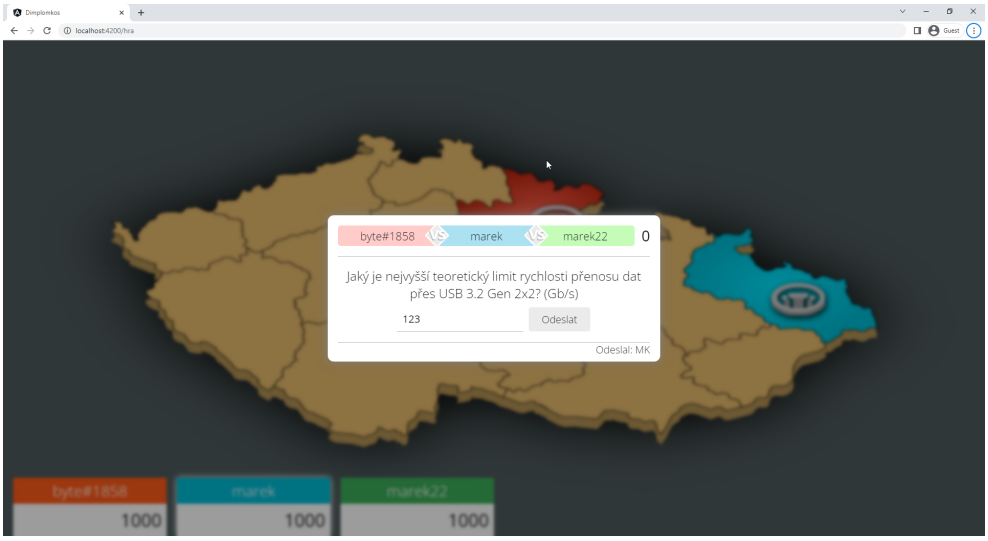
Obrázek B.4: Historie odehraných her



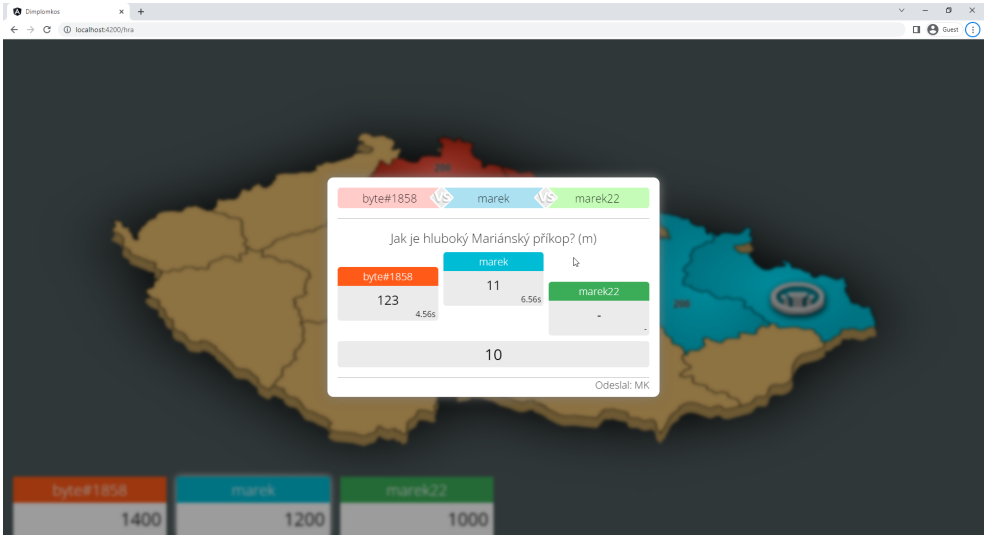
Obrázek B.5: Výběr regionu



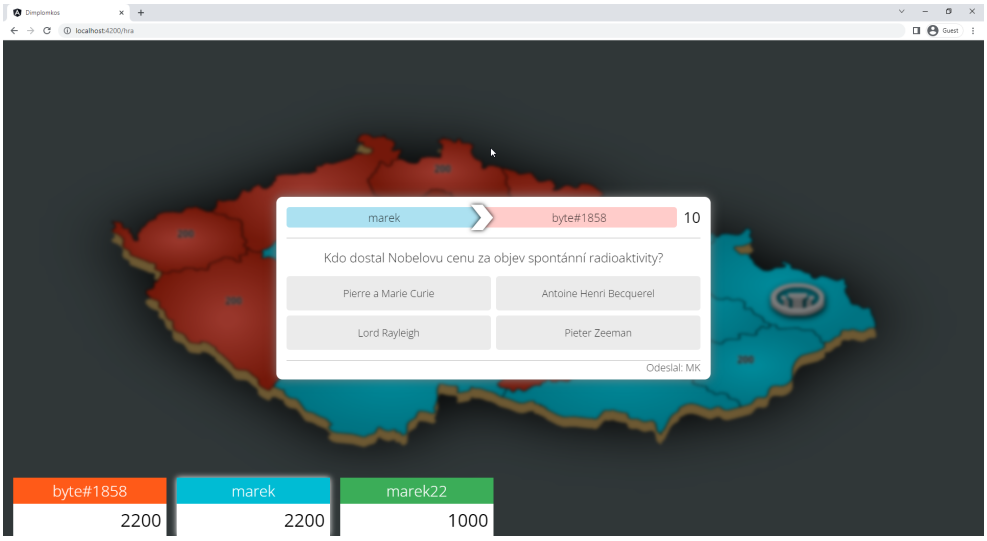
Obrázek B.6: Herní mapa



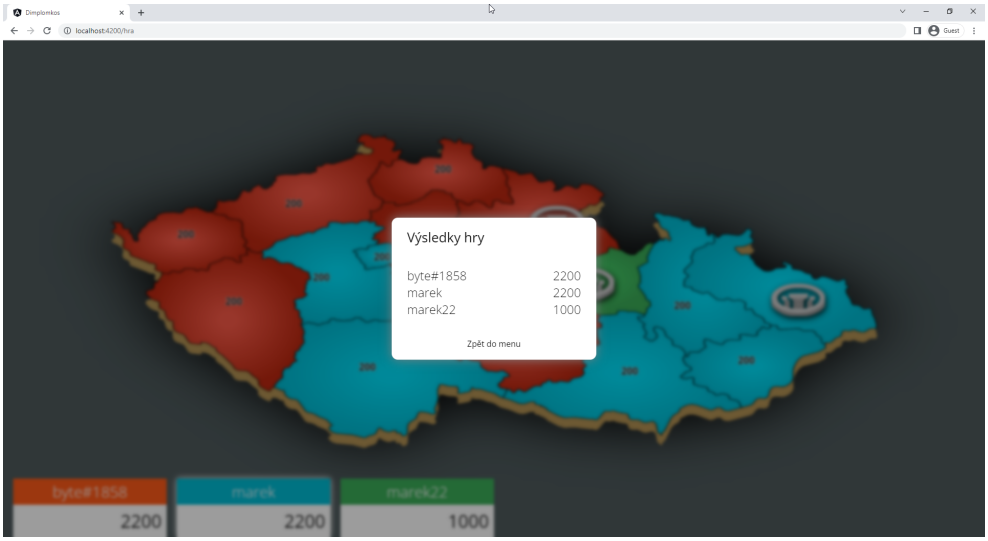
Obrázek B.7: Tipovací otázka



Obrázek B.8: Výsledky tipovací otázky



Obrázek B.9: Otázka s jednou správnou odpovědí



Obrázek B.10: Výsledky hry

Příloha C

Obsah přiloženého CD

```
readme.txt ..... stručný popis obsahu CD
src
    thesis ..... zdrojová forma práce ve formátu LaTeX
screenshots ..... snímky obrazovky z výsledného řešení
text ..... text práce
    thesis.pdf ..... text práce ve formátu PDF
```