KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Trasovací systém uživatelů mobilních zařízení

Tomáš Květoň

PLZEŇ 2023

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Trasovací systém uživatelů mobilních zařízení

Bc. Tomáš Květoň

Vedoucí práce
Ing. Ladislav Pešička

PLZEŇ **2023**

© Tomáš Květoň, 2023.

Všechna práva vyhrazena. Žádná část tohoto dokumentu nesmí být reprodukována ani rozšiřována jakoukoli formou, elektronicky či mechanicky, fotokopírováním, nahráváním nebo jiným způsobem, nebo uložena v systému pro ukládání a vyhledávání informací bez písemného souhlasu držitelů autorských práv.

Citace v seznamu literatury:
KVĚTOŇ, Tomáš. *Trasovací systém uživatelů mobilních zařízení*. Plzeň, 2023. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Ladislav Pešička.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Tomáš KVĚTOŇ**
Osobní číslo:	**A22N0128P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Trasovací systém uživatelů mobilních zařízení**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Prozkoumejte různé způsoby trasování uživatelů mobilních zařízení. Prozkoumejte vybrané aplikace, které se podobnou problematikou zabývají.
2. Vyberte vhodné technologie a metody trasování, které by bylo vhodné použít pro připravovaný systém.
3. Navrhněte systém, který bude trasovat mobilní zařízení zvolenými metodami a který bude podporovat OS Android.
4. Navržený systém realizujte, ověřte jeho vlastnosti a navrhněte další možná rozšíření.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Ladislav Pešička**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **9. září 2022**
Termín odevzdání diplomové práce: **18. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2022

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

V Plzni dne 18. května 2023

. . . .

Tomáš Květoň

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

Tato práce popisuje komplexní řešení systému pro trasování polohy mobilních zařízení, které je založeno na modelu klient-server.

Dokument popisuje klientskou část, která zde představuje samotná mobilní zařízení, na kterých jsou uživatelům promítána nezbytná data do 3D mapy, včetně jejich polohy. Na druhé straně je zde serverová část, kde popisuji relativně složité, ale sofistikované řešení návrhu celé této části systému. Jedním z cílů této práce je zaznamenat celý softwarový vývoj robustního systému od samotného vzniku nápadu, analýzu aktuálního trhu, a návrh až po jeho realizaci ve formě funkční aplikace. V rámci toho budu řešit nemalé množství překážek, které se během vývoje vyskytly, a to především v době analýzy, kde bylo nutné provádět velmi opatrně každé rozhodnutí pro minimalizaci dopadů v pokročilé fázi vývoje.

Správně zvolená a udržovaná architektura je klíčový bod vývoje softwaru z pohledu udržitelnosti a ne vždy je na ni kladen takový důraz, jaký by měl správně být. Práce tedy také popisuje architektonické řešení, kromě samotného řešení práce s geolokačními daty.

Mimo hlavní cíl práce, tj. samotný návrh a realizace popisovaného systému, je důležitý i efekt tohoto cíle. Tato práce demonstruje robustní a funkční řešení uceleného systému pro práci s geolokačními daty, pro který je dlouhodobá udržitelnost jedním z hlavních požadavků.

Abstract

This thesis describes a complex solution of a system for tracking the location of mobile devices, which is based on the client-server model.

The document describes the client app running on mobile devices where the necessary data are projected onto a 3D map for users, including their location. On the other hand, there is the server part. This part describes a complex but sophisticated solution of the design of this entire part of the system. One of the key points for this work is to go through the entire software development process of the system, from the very creation of the idea, an analysis of the current market, to its implementation in the form of a working application. As part of this, I will address some of troubles

that occurred during the development, especially during the analysis, where it was necessary to make every decision very carefully to minimize the impacts in the advanced development phase.

A properly chosen and maintained architecture is the key to software development from a sustainability point of view. It is not always evaluated as a high priority as it might be. In addition, the thesis also describes the architectural solution.

Apart from the main goal of this thesis, meaning the actual design and implementation of the described system, the effect of this goal is also important. The thesis demonstrates a robust and functional solution of a complete system to work with geolocation data, for which long-term sustainability is one of the main requirements.

Klíčová slova

.NET architecture • Unity • geolocation • mobile game

Poděkování

Na tomto místě bych rád poděkoval svému mentorovi a vedoucímu mé diplomové práce Ing. Ladislavu Pešičkovi za cenné rady, nápady, připomínky a především za čas, který strávil během tvorby.

Obsah

1 Úvod **3**

2 Rozbor práce, analýza trhu a možnosti trasování **5**
2.1 Analýza vzniku systému 5
2.2 Potřebné vlastnosti 6
2.3 Související aplikace 7
2.3.1 Aplikace, které trasují uživatele 8
2.3.2 Aplikace s herními prvky 14
2.4 Rozbor návrhu pro budoucí vývoj 15
2.4.1 Příklady použití v reálném životě 16
2.5 Možné způsoby trasování 16
2.6 Očekávaný cíl této práce 18

3 Výběr a popis použitých technologií **21**
3.1 Technologické požadavky a omezení 21
3.2 Analýza technologických řešení v úvaze 22
3.2.1 Řešení serverové části 22
3.2.2 Řešení klientské části 24
3.2.3 Řešení vykreslování map 25
3.2.4 Řešení datového uložiště 28
3.3 Výběr technologií pro zpracování 29

4 Analýza architektonických řešení **33**
4.1 Architektonický design 36
4.1.1 Hexagonal Architecture 36
4.1.2 Onion Architecture 37
4.1.3 Clean Architecture 38
4.2 Architektonické vzory 39
4.2.1 CQRS pattern 39
4.2.2 Event Sourcing pattern 39
4.2.3 Specification pattern 40

4.2.4 Repository pattern 41
4.2.5 Validace & Always-Valid Domain Model 41
4.3 Komunikační způsoby zvolených technologií 42
4.3.1 SignalR 42
4.4 Výběr zvolených řešení 42

5 Návrh řešení a popis implementace 45
5.1 Architektura systému 45
5.2 Serverová architektura 47
5.2.1 Monolit vs. Microservices 51
5.2.2 Organizace a tok dat 53
5.2.3 Získávání a ukládání dat 56
5.2.4 Správa geolokačních dat 64
5.3 Architektura klientské aplikace 66
5.4 Komunikace a přenos dat v systému 67
5.5 Zobrazování polohy zařízení a implementace Mapbox SDK 71
5.5.1 Získávání geolokačních dat 72
5.5.2 Oprava chyb a zpětná vazba k vývojářům 73
5.5.3 Pohyb po mapě a výseč pro hráče na mapě 75
5.5.4 Synchronizace se serverem, vykreslování dat 81
5.6 Implementované funkce a vlastnosti systému 84

6 Návrh rozšíření systému 87

7 Organizace projektu & Testování 89
7.1 Plánování 89
7.2 Dockerizace 90
7.3 Testování 91

8 Závěr 95

A Instalace & Uživatelská příručka 97

B Obsah přiloženého souborového archivu 103

Bibliografie 105

Seznam obrázků 113

Seznam tabulek 115

Seznam výpisů 117

Úvod 1

V dnešní době drtivá většina lidí po celém světě vlastní svůj vlastní chytrý mobilní telefon. Každý z nás jej používá pro různé účely, jako jsou mobilní hry, udržení kontaktu s ostatními na dálku nebo pro nakupování.

S ohledem na rozdíly užití mobilních telefonů jednotlivých uživatelů, máme zde jednu funkci, kterou mají dnes již naprostá většina zařízení a to je ***GPS***[1]. Výše zmíněné příklady užití se na první pohled nemusí zdát, že by využívaly vaší reálnou polohu, ale mohou ji využít k zlepšení poskytovaných služeb. Příkladem by mohla být dodávková služba, kde v takové aplikaci již máte spočítanou cenu dovozu dle vaší polohy apod.

Jsme obklopeni elektronikou, která velmi často pracuje s naší geolokační polohou, aniž bychom si to přímo uvědomovali. Cílem této práce je vytvoření systému, který bude užitečný pro lidi v jejich každodenním životě na principu trasování mobilních zařízení. Zároveň by měl tento systém sloužit jako vhodný základ projektu pro vývoj podobných aplikací.

Chtěl bych ostatním nabídnout systém pomocí kterého mohou být v kontaktu se svými přáteli nebo svými nejbližšími. Nejen ve formě textové komunikace, ale ve formě jejich vzájemné kontroly polohy. Inspirace a tím i prvotní myšlenka, ze kterého jsem vycházel, byl princip rodičovské kontroly s možnou aplikací v širším zaměření, než je rodič-potomek. Nicméně více o samotné myšlence bude řečeno v další kapitole 2. Tato kapitola bude zároveň více úvahově zaměřená.

V kapitole 3 provedu výběr technologií, kterými celé řešení budu chtít zpracovat. Následně v kapitole 4 analyzuji různé architektonické přístupy, které jsou pro takový systém nezbytné. V této práci se dostanu od samotného nápadu až k samotnému funkčnímu systému. Během toho budu řešit návrhová a implementační rozhodnutí

[1] Global Positioning System, také jako Globální Polohový Systém je globální družicový polohový systém, který umožňuje pomocí elektronického přijímače určit přesnou polohu na povrchu Země.

(v kapitole 5), která jsou pro takový systém důležitá z hlediska udržitelnosti. Systém by měl být dlouhodobě udržitelný pro vývoj, a to s minimálním předpokladem pěti a více let. Proto uvedu několik možných způsobů rozšíření, které popisuji v kapitole 6. Dále v kapitole 7 popíši způsob testování a organizaci projektu celého systému.

Pokud chci všeho tohoto dosáhnout, nebude to jednoduché. Bude nutné vytvořit robustní infrastrukturu a důkladně dbát na prvotní návrh, aby po odstupu času nedošlo k výrazným limitacím ve vývoji a tím narušení celkové architektury systému. Toto bude také jeden z hlavních bodů celé práce, který je nezbytný pro celý systém. Zároveň bych touto prací rád poukázal na některá architektonická řešení, která mohou být přehlížena či zanedbávána bez ohledu na jejich důležitost.

Rozbor práce, analýza trhu a možnosti trasování

2

V této kapitole bych rád představil vznik myšlenky, její vývoj[2.1], analýzu aktuálního trhu[2.3], příklady použití v reálném světě[2.4], způsoby trasování[2.5] a v neposlední řadě výsledek, který bych od této práce očekával[2.6].

2.1 Analýza vzniku systému

Již v úvodu[1] jsem zmínil prvotní myšlenku, kterou byla rodičovská kontrola. Jednalo se o zatím „neopracovanou" myšlenku, kterou jsem postupně dále rozvíjel. Snažil jsem se přijít s něčím, co by mohlo být použito i jinak, než pouze jako kontrola mezi potomkem a rodičem, a to například v okruhu přátel.

Jedna z prvních inspirací byla aplikace Nearby Friends integrovaná v sociální síti Facebook od společnosti ***Meta***[1]. Tato aplikace umožňovala svým uživatelům mezi sebou sdílet svoji reálnou polohu. Do svého systému však musím zanést i vlastní originální funkce. Jedna z takových funkcí, kterou jsem chtěl implementovat byla historie pohybu jednotlivých uživatelů, která by s sebou přinesla možnost zpětného zobrazení všech uživatelů sdílející svoji polohu ve skupině. Díky tomu by bylo možné mezi sebou najít společné zájmy, konkrétně zájmová místa všech uživatelů ve skupině.

Provedl jsem další analýzy aktuálního trhu a vyhledal možné aplikace se spojitostí k rodičovské kontrole nebo třeba jen trasování vaší polohy. Několik z nich dále popíši v sekci 2.3. Po dlouhé analýze těchto aplikací jsem však nebyl zcela spokojen s výsledkem toho, že se dokáži odlišit pouze malou podmnožinou vlastností, kde základ aplikace by byl totožný a poskytoval by velmi podobnou službu uživatelům.

[1] Společnost dříve známa jako Facebook, Inc., nyní vlastnící jedny z nejpopulárnějších sociálních sítí jimiž jsou: Facebook, Instagram a WhatsApp.

Postupně se tak z úvah o použití dat pohybu uživatelů začala stávat myšlenka směřující k využití soutěživosti mezi jednotlivými uživateli. Začal tak vznikat ucelenější pohled, jak by celý systém aplikace mohl fungovat. Řekl bych, že do jisté míry celou myšlenku jistě ovlivnil i fakt přítomnosti a aktuální téma epidemie ***COVID 19***, během které byla převážná většina lidí po celém světě nucena omezit blízký kontakt s ostatními. V kombinaci s faktem, že dnešní doba sociálních sítí veškerou standardní komunikaci jen oddaluje, tak jsem na základě tohoto chtěl docílit, aby to mělo i nějaký hlubší význam, než jen pouhou soutěživost a další řadovou mobilní hru. Po několika dalších dnech jsem dospěl k nápadu výzev. Abychom jsme se jako jednotlivci posouvali, může nám k tomu dopomoci si stanovit cíle, případně pro sebe vytyčit i nějakou výzvu, kterou se snažíme následně pokořit. Spojíme-li to s kontextem více uživatelů, vzniká nám tady jistá rivalita, soutěživost.

Od počáteční myšlenky s rodičovskou kontrolou jsem se dostal až k uživatelské službě, která nabízí herní, sociální a soutěživé prvky. Jde o poměrně velký skok a s ním jsem samozřejmě stále dohledával aplikace s podobnou tématikou[2.3] a posuzoval, zda je to něco, co lze dále rozvíjet. Postupně jsem práci ještě více rozpracoval, způsoby výzev a další funkcionality, o čemž budu hovořit více v jedné z dalších kapitol[6].

Finální verzí je platforma, ve které se příchozí uživatel může socializovat s dalšími uživateli. Myšlenkou je dát možnost lidem více komunikovat prostřednictvím běžné reálné komunikace a prostřednictvím herních prvků je pobídnout k vlastnímu pohybu.

2.2 Potřebné vlastnosti

Nyní bych rád zmínil vlastnosti, které vyvíjený systém bude nutně potřebovat. Pokusím se zmínit všechny vlastnosti, které je potřebné během vývoje zohlednit a mít o nich povědomí.

Začnu nutnými vlastnostmi, které plynou ze zadání této diplomové práce:

- Systém by měl fungovat/pracovat s geolokačními daty. Měl by trasovat pozici reálného mobilního zařízení.
- Systém by měl podporovat operační systém Android.
- Systém by měl být schopen dalšího rozšíření i po dokončení této práce.

Jedná se spíše o technické vlastnosti, které je důležité zmínit. Teoretické požadavky na práci jsou součástí tohoto dokumentu. Mimo zadané požadavky jsou zde i požadavky, které vyplývají z tématu nápadu tohoto systému[2.1]. Tyto požadavky jsou pro mě rovnocenně důležité, i když nejsou součástí samotného zadání práce.

Celý systém by měl sloužit pro velké množství uživatelů současně jako služba. Z tohoto hlediska tedy nevytvářím jednoduchou aplikaci pro jednoho uživatele, ale je nutné zvážit, že jsou zde možná úskalí v dostupnosti a vytížení. Systém také bude třeba pravidelně udržovat. Bude tedy nutné si vývoj pořádně rozmyslet a vytvořit takovou infrastrukturu, která bude snadno rozšiřitelná, a hlavně snadno udržitelná pro další vývoj. To se může zdát jako jednoduchý požadavek, nicméně je zde mnoho „ale“ a „co když“, které musím vzít v úvahu.

V první řadě bude velmi důležité navrhnout komunikaci mezi uživateli, aby se data mezi nimi mohla synchronizovat. Nebudu zde plně popisovat návrh architektury, jelikož od tomu nechávám prostor ve zvláštní sekci[5.1]. Nicméně, nejlépe uchopitelný návrh je pomocí nějakého prostředníka, tj. v našem případě server. Z tohoto důvodu potřebujeme, aby server měl následující vlastnosti:

- Rozšiřovatelný a udržitelný systém komunikace mezi serverem a klientskými aplikacemi uživatelů.
- Robustní architektura serveru, která nám v budoucnu nebude bránit možným změnám, ke kterým může dojít.
- Škálovatelnost. Lze předpokládat, že velké množství uživatelů neudrží pouze jeden stroj, jedna aplikace. Je nutné taky myslet i na škálovatelnost mezi regiony po celém světě.

2.3 Související aplikace

V této sekci bych rád mluvil o aplikacích, které jsem vyhledával na aktuálním trhu v souvislosti s mnou vyvíjenou službou. Snažil jsem se hledat podobnosti a vlastnosti, které dané aplikace nabízejí a udržují si jimi aktivní uživatele.

Rád bych rozdělil tento popis do dvou menších sekcí, kde v první budu porovnávat aplikace, které se podobali prvotním myšlence celé práce a ze kterých jsem vycházel[2.3.1]. V druhé sekci potom porovnám aplikace, které se myšlenkou přibližují více té mé finální2.3.2. Pokusím se najít oblasti, ve kterých bych mohl nabrat na originalitě a vlastnosti, které bych mohl dělat lépe.

2.3.1 Aplikace, které trasují uživatele

V této části bude popsáno několik souvisejících aplikací. Následně každou z popisovaných aplikací uvedu v rámci své vlastní podsekce, kterou vytvořím srovnání těchto aplikací.

Mezi jedny z nejzákladnějších aplikací (dle mého názoru), které nabízejí možnost trasování, patří již zmíněná aplikace ***Facebook (Nearby Friends)*** a ***Google Maps (Share Location)***. Jsem toho názoru, že se jedná o jedny z nejvíce používaných aplikací a obě dvě jsou více-méně sociální sítě v rámci kterých jsou nabízené tyto speciální služby sdílení své pozice s ostatními v reálném světě. Obě dvě aplikace nabízejí pouze samotné sdílení vlastní pozice s ostatními zvolenými uživateli. Bohužel, nedokáži přesně dále říci jakými vlastnostmi nabývala aplikace Nearby Friends, jelikož se její společnost rozhodla další vývoj a samotný chod ukončit, jak je popisováno v jednom ze zveřejněných článků[Wei22] (proto ji vynechám v následném porovnávání).

Mimo tyto základní dvě aplikace jsem dále hledal i mezi aplikacemi z žebříčku několika nejvíce doporučených za aktuální rok 2022. Nebudu zde uvádět všechny nalezené aplikace, ale uvedu jen ty nejzajímavější pro srovnání. Ke každé z nich uvedu důležité parametry aplikace a případně i poznamenám, co konkrétně přispělo k mému vyvíjenému systému. Nicméně, když projdu všechny tyto informace z níže uvedených aplikací, tak je již možná zjevné, že všechny jsou podporovány oběma platformami - Android a iOS. Dále jsou všechny dostupné zdarma.

Každopádně ze všech těchto informací dále vyplývá ještě jedna důležitá věc a tou je nasycenost trhu. Jde pouze o zlomek aplikací, které se tématem rodičovské kontroly nebo podobného zabývá. Bylo by velmi obtížné přijít s něčím novým, a ještě se k tomu prosadit. Rozvoj myšlenky tedy musel pokračovat, avšak pomocí této analýzy jsem získal mnoho užitečných informací, jak již zmiňuji v níže specifických sekcích. Když se myšlenka začala postupně formovat k finální podobně, začal jsem ji porovnávat (v sekci 2.3.2) s trochu odlišným typem aplikací, které se mojí výsledné představě zase o něco více přibližují.

Share Location (Google Maps)

Velkým kladem této aplikace je sdílení lokace bez dalších speciálních funkcí. Dále je zde jednoduchost použití díky instantním URL, které lze sdílet a pouze uživatel s konkrétní URL může vaši polohu sledovat.

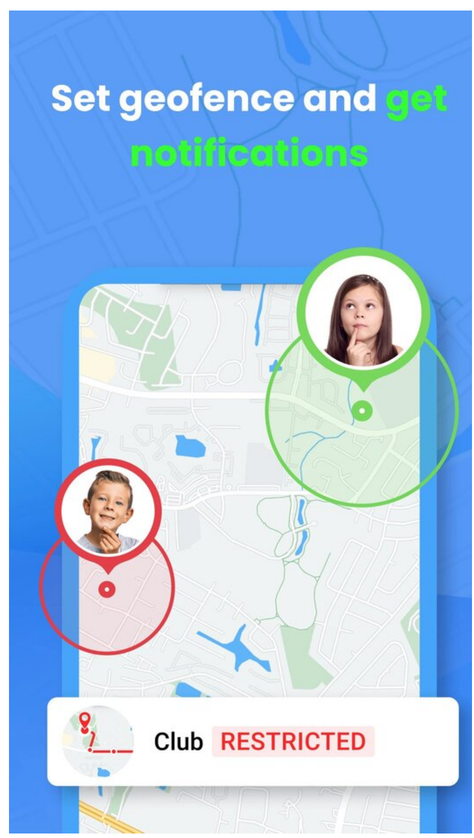
- Sdílení své vlastní polohy pomocí vygenerovaného URL odkazu nebo přidělením přístupu vybraným kontaktům ze seznamu přátel.
- Sdílet svoji pozici lze po stanovenou časovou periodu, tj. například odkaz bude platný jen po předem stanovenou dobu.
- Podpora pro Android a iOS.
- Aplikace je zdarma.
- Více se lze dozvědět v přiložených oficiálních dokumentacích[Goo23a][Goo23b] nebo v jednom z mnou zmíněných článku[Coh21].

mSpy

Pravděpodobně nejvyspělejší ze všech uváděných aplikací pro rodičovskou kontrolu. Nabízí rozmanité funkce monitorující zprávy různých sociálních aplikací, hovory, odposlech, a mnoho dalšího. Je pouze na uživateli aplikace, jak si daný monitoring přizpůsobíte. Dále uvedu v několika bodech fakta o této aplikaci:

- Mnoho funkcí v rámci jedné aplikace.
- Upozornění na nejrůznější akce, jako jsou nevhodné (nechtěné) kontakty nebo na základě pohybu po mapě (příchody/odchody na předem nastavitelná místa).
- Podpora pro Android a iOS.
- Zdarma pouze omezená verze mLite. Placená plná verze funguje na bázi předplatného, a to pouze pro jedno zařízení. Výše ceny je zmíněná na oficiálních stránkách[23ah].
- Jakékoli další podrobné informace lze čerpat z oficiálních webových stránek[23ag] nebo z jejich prezentace jedné z verzí aplikace v obchodě ***Google Play***[23ad].

Ukázka aplikace na obr. 2.1 zobrazuje použití uživatelem definované výseče přímo na mapě a na základě pohybu jednotlivých uživatelů jsou spouštěny konkrétní akce.



Obrázek 2.1: Ukázka implementace výseče na mapě v aplikaci mLite [23ad].

Life360

Aplikace vytváří dojem zjednodušené méně komplikované verze podobných aplikací, která nabízí pouze kompaktní navržené funkce. Dále uvedu několik důležitých bodů o této aplikaci:

- Nabízí možnost vytvářet skupiny v rámci kterých lze posílat ostatním textové zprávy a sdílet s nimi svoji reálnou polohu.
- Nabízí možnost navigace k ostatním.
- Podobně jako předchozí již zmiňované, tak také nabízí možnost upozornění, pokud se někdo ze skupiny dostane na určité místo.
- Aplikace je zdarma a nabízí i placenou verzi, která zvyšuje např. počet dnů sledování historie lokací uživatelů.
- Podpora pro Android a iOS.
- Podrobnější informace lze dohled na oficiálních stránkách, které jsem zmínil v této práci[23w].

FollowMee GPS Tracker

Jednoduchý GPS trasovací systém, který nabízí webové rozhraní a správu zařízení přidaných do toho systému. Vhodný pro firemní nebo osobní účely. Nabízí hromadnou správu zařízení a jejich sledování aktuální polohy i jejich historii. Důležitými informaci o této aplikaci jsou:

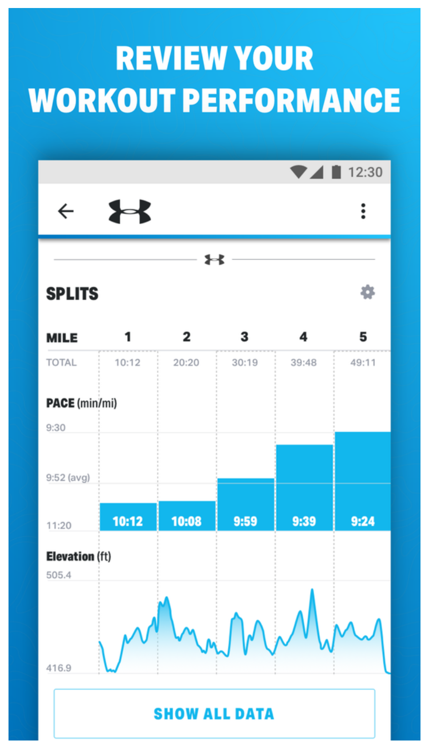
- Jedná se spíše o aplikace pro více technicky zaměřené uživatele.
- Nabízí hromadnou správu zařízení a sledování jejich polohy. Zdá se být velmi vhodná pro firemní nasazení a tím udržovat přehled o firemních zařízeních.
- Aplikace je zdarma.
- Podpora pro Android a iOS.
- Více informací lze nalézt na jejich oficiálním portále[23p].

Myšlenka administračního portálu a správa více zařízení se mě držela velmi dlouho a celý tento nápad aplikace mě následně přivedl k návrhu[2.1] s výzvami mezi uživateli, kde administraci představovaly právě samotné výzvy a uživatelé, kteří je mohou spravovat.

Map My Run by Under Armour

Aplikace, která je spíše sportovněji zaměřená. Nabízí podobnou sadu funkcí, kde základem je historie cesty, kterou daný uživatel urazil. Zajímavostí je však využití těchto dat pro vizualizaci cvičebního výkonu daného jedince, na jehož základě doporučuje i cvičební plány. Mezi důležité vlastnosti aplikace patří:

- Nabízí trasování vaší polohy a procházení její historie na základě které vytváří grafy (viz obr. 2.2) efektivnosti cvičení.
- Doporučuje cvičení na základě výkonu podaného za použití této aplikace.
- Aplikace je zdarma.
- Podpora pro Android a iOS.
- Detailnější informace lze najít na oficiálních stránkách, které jsem zmínil v této práci[23ac].



Obrázek 2.2: Funkce vykreslování statistik v aplikaci Map My Run[23ab].

Glympse

Aplikace nabízí možnost vidět ostatní uživatele ve své aplikaci na mapě, i když nejsou součástí vaší skupiny[Kay15]. Charakteristickými vlastnostmi jsou:

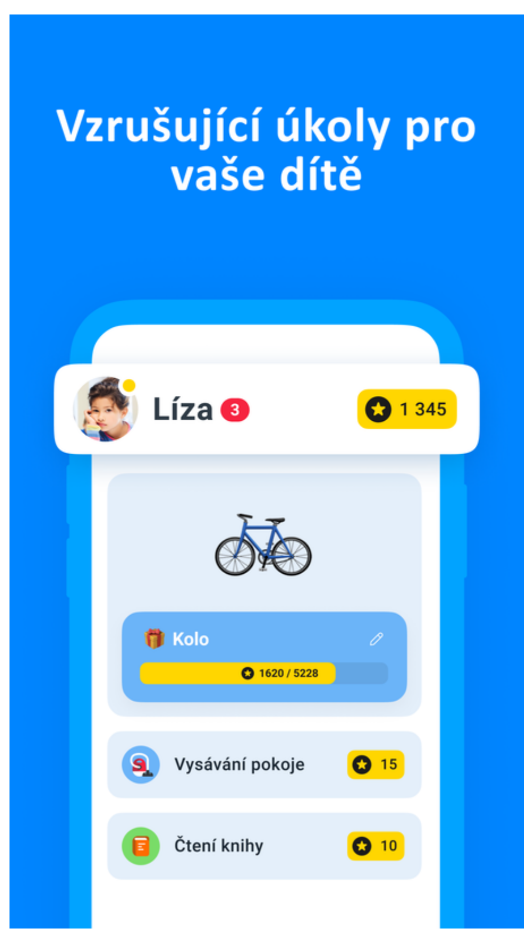
- Nabízí standardní sdílení polohy mezi uživateli ve stejné skupině jako předchozí konkurenti.
- Umožňuje sdílet polohu a vytvářet si přátele.
- Aplikace je zdarma a nabízí i placenou verzi rozšiřující základní funkce aplikace.
- Podpora pro Android a iOS.
- Konkrétní informace lze dohled prostřednictvím oficiálních webových stránek[23r].

Glympse s sebou přináší zajímavou funkci zobrazení polohy ostatních uživatelů na mapě bez ohledu na členství v určité skupině. Tato funkce taktéž přispěla k mému finálnímu návrhu systému.

Find My Kids

Jedná se o aplikaci zaměřenou přímo pro kontrolu dětí rodiči. Nabízí širokou škálu funkcí od nahrávání zvuku okolí, přehled užívaných aplikací a dokonce i možnost vytvářet pro děti úkoly, za které dostávají odměny (viz obr. 2.3). Charakteristika aplikace:

- Aplikace je zdarma.
- Podpora pro Android a iOS.
- Podrobnější informace lze dohled na stránce obchodu ***Google Play***[23n].



Obrázek 2.3: Ukázka definování úkolů v rámci aplikace Find My Kids[23ab].

Tento nápad se začíná velmi podobat tomu, čeho bych chtěl sám dosáhnout. Určitě bych ho rád rozšířil a prosadil pro širší skupinu, než jsou jen děti a rodiče, více však v další sekci 2.3.2.

2.3.2 Aplikace s herními prvky

Během mého vývoje prošla myšlenka několika fázemi. Ve chvíli, kdy jsem začínal do samotné trasovací aplikace vytvářet mnoho herních prvků, tak jsem si uvědomil, že se tím mění i konkurenční trh. Začal jsem tedy prozkoumávat aplikace z herního prostředí založené na GPS. Mezi těmito jsou převážně zástupci ***RPG***[2] a různých způsobů ***geocaching***[3] aktivit. Na rozdíl od předchozí sekce, zde popíšu aplikace, které přispěly k rozvoji systému.

Podle portálu ***Digital Trends***[Wil22] mezi nejpopulárnějšími je ***Ingress Prime***[4] (dále pouze Ingress) a ***Pokémon GO***[5]. Z vlastní zkušenosti obě dvě si jsou v jádru podobné. V rámci mapy jsou definovány kontrolní body, se kterými může hráč interagovat a sbírat potřebné předměty. Tyto kontrolní body lze také zabírat svojí frakcí, ke které jste jako hráč přiřazeni. Zatímco je Pokémon GO více sběratelská záležitost jednotlivce, Ingress je více zaměřený na spolupráci a celkový postup celé vaší frakce. Každá však nabízí trochu jiné možnost interakce s ostatními hráči (uživateli). Pokémon GO nabízí možnost interakce s ostatními hráči prostřednictvím bitev, kterých se lze účastnit, pokud jsou hráči ve vzájemném dosahu. Jedná se tak prakticky o jedinou možnost interakce, pomineme-li možnost obchodování. Na druhé straně Ingress nabízí možnost textové komunikace v rámci frakce ve zvolené vzdálenosti podle lokace hráče na mapě, tj. nelze vést privátní konverzaci mezi dvěma hráči.

Mezi další představitele aplikací, které lze považovat jako možnou konkurenci na tomto poli her, je rovnocenně umístěno: ***Orna: The GPS RPG***[6] (dále jen Orna) a ***Magic Streets: The GPS realm***[7] (dále jen Magic Streets). Obě dvě hry jsou si velice podobné a na první pohled nelze najít mnoho rozdílů. Podívám-li se na funkce, které obě dvě hry přinášejí, tak jde především o RPG možnosti, které obě dvě hry nabízejí, ale co je důležitější, tak je interakce mezi hráči. Orna s sebou přináší možnosti klanů, v rámci kterých lze spolupracovat s ostatními hráči a bojovat proti ostatním. Zajímavostí a možnou úvahou nad další implementací mého systému je fakt, že nepřátelé, kteří jsou ovládáni počítačem se neobjevují náhodně okolo hráče jako je tomu např. u Pokémon GO nebo jiných, ale mají fixní pozice, na kterých se pravidelně objevují.

[2] Role-Playing Game je druh hry, ve které hráči zaujímají role fiktivních postav, za které podle daných pravidel v samotné hře jednají.

[3] Geocaching je hra na pomezi sportu a turistiky, která spočívá v použití GPS při hledání skryté schránky nazývané „cache“ o níž jsou známy její zeměpisné souřadnice

[4] Online hra více hráčů s rozšířenou realitou, kterou vytvořilo studio ***Niantic Labs***[23v].

[5] Mobilní videohra na principu rozšířené reality, vyvíjená společností ***Niantic Labs***[23am].

[6] Klasické tahové GPS RPG s komplexními mechanikami a interakcí s ostatními hráči[23ak].

[7] Tahové RPG více zaměřené na hledání nových předmětů, které lze pak uplatnit v boji s prostředím generovaným na základě polohy GPS hra více hráčů s prvky RPG[23y].

Velmi nápaditou geocaching aktivitu přináší i aplikace ***MUNZEE***[8], která mezi uživatele přináší hledání předmětů v reálném světě, které jsou prezentovány pomocí QR kódů. Tyto QR kódy jsou mezi hráči distribuovány, udržovány a následně hledány. Oproti ostatním zde hledáme fyzické předměty na rozdíl od virtuálních.

Poslední hru, kterou jsem shledal v jistém směru unikátní je ***Resources Game***[9]. Nabízí hráčů vytvářet vlastní body na mapě, které jsou viděny všemi ostatními hráči. Tyto body představují například doly, které těží dané suroviny. Na základě GPS lokace jsou stanovená ložiska a hráč si může vybírat od méně vhodného až po více vhodné místo, kde svůj důl umístit. Bod zlomu přichází v dobu, kdy více hráčů chce okupovat stejné území. O toto území se následně strhne boj. Zde se vývojáři dostali od předem stanovených bodů až k bodům, které si hráči vytvářejí sami.

2.4 Rozbor návrhu pro budoucí vývoj

Po získání informací z analýzy zmíněné v sekci 2.3 lze usoudit, že nebude snadné se prosadit. Aplikací, které vynikají není mnoho, ale zároveň ty které jsou, tak svoji funkci odvádějí při nejmenším dobře. Můj systém bych nechtěl vytvořit čistě jako hru, ale zároveň ani ne jako další aplikaci pro rodičovskou kontrolu. Rád bych tyto dva světy spojil a vytvořil tak trasovací systém, kde sociální stránka bude hrát svoji důležitost. Již zmíněné hry, které jsem sledoval měli tuto stránku značně limitovanou pouze do interakce bitev či jiných aktivit spjatých s danou tématikou hry. Nejvolnější interakci mezi hráči měl Ingress Prime, který hráčům nabízel možnost textové komunikace v blízkém rozsahu dle volby.

Samozřejmě není mým cílem vytvořit uživatelskou aplikaci, která vytvoří hráči prostředí, kdy jakákoli verbální komunikace s hráči v blízkém okolí bude zcela zbytečná. Textová komunikace by měla nejlépe sloužit jako možný iniciační způsob komunikace nebo snadnější nalezení ostatních uživatelů. Zároveň tak nechci nutit hráče do verbální komunikace.

Vytvořením takové uživatelské služby přinese kompletně nový rozměr těchto mobilních her, kdy spojím svět sociálních sítí, her, sportu a cestování do jednoho. Vytvářím tím zcela něco odlišného. Je nutné vytvořit vhodný prototyp, který by popisované scénáře nabídl k vyzkoušení a následně pak provedl další analýzu. Rád bych uvedl několik základních příkladů užití s použitím základních funkci v následující sekci 2.4.1.

[8] Geocaching aplikace v pravém slova smyslu[23ai].
[9] Ekonomicky zaměřená GPS hra více hráčů[23as].

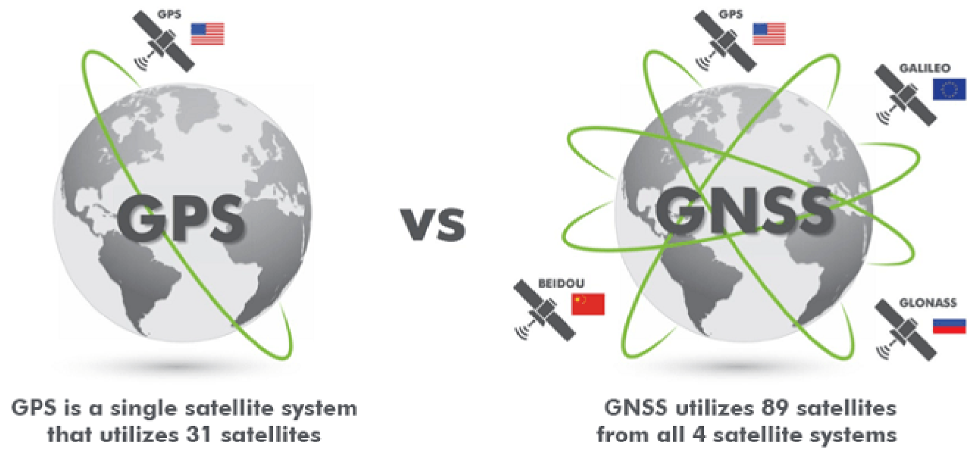
2.4.1 Příklady použití v reálném životě

Mezi možnou cílovou skupinu se řadí středoškoláci, samozřejmě ale nejsou jedinou cílovou skupinou. Systém by měl být schopen zaujmout jakéhokoli uživatele různého věku a zájmu. Nabízí to velmi vhodné prostředí, kde uplatnit přátelskou rivalitu mezi spolužáky ve formě výzev. Výzvy mohou plnit během přestávek, volných hodin nebo na cestě domů či do školy. Každým dalším dnem lze porovnávat svůj postup s ostatními a udržet si tím zápal do hry.

I přes to, že hlavním zaměřením je vytvoření herního prostředí, tak i mimo to, aplikace může nabízet možnost rodičovské kontroly, nikoli však v pravém slova smyslu, ale formou hry pro nutné setrvání v dané lokalitě. Bude tím vědět, že je jeho potomek celou dobu ve škole, pokud to bude daná lokace. Více to však spočívá v návrhu konkrétních trasovacích metod, které podrobněji popíši v kapitole 6, která se bude týkat možnosti rozšíření.

2.5 Možné způsoby trasování

GNSS, neboli Globální navigační satelitní systémy, jsou globální systémy určování polohy založené na použití satelitů umístěných na oběžné dráze Země. Nejznámějšími a nejpoužívanějšími GNSS jsou ***GPS*** (Americký systém), ***GLONASS*** (Ruský systém), ***Galileo*** (Evropský systém) a ***BeiDou*** (Čínský systém) [23s]. Na ilustraci 2.4 níže lze vidět vztah mezi GNSS a GPS, pro lepší představu.



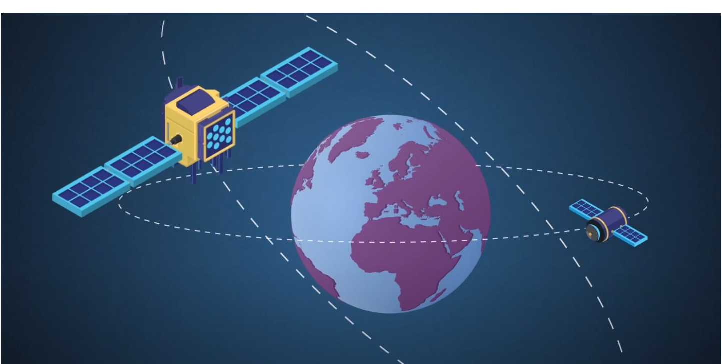
Obrázek 2.4: Ilustrace rozdílu mezi GPS a GNSS [20o].

GPS (Global Positioning System) byl vyvinut v USA a první satelit byl vypuštěn do oběžné dráhy v roce 1978. Systém se skládá z 31 aktivních satelitů umístěných na oběžné dráze Země. GPS umožňuje určení polohy a času na základě signálů vysí-

laných z těchto satelitů. Přesnost pozice dokáže zaměřit v rozmezí 3,5-7,8 metrů. Systém je dlouhodobým standardem a nabízí stabilní zázemí pro sledování polohy na Zemi [Rut22].

GLONASS (Globalnaya Navigatsionnaya Sputnikovaya Sistema) byl vyvinut v Rusku a první satelit byl vypuštěn do oběžné dráhy v roce 1982. GLONASS se skládá z 24 satelitů umístěných na oběžné dráze Země. Systém umožňuje určení polohy a času pomocí signálů vysílaných z těchto satelitů. Přesnost tohoto systému je mezi 5-10 metry a systém je přesnější ve vyšších nadmořských výškách [Rut22].

Galileo byl vyvinut Evropskou unií a první satelit byl vypuštěn do oběžné dráhy v roce 2011. Systém se skládá z 22 aktivních satelitů umístěných na oběžné dráze Země. Galileo umožňuje určení polohy a času pomocí signálů vysílaných z těchto satelitů. Jedná se o velmi nově nasazený systém a uvádí se, že plného potenciálu dosáhl teprve v roce 2020. Překvapivé u tohoto systému je jeho přesnost, která překonala přesnost GPS, a to na méně než metr. Tento satelitní systém je na vyšší oběžné dráze, a proto nabízí lepší přesnost a pokrytí, zejména na pólech Země (viz obr. 2.5). Tento systém má vyšší přesnost v Evropě než v jiných částech světa [Rut22].

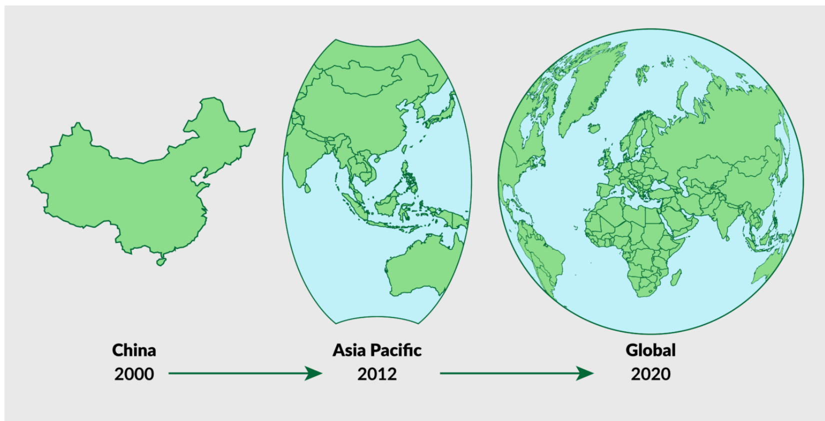


Obrázek 2.5: Galileo satelity se srovnáním k GPS.

BeiDou byl vyvinut v Číně a první satelit byl vypuštěn na oběžnou dráhu v roce 2000. Od toho roku měla s tímto satelitním systémem dlouhodobé plány (viz obr. 2.6). Systém se skládá z 35 aktivních satelitů umístěných v oběžné dráze Země. BeiDou umožňuje určení polohy a času pomocí signálů vysílaných z těchto satelitů. Přesnost tohoto systému je uváděna do 10 metrů. Tento systém má vyšší přesnost v Asijských zemích než ve zbytku světa [Rut22].

Dalším navigačním menším satelitním systémem, který stojí za zmínku, je ***QZSS*** (Quasi-Zenith Satellite System). Tento systém byl vyvinut v Japonsku a skládá se z čtyř satelitů, které se pohybují po oběžné dráze, tak aby alespoň jeden z nich byl

téměř vždy nad Japonskem. Tento systém umožňuje vyšší přesnost a spolehlivost při určování polohy v oblastech s vysokou zástavbou budov a v údolích. Navíc QZSS může pracovat v režimu s ostatními GNSS systémy, jako jsou GPS a GLONASS, což zvyšuje celkovou přesnost určování polohy [Rut22].



Obrázek 2.6: Ilustrace postupu vývoje BeiDou GNSS [ZHA20].

Odlišným způsobem pro možnost trasování je už samotná práce s již zmiňovanými systémy a to například pomocí tzv. ***Geofencing***[23q] pro systém Android. Umožňuje nám vytvoření oblastni, kterou lze následně na zařízení kontrolovat, zda ji dané zařízení tuto oblast opustí či do ní vstoupí. Tato možnost by v budoucím vývoji mohla značně usnadnit a optimalizovat způsob trasování, když by bylo nutné aplikaci spustit pouze na pozadí, tj. uživatel by aktivně nevěnoval pozornost aplikaci.

2.6 Očekávaný cíl této práce

Zmiňované nápady jsou během na dlouhou trať a vývoj takového systému mohu odhadovat v nízkých jednotkách let, bavíme-li se o pouze jediném pracujícím vývojáři. To samozřejmě mě nijak neomezuje pro zpracování této práce. V této kapitole jsem vytvořil cíl. Tímto cílem bude vytvoření dostatečné demonstrace funkčního systému, který se bude moci využít pro tvorbu komplexní služby z pohledu byznysu.

Jak již jsem v této kapitole jednou uvedl, bude nutné vytvořit robustní infrastrukturu celého projektu, aby jsem byl schopný dosáhnout stabilního systému pro udržitelnost a další rozvoj. Z čistě inženýrského hlediska nemusí být dosažení úspěchu služby natolik zajímavé jako spíše technická stránka takového řešení. Z toho důvodu bych se chtěl primárně zaměřit na samotnou funkčnost, provedení celého systému a vytvořit tím stabilní programové zázemí pro rozvíjení nebo případně i jiné myšlenky pro trasování zařízení pomocí GPS.

Chtěl bych vytvořit funkční, robustní a stabilní serverovou infrastrukturu, která bude zpracovávat příchozí požadavky klientských aplikací trasující jejich uživatele. Zároveň by se tato práce neobešla bez hlavní části, kterou je klientská aplikace. Tato aplikace bude demonstrovat funkční autorizaci uživatelů, práci s mapou a interakci s ostatními uživateli. Jinými slovy by mělo jít o funkční aplikaci, která je však stále ještě velmi obecná a může zároveň sloužit jako vhodný architektonický příklad pro tvorbu práce zaměřenou na práci s mapami.

Výběr samotného způsobu trasování spočíval v analýze jednotlivých možností popisovaných výše. V dnešní době existuje relativně mnoho systémů, které k tomuto trasování použít, kde hlavním a pravděpodobně nejznámějším zástupcem je GPS. Výběr se může odvíjet od několika faktorů, kde prvním může být počet satelitů a další podpora na uživatelském zařízení. Každý z těchto systémů vysílá signál na určité frekvenci, což může mít vliv na přesnost určování polohy a spolehlivost signálu. Signály od jednotlivých systémů mohou být dostupné v různých částech světa, což může mít vliv na použitelnost systému v určitých oblastech. Každý z těchto systémů má odlišnou přesnost určování polohy, která může být ovlivněna mnoha faktory, například již zmíněný počet dostupných satelitů na oběžné dráze Země. Taktéž je velmi důležité zvážit standardy a fakt, že vývojem této práce není nic ihned nutné nasadit do produkce. Je nutné vytvořit funkční aplikaci, kterou lze ještě dále rozšiřovat a nabídnou příklad takové služby.

Výběr a popis použitých technologií 3

V této kapitole budu popsány technologie, které jsem zvažoval použít pro vytvoření tohoto systému. Následně bude proveden výběr použitých technologií včetně odůvodnění jejich výběru. Také budou specifikovány technologické požadavky na základě předchozí kapitoly 2 a diskutována různá omezení.

3.1 Technologické požadavky a omezení

Celý systém bude rozdělen do klientské části a serverové části, kde za klientskou část budu považovat uživatelskou aplikaci, která se bude připojovat a komunikovat se serverem na straně vývojáře, resp. poskytovatele služby. Z toho principu nejsem omezen nutně na použití jedné technologie. Bude třeba definovat vhodný způsob komunikace mezi oběma stranami, kde je nutné, aby obě dvě podporovali stejný způsob komunikace, který mezi nimi zavedu. Mezi hlavními technickými požadavky bude realizovatelný způsob komunikace mezi klientskou a serverovou stranou. V současné době zde připadají 2 vhodné možnosti realizace: ***REST API***[1] a ***socketová komunikace***[2].

Serverová část nemá takové omezení, jako je tomu na straně klientské a výběr technologie, konkrétně programovacího jazyka je tudíž volnější. Různé možnosti následně rozebereme více podrobněji v jedné z následujících sekcí. V této části se také dostáváme k perzistentní vrstvě, resp. jakým způsobem ukládat data a pomocí které technologie. Samotný výběr není příliš široký, ale každé řešení s sebou své klady a zápory. Podrobně je též popíši v jedné z následujících sekcí. Systém bude vyžadovat důkladné a efektivní ukládání geolokačních dat a především, možnost v těchto datech efektivně vyhledávat. V této práci nepůjde jen o vzdálenosti mezi konkrétními

[1]Také známé jako aplikační programové rozhraní, je sada pravidel, která definují, jak se mohou aplikace nebo zařízení komunikovat v souladu s principy REST[23az].

[2]Jedná se o rozhraní pro programování mezi procesorovou komunikaci mezi aplikacemi, v lokálním, distribuovaném systému nebo TCP/IP síťovém prostředí[22b].

body, ale nutné bude vyhledávat i v okruhu kolem jednoho bodu a cílem bude nalézt všechny body, které se v tomto okruhu vyskytují.

Jedním z možných rizik, které by mohlo nastat, je použití nevhodné technologie, pomocí které nebudu schopen komunikovat se zařízením, na kterém poběží klientská aplikace. Nechci si zamezit možnost použití kamery nebo integrace s jinou aplikací v telefonu, například.

3.2 Analýza technologických řešení v úvaze

V této sekci probereme technologie, kterými jsem schopen zpracovat systém.

3.2.1 Řešení serverové části

Každá se zmíněných technologií je stručně obecně popsána pro obeznámení se základními vlastnostmi dané technologie.

PHP

PHP[20l] je velmi populární skriptovací jazyk pro univerzální účely. Běžně se kód zpracovává na straně webového serveru pomocí PHP interpreta implementovaného například pomocí CGI[3]. Jazyk PHP je nabízí lepší podporu pro běh na systému UNIX, nicméně dokáže být provozován i na jiných operačních systémech, maximálně s mírně omezenou funkcionalitou.

V prostředí PHP existuje i několik velmi populárních frameworků, které by mohly mému systému nabídnout požadovaný kvalitní základ a možnost snadno rozšiřovatelného kódu. Mezi ty nejlepší se řadí například ***Laravel***[20j] a ***Symfony***[20n].

Laravel je open-source PHP web framework se záměrem vytvářet webové aplikace s použitím architektury MVC[4]. Základ tohoto frameworku je vystavěn právě již na zmíněném frameworku Symfony a dále tento základ rozšiřuje o vlastní funkce. Oba dva jsou si velmi podobné.

Java

Java[20h] je objektově orientovaný programovací jazyk pro obecné použití. Java je především známá svoji možností fungování na velkém množství zařízení s různými operačními systémy, tj je multiplatformní. Primárně je používaná pro desktopové

[3] Common Gateway Interface
[4] Model-View-Controller, architektonický model

aplikace a zároveň může zastat „back-end“ webového serveru.

Jedním z možných řešení jsou vlastní styly a šablony, které lze použít pro tvorbu uživatelské části, ale je zde více preferované použít standardní ***HTML***[20e] a ***CSS***[20b].

Javascript

Javascript[20i] je vysoko-úrovňový, často just-in-time kompilovaný jazyk. Jazyk mimo jiné také nabízí objektově orientované možnosti řešení.

HTML a CSS i Javascript jsou nejdůležitější technologie ve světě webu. Javascript nabízí interaktivní webová řešení a je nedílnou součástí webových aplikací. Jednou z největších výhod je, jakým způsobem funguje - a to na straně klientské aplikace. Všechny webové prohlížeče běžně používané a dostupné na trhu mají vestavěný Javascript engine pro možné spouštění Javascriptového kódu na straně klienta.

Javascript engine byl používán dříve jen v internetových prohlížečích, nicméně v dnešní době se používá i v serverových webových řešeních, běžně pomocí technologie ***Node.js***[20k]. Také se používá v řadě aplikací vytvořených pomocí populárních frameworků jako jsou například: ***Electron***[20c], ***React.js***[20m] nebo ***Ionic***[20g].

Electron by se dal zvážit jako možné řešení pro vyvíjený systém, i z pohledu toho, že by se do budoucna mohla být vhodná možná implementace uživatelského rozhraní i na straně serveru. Jedná se o multiplatformní řešení, které má již ve svém portfóliu velmi známé aplikace, které jej využívají. Jedná se o open-source framework spravovaný společností GitHub. Používá ***Chromium***[20f] pro renderování Node.js jako runtime. Electron je jedním z hlavních GUI[5] frameworků[20d].

C++

Rodinu programovacích jazyků C představuje i jeho objektově orientovaná verze C++. V základu nenabízí možnost tvorby grafického uživatelského rozhraní, ale jedná se o vysoce výkonný programovací jazyk, zejména pokud je efektivně programováno. Programovací jazyk je nativně podporován systémy UNIX, ale je možné jej provozovat i na jiných operačních systémech.

C#

Další z jazyků C, který je vyvíjen společností Microsoft. Jde o objektově orientovaný jazyk. Byl vyvinut jako součást .NET frameworku.

[5] Graphical User Interface

.NET framework[20a] je vyvinut nativně pro operační systém Windows. Zahrnuje rozsáhlou pojmenovanou knihovnu Framework Class Library (FLC). Programy napsané pro rozhraní .NET Framework se spouštějí v softwarovém prostředí, které má název Common Language Runtime (CLR). CLR je aplikační virtuální stroj, který poskytuje služby jako je zabezpečení, správa paměti a zpracování výjimek. Programový kód napsaný pomocí tohoto rozhraní .NET Framework se proto nazývá „managed code“. FCL a CLR společně tvoří .NET Framework.

3.2.2 Řešení klientské části

Každá ze zmíněných technologií je stručně a obecně popsána pro obeznámení se základními vlastnostmi.

Electron

Electron[20c] je open-source software framework. Je vhodný pro vytváření aplikací pomocí webových technologií (hlavně ***HTML***[20e], ***CSS***[20b] a ***Javascript***[20i]). Výstup je renderován pomocí enginu prohlížeče ***Chromium***[20f] a „back-end“ pomocí ***Node.js***[20k].

Electron by se dal zvážit pro vytvořeného navrhovaného systému hlavně z důvodu širokého používání a velké komunity, která je kolem tohoto frameworku vybudována[20d].

Flutter & Dart

Flutter[23o] byl navržen, aby splnil požadavky společnosti Googlu, pro které jej společnost také vyvinula. Google doporučuje použít programovací jazyk ***Dart***[23j] z důvodu jeho statického typového systému[Fan23]. Jedná se o velmi výkonný framework a pravděpodobně nejvýkonnější ze všech zmíněných v této sekci.

Může být obtížné začít v tomto frameworku programovat a je možné, že učení tohoto frameworku nebude tak přímočaré jako tomu může být u jiných. Tento framework má výhodu, že může být použit i pro vývoj webových aplikací, i přes jeho možnosti pracovat s nativními prvky zařízení. Neumožňuje však snadné znovu použití kódu pro jinou platformu, pokud pracuji nad konkrétním nativním prvkem.

React Native

React Native[23aq] je open-source UI software framework založený na skriptovacím jazyce Javascript. Je nutné si jej neplést s React.js, jedná se o odlišný framework,

avšak vychází z podobného nápadu. Umožňuje vytvářet aplikace na Android i iOS zařízení, stejně jako vývojový framework Flutter.

Již z názvu vychází, že se jedná o framework zaměřený pro práci s nativními prvky zařízení. React Native nabízí vlastní API, které může být použito při vývoji aplikací pro iOS a Android. Je možné tak programovat sdílené komponenty mezi např. iOS a Android aplikacemi.

Unity

Unity[23aw] je multi-platformní herní engine vyvíjený společností Unity Technologies. Je velmi populární pro vývoj na Android a iOS. Je považován za velmi jednoduchý k použití a naučení a je velmi populární pro ***indie***[6] hry.

Logika vytvářeného systému je programovaná pomocí jazyk C#. Zbytek nastavení celého herního enginu je velmi specifický a probíhá pomocí uživatelského rozhraní editoru Unity. Výhodou v tomto případě je, že nabízí herní engine jako takový, to znamená, že veškeré animaci, případně práce ve 3D bude mnohonásobně jednodušší než s ostatními zmíněnými technologiemi.

3.2.3 Řešení vykreslování map

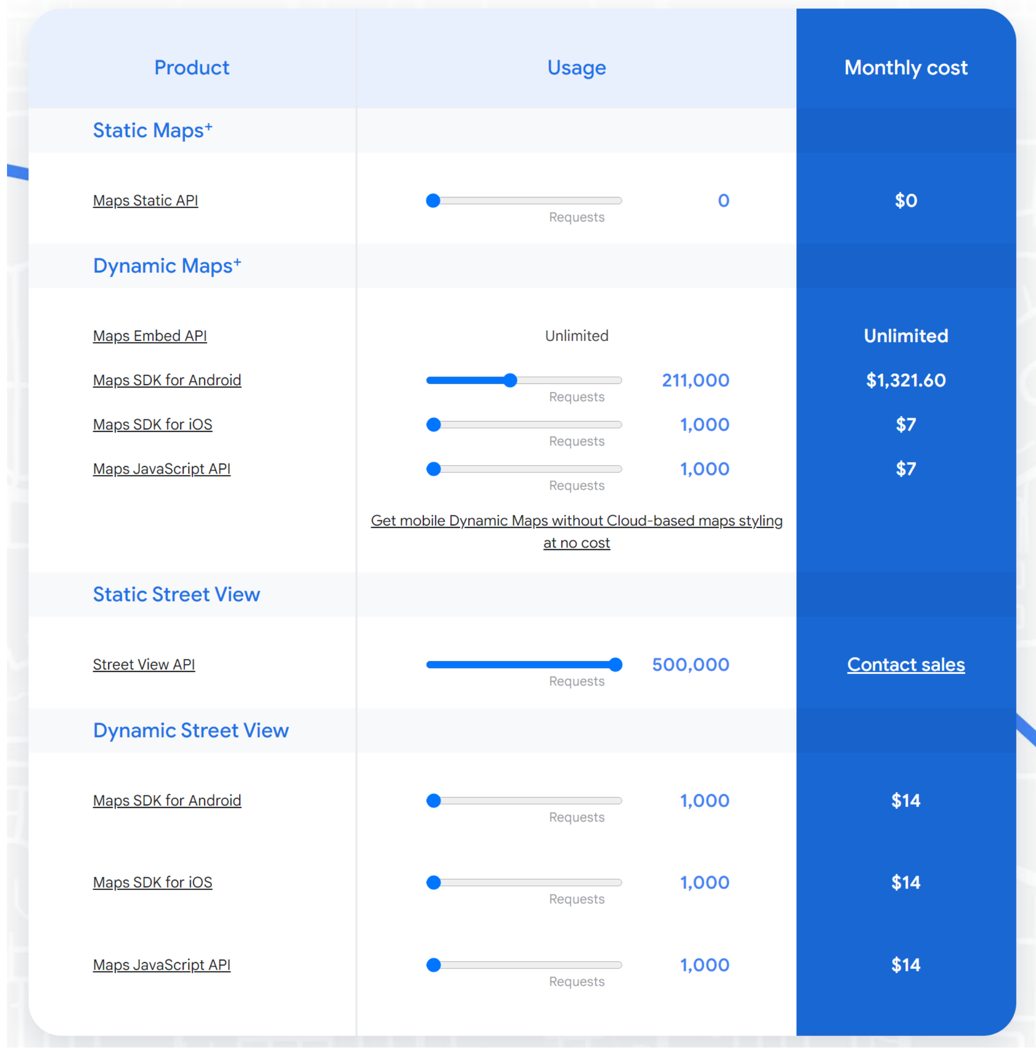
Google Maps

Google Maps je webové mapová platforma poskytovaná společností Google. Nabízí satelitní záběry, záběry ulic a pohled v úhlu 360, „real-time“ dopravu, hledání cest a další.

Z pohledu API, Google Maps nabízí platební model „pay-as-you-go“, kdy platím za poskytnutí jednoho volání API (viz obr. 3.1). Avšak, pokud nepřesáhneme cenovou hranici $200, tak lze použít API zdarma.

Google Maps také nabízí (resp. nabízelo) možnost tzv. Gaming Service, která umožňovala mapy vykreslovat ve 3D. Vytvářelo to velmi vhodné prostředí pro start nového projektu pro herní účely. Bohužel, se Google z nespecifikovaných důvodů tuto službu stáhl ze svého portfolia poskytovaných služeb[23t].

[6] Nezávislá videohra, většinou vytvořena jednotlivcem nebo malým týmem.



Obrázek 3.1: Náhled zprostředkování API pro Google Maps[23u].

OpenStreetMaps

OpenStreetMaps[23aj] je otevřená geografická databáze aktualizovaná a udržovaná komunitou dobrovolníků. Tato služba je nabízena plně zdarma a mnoho jiných mapových služeb, například Mapbox z této databáze vychází. Jde o surová data, a může být tedy obtížné pomocí této databáze něco začít tvořit. Na druhou stranu to dává největší možnou volnost, kdy vše si můžeme udělat podle svého.

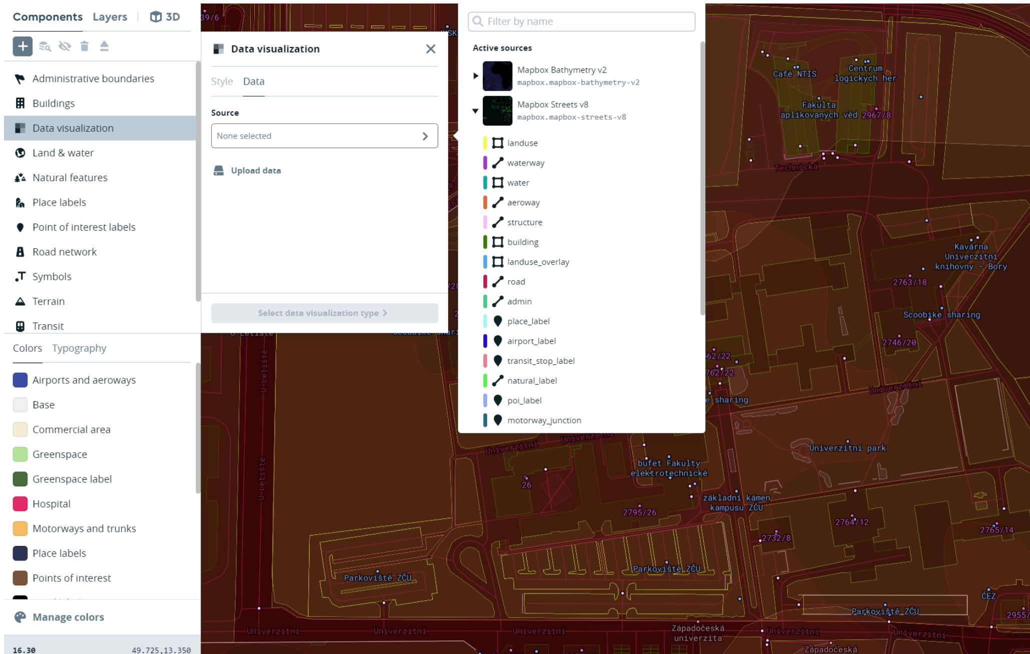
Mapbox

Mapbox[23aa] je moderní technologie pro tvorbu a vizualizaci map. Mapbox vedle samotných map nabízí i vlastní editor, kde si lze definovat vlastní styl map a velmi širokou možnost filtrování zdrojů mapy a dat, která chci používat a stahovat (viz obr.

3.2). Vykreslování map v Mapbox je založeno na využití mapových vrstev a stylů. Uživatelé mohou vytvářet vlastní mapové vrstvy, které obsahují různé prvky, jako jsou body, linie, plochy, text a tím si mapu přizpůsobit dle vlastních potřeb[Dzi23].

Platební model je velmi podobný jako již zmíněné Google Maps „pay-as-you-go“ s jedním menším rozdílem. V tomto případě zde neplatíme za počet API dotazů, ale za počet aktivních uživatelů. Počet API požadavků, který každý uživatele provede není důležitý.

Mapbox také vyvíjí vlastní knihovny pro práci s Unity engine a umožňuje tak snadnější přístup pro práci s mapami v herním světě.



Obrázek 3.2: Náhled do Mapbox editoru mapového zdroje dat.

ArcGIS

ArcGIS[23d] je technologie pro správu, analýzu a vizualizaci prostorových dat. Tato technologie umožňuje uživateli vytvářet, sdílet a používat mapy pomocí geografického informačního systému (GIS) na základě různých zdrojů prostorových dat.

Vykreslování map v ArcGIS zahrnuje mnoho funkcí a nástrojů, které umožňují uživateli vytvářet a upravovat mapové vrstvy, symboly atd. Uživatelé mohou také použít různé efekty, jako jsou stíny, hladiny, hladké linie a gradienty, aby dodali

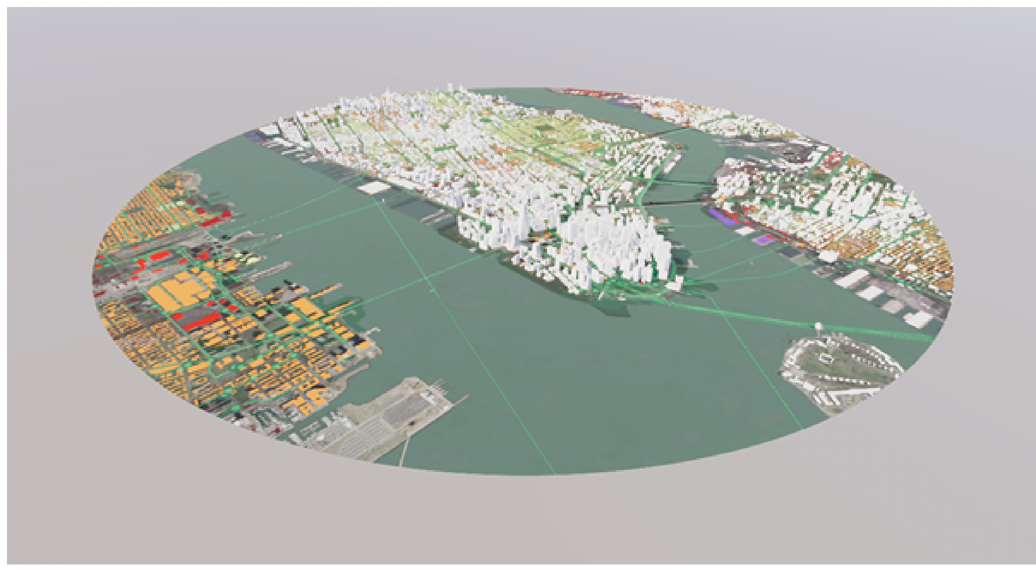
vizuální složku svým mapám.

Dotazování na mapové API v ArcGIS umožňuje uživatelům získávat prostorová data z různých zdrojů a používat je pro analýzu a vizualizaci.

ArcGIS také poskytuje uživatelské rozhraní a API pro tvorbu a sdílení mapových aplikací, mimo jiné také nově vyvíjenou knihovnu pro Unity engine.

Platební model podobně jako o konkurentů výše uvedených je „pay-as-you-go“ na počet vykreslovaných dlaždic mapy.

Tato poskytovaná služba je velmi podobná zmíněnému Mapboxu, avšak v řadě věcí se liší. Zajímavou vlastností, kterou nabízí oproti Mapboxu je možnost vykreslování mapy kolem daného bodu ve výseči, nikoli pouze v mřížce (viz obr. 3.3).



Obrázek 3.3: Vykreslení mapy pomocí ArcGIS.

3.2.4 Řešení datového uložiště

Dále budou uvedena řešení, která jsou vhodná pro uchovávání mapových dat a zároveň to jsou robustní a moderní technologie pro uchovávání dat v běžných systémech a aplikacích.

PostgreSQL + PostGIS

PostGIS[23an] je rozšíření relačního databázového systému ***PostgreSQL***[23ap], které umožňuje práci s geografickými daty. Samotný PostgreSQL je relační databázový systém, který umožňuje práci s různými typy dat, včetně geografických dat.

Dotazování na mapové API v PostGIS je založeno na využití prostorových funkcí, které umožňují uživatelům dotazovat se na prostorová data uložená v databázi. Tyto funkce umožňují uživatelům provádět prostorové dotazy, jako jsou například hledání bodů v určité oblasti, nebo výpočet vzdálenosti mezi body. Uživatelé mohou také použít prostorové funkce k analýze dat, jako je například seskupování dat podle určité oblasti nebo vypočítání průměrné vzdálenosti mezi body.

MongoDB

MongoDB[23af] je No-SQL (dokumentová) databáze, která umožňuje ukládat a pracovat s různými typy dat, včetně geografických dat. MongoDB obsahuje vestavěnou podporu pro geografická data a umožňuje ukládání geografických dat jako součástí dokumentů. MongoDB lze použít jako back-endovou databázi pro různé mapové aplikace, včetně vykreslování map a dotazování se na mapové API.

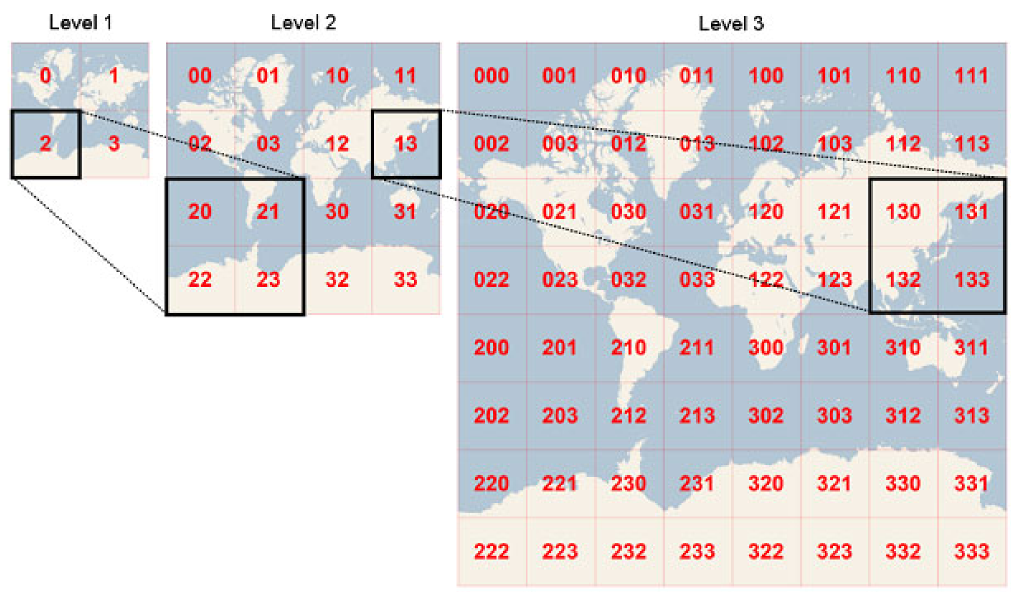
Dotazování na mapové API v MongoDB lze provést pomocí geografických dotazů, které jsou poskytovány pomocí vestavěné geografické podpory v MongoDB. Tyto dotazy umožňují uživatelům vyhledávání geografických dat, jako jsou například hledání bodů v určité oblasti, nebo výpočet vzdálenosti mezi body. MongoDB také poskytuje funkce pro filtrování a řazení dat, což je užitečné pro rychlé a efektivní dotazování na velké množství dat.

3.3 Výběr technologií pro zpracování

Z výše uvedených technologií je vytvořený široký výběr. Unity by byla vhodná volba pro tvorbu uživatelské aplikace, když je důležité vytvořit herní prostředí a aby se aplikace v budoucnu chovala jako plnohodnotná hra. I když je hlavním tématem této práce trasovací část aplikace, tak abych byl schopný vytvořit ucelený systém je nutné nejdříve vytvořit server, který bude tvořit stabilní „jádro“ tohoto systému.

V oblasti programování serverových aplikací mám největší zkušenosti s .NET, resp. s programovacím jazykem C#. PHP by bylo vhodné pro menší webové aplikace, pro sestavení serveru mi dle vlastních zkušeností nijak zvlášť nevyhovovalo ani nesedělo do návrhu a smyslu serveru, i s faktem toho, že nové dnešní verze PHP jsou vysoce kvalitní na dnešní poměry, podle informací získaných z oficiálních stránek.

Dále jsem přemýšlel, jestli by nebylo vhodnější server programovat v jazyce C++ z důvodu výhody jeho možných optimalizací a efektivnosti kódu. Tato myšlenka rychle odpadla z důvodu převažujících zkušeností v jazyce C#, který jsem tak zvolil jako ten, ve kterém budu serverovou část vyvíjet. Nejde pouze jen o zkušenosti, které s tvorbou v tomto jazyce mám, ale také budoucnost platformy .NET, která v posledních letech vytváří velmi kvalitní a moderní prostředí, a to přibližně od doby sloučení celého frameworku .NET 5.0. I přes tento fakt, jsem dlouho uvažoval, zda nepoužít jeden z Javascriptových frameworků, které jsou z mé zkušenosti v poslední době velmi populární a rozmanité. Po několika dnech rozhodování jsem usoudil, že bude vhodné zůstat u C# z důvodu převažujících zkušeností a faktu, že je nutné vytvořit robustní server, který bude představovat minimální možnou míru návrhových chyb. Z pohledu výkonnosti jsem měl možnost vytvořit benchmark v předmětu KIV/VSS starších standardních verzí .NET oproti právě zmíněným novým verzím. Podle mých výsledků nebo výsledků, které jsou snadno vyhledatelná v online diskusích, je vidět vysoké zlepšení[23g].



Obrázek 3.4: Ukázka vyhledávání a sestavování řetězce, resp. geohash.

Z návrhu systému lze předpokládat, že při dlouhodobém užívání celého systému bude přibývat i velké množství dat. Zároveň potřebuji takové uložiště, které bude schopné ukládat geografická data. Jako nejvhodnější jsem vyhodnotil databázový systém PostgreSQL, který výše popisuji. Mezi dalšími relativně populárními je také MongoDB, která poskytuje také efektivní uložiště pro geografická data. Vybral jsem dle své volby nejlepšího zástupce pro relační databáze a No-SQL databáze. V mém případě je nutné v systému v mnoha případech udržet konzistentnost, zejména

mezi vazbami uživatelů. Teorém ***ACID***[7] běžně splňují relační databáze. Avšak, je nutné, aby tato technologie (PostgreSQL + PostGIS) podporovala nejzákladnější hledání, a to v mém případě je vyhledávání požadované vzdálenosti od daného bodu. Porovnával jsem přístup systému MongoDB, tak i přístup systému PostGIS a hledal rozdíly a možná úskalí. Výkonnostní rozdíly nebyli významně odlišné[Dom23]. Mezi hlavní porovnávání, které jsem vyhledával bylo však porovnání funkcí, které dokáží ve svých datech vyhledávat dle již zmíněného způsobu. Těmito metodami jsou: `ST_DWithin`[23ao] pro PostGIS a `geoNear`[23ae] pro MongoDB. Obě dvě metody fungují na principu ***geohash***[8], pro který je velmi důležité, jakým způsobem jsou data ukládána.

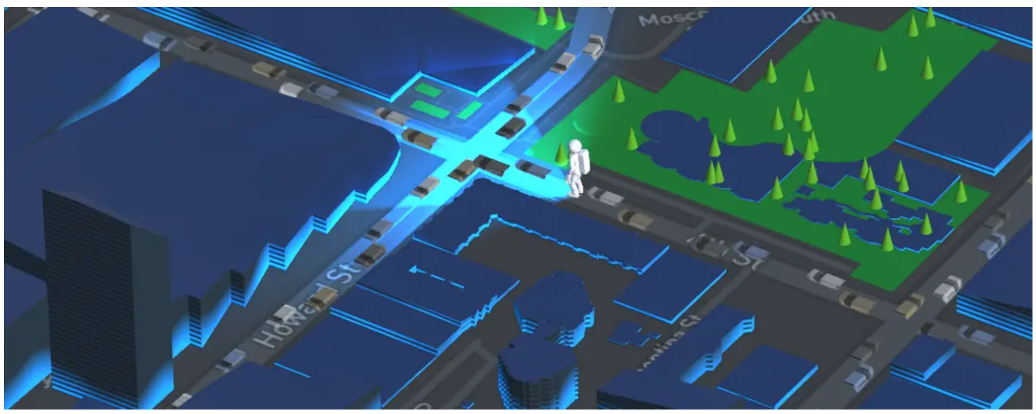
Výběr technologií pro vývoj klientské aplikace byl nejsložitější, protože nabízel nejvíce rozmanité možnosti. Z prvopočátku, jsem chtěl jít směrem Javascriptového frameworku. Postupně jsem získával více informací a požadavků na celý systém a potřeboval jsem rychlý a efektivní způsob pro práci s komponenty daných zařízení. Postupně se mi přestával líbit fakt, že by se jednalo pouze o prohlížeč zobrazený v mobilní aplikaci a začal jsem přemýšlet o možných alternativách, které by však umožňovali multiplatformní vývoj. Delší dobu jsem se rozhodoval mezi technologií Flutter a React Native, se kterým jsem již měl základní zkušenosti. V tuto dobu, kdy jsem o těchto technologiích rozmýšlel, tak jsem předpokládal v použití API od Google Maps. Postupně jsem chtěl tuto myšlenku rozvinout a objevil možnost vykreslování 3D map, které Google Maps nabízely. Bohužel, Google se rozhodl tuto platformu ukončit a stáhnout z nabídky poskytovaných služeb (viz 3.2.3).

Postupně jsem analyzoval ostatní zmíněné technologie jako je například Mapbox či ArcGIS. Zvažoval jsem použití OpenStreetMaps, které nabízí bezplatné řešení, ovšem to s sebou přináší vytvoření vlastního API a čtení mapových dat. Ve srovnání s ostatními technologiemi to je výhodou, ale mnoho práce, které nejsou zcela součástí této práce a vyžadují o mnoho více času pro zpracování. Zaujal mě tedy Mapbox, který s sebou přináší také vlastní editor, který mi umožňuje si data na mapě přizpůsobit dle vlastních potřeb. Nejsme nuceni přenášet data, o která se můj systém zajímat nebude. Avšak, na druhé straně stála velmi podobně vybavená technologie ArcGIS. Nemluvě o možnosti obou dvou poskytující rozhraní pro Unity a zároveň tím možnost i vykreslovat snadno mapy ve 3D.

[7] Představuje základní vlastnosti: Atomicita, Konzistence, Izolace a Trvanlivost

[8] Geohash je unikátní identifikátor specifických regionů na Zemi. Základní myšlenkou je rozdělení Země do regionů uživatelem definované velikosti a přidělení unikátního ID. Zeměpisná šířka a výška jsou transformovány do textového řetězce (viz obr. 3.4), který je reprezentuje. Vyhledávání následně funguje v dané oblasti a jeho přiléhajících sousedech.

Rozhodl jsem se nakonec vybrat technologii, kterou nabízí Mapbox z toho důvodu, že nabízí lehce odlišný platební model a zároveň s sebou přináší editor zdroje map velmi bohatý na přizpůsobení. Obě technologie nabízejí „pay-as-you-go“ model, nicméně Mapbox je založený na počtu aktivních uživatelů a nikoli na počtu dotazů do API. Potřeboval jsem tuto technologii ověřit, zda bude fungovat tak, jak si představuji. Během tohoto procesu jsem objevoval nové možnosti, se kterými Mapbox pracoval a umožňoval vykreslovat 3D grafiku (viz obr. 3.5).



Obrázek 3.5: Ukázka vykreslování 3D map pomocí Mapbox[23x].

Tyto možnosti byly dostupné pro Unity. Začal jsem tedy zkoumat, pokud bych se rozhodl jít cestou vývoje v Unity, zda bych si tím nemohl uzavřít některé možné způsoby interakce se zařízeními, na kterých tato aplikace poběží. Z výkonnostního hlediska bylo téměř jasné, že se nebude jednat o výkonnější způsob implementace, ale po zvážení návrhových požadavků jsem usoudil, že použití Unity bude vhodná volba a bude s sebou přinášet i další výhody i na možnou nižší výkonnost samotné aplikace. Už jen fakt, že se jedná o vývoj v C#, tak to s sebou přinese možnost sdílet kód mezi serverem a klientskou aplikací, a to může být velice efektivní a užitečné zejména pro sdílení datových modelů použitých pro komunikaci.

Když jsem měl rozhodnuté technologie, které použít, musel jsem však ještě vyzkoušet jejich schopnost pro mé použití. Mimo téma této práce jsem vytvořil malý prototyp aplikace v Unity za pomoci mapové technologie Mapbox. Samotnou implementaci a vývoj zmíním v další kapitole 5.

Než se však k samotné implementaci dostanu, je nutné zanalyzovat možné architektonické řešení, které v rámci vybraných technologií použiji. Tyto řešení představím v následující kapitole 4. Výběr technologií se během samotné analýzy návrhu systému hodně měnil než jsem se dostal k finálnímu výběru.

4 Analýza architektonických řešení

Tato kapitola popisuje různé architektonické způsoby, ze kterých následně vyberu ta nejvhodnější pro vytvářený systém.

Software se postupem času mění, zvláště k těmto změnám dochází při agilním vývoji, kdy se vyvíjený projekt mění velmi rychle a často. Architektura může být přehlížena a postupně může být obtížnější ve vyvíjeném projektu propagovat změny. Je zde několik typů architektur, kde každá se snaží řešit jiný problém. Jeden z nejvíce známých a základních způsobů třídění projektu je rozdělení samotné infrastruktur způsobem MVC[1] architektury[23a].

Nejvýraznější typy architektur jsou však:

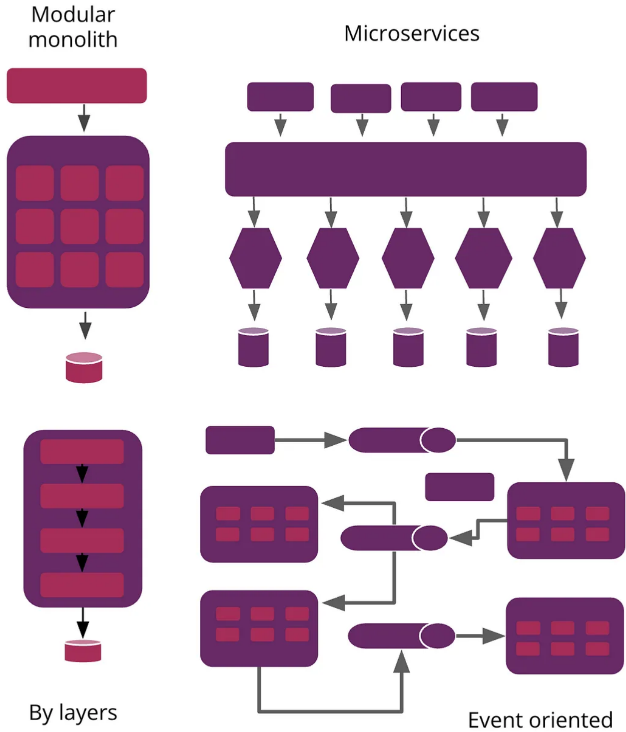
- Monolitická architektura
- Distribuovaná architektura

Monolitická architektura je taková, která kombinuje vrstvu uživatelského rozhraní, doménovou (business) vrstvu a databázovou (persistence) vrstvu do jednoho funkčního systému/celku. To znamená, že je systém zodpovědný za všechny kroky pro provedení uživatelských požadavků. Zástupcem takové architektury může být například vrstvená architektura nebo modulární architektura (viz obr. 4.1).

Distribuované architektury jsou takové, u kterých jsou na rozdíl od monolitu, jejich procesy distribuované. To znamená, že pokud provedeme dotaz do databáze, rozdělíme všechny procesy do různých uzlů, což může zlepšit výsledný výkon aplikace. Zástupci mohou být: Microservice architektura nebo Event-driven architektura (viz obr. 4.1).

[1] Model-View-Controller

Jak je vidět, je zde tedy mnoho typů architektur, které se liší svým účelem a složitostí. Mezi těmito architekturami neexistuje koncept „nejlepší“ architektury, protože vše závisí na kontextu problému.

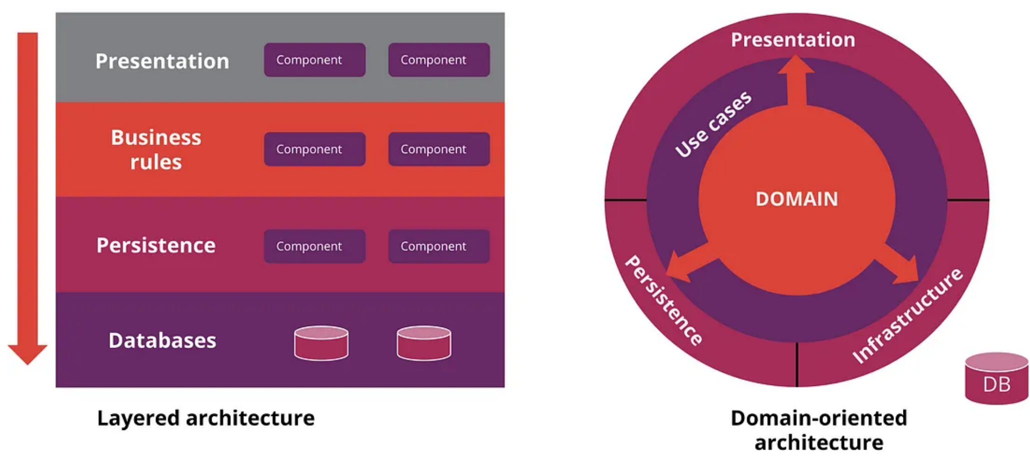


Obrázek 4.1: Graficky znázorněné typy architektur jako ukázka k přiloženému textu[Las22].

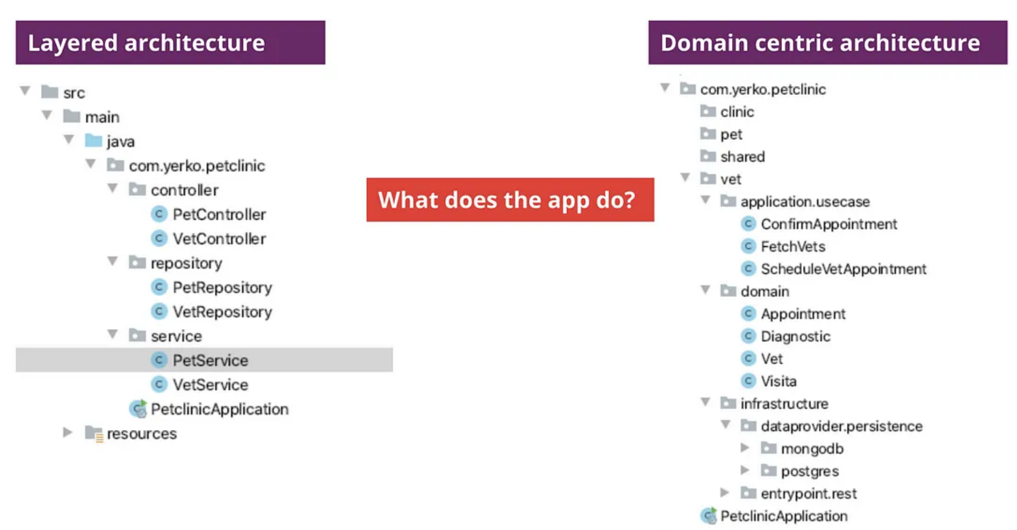
Existují však ještě principy, které nazýváme Domain-based design nebo také ***Domain Driven Design*** (DDD). Je to doménově zaměřený („domain-centric“) softwarový jazyk a přístup k modelován složitých problémů. Termín vytvořil Eric Evans ve své knize Domain Driven Design[Eva03]. Skládá se ze souboru vzorů, principů a postupů, které umožňují týmům soustředit se na to, co je zásadní pro obchodní úspěch, při vytváření softwaru, který zvládá složitost v technických a obchodních kontextech.

Ve vrstvené architektuře, závislost jde z prezenční vrstvy do vrstvy databáze, a to je vhodné pro jednoduché aplikace s malým rozsahem. Nicméně, jakmile roste složitost projektu, doménová pravidla (business rules) jsou pravděpodobně spojována do dalších vrstev a tím poroste složitost udržování projektu. Na druhé straně doménově zaměřená architektura funguje obráceně a doména je centrem všeho (viz

obr. 4.2). To znamená, že doménová logika nezávisí na technologiích, takže je mnohokrát jednodušší udržovat systém a zároveň provádět v něm změny, jako může být i výměna různých technologií. Na obrázku 4.3 lze vidět příklad porovnání souborových struktur stejného projektu s implementací vrstvené architektury oproti doménově řízené architektuře.



Obrázek 4.2: Rozdíl mezi vrstvenou a doménově orientovanou architekturou[Las22].



Obrázek 4.3: Rozdíl souborové struktury projektu s architekturou vrstvenou a doménově orientovanou[Las22].

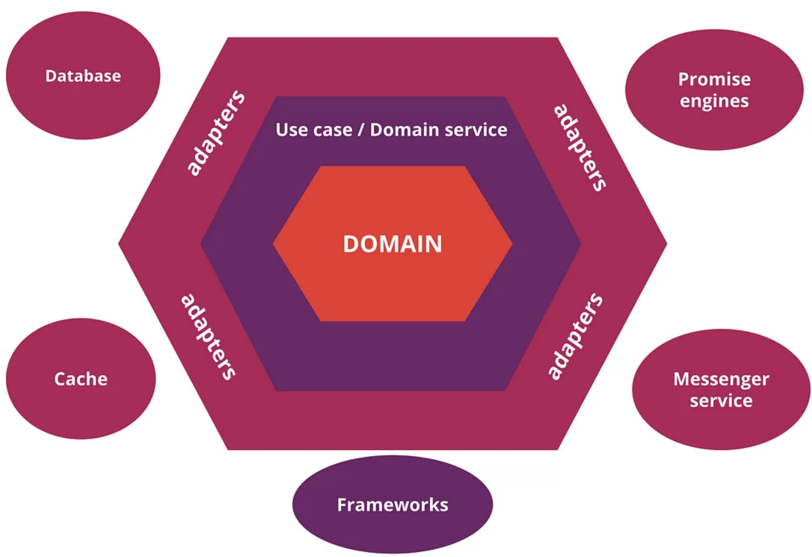
4.1 Architektonický design

V této sekci popíši několik architektonických přístupů, které zvažuji k implementaci.

4.1.1 Hexagonal Architecture

Hexagonal Architecture (viz obr. 4.4), také známá jako Ports and Adapters Architecture, se zaměřuje na oddělení jádra aplikace od vnějších závislostí. To znamená, že jádro aplikace, které obsahuje hlavní logiku, je odděleno od vnějších závislostí, jako jsou databáze, uživatelské rozhraní a další externí systémy. Poprvé byla prezentována v roce 2005. Koncept Hexagonal Architecture spočívá v tom, že vaše základní aplikační logika je napsána pouze s konceptem jakýchkoli externích závislostí, které má. V objektově orientovaném světě to znamená, že deklaruje rozhraní v jádře aplikace a implementaci ponechá mimo jádro [Dam21].

Jednou z hlavních výhod Hexagonal Architecture je možnost jednoduchého nahrazení externích závislostí, což umožňuje snadné testování aplikace a zlepšení její flexibility. Další výhodou této architektury je snadné škálování, protože jádro aplikace je odděleno od vnějších závislostí.



Obrázek 4.4: Příklad Hexagonal Architecture[Las22].

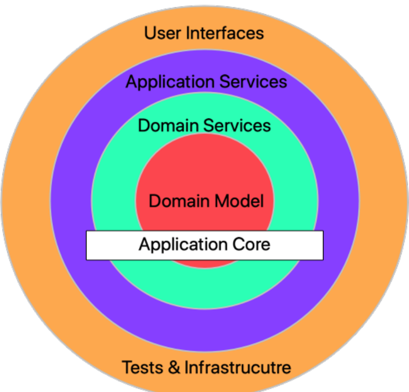
Nicméně, Hexagonal Architecture může být složitá na implementaci a vyžaduje vysokou úroveň abstrakce a rozdělení aplikace na samostatné moduly. Navíc, vzhledem k tomu, že jádro aplikace je odděleno od vnějších závislostí, může být obtížné spravovat a zajišťovat konzistenci dat. Zároveň důležitým faktem je, že konkrétně nespecifikuje, jak se má taková architektura implementovat. Ze tří uváděných DDD

architektur se jedná o nejstarší a vytváří tak základ pro ostatní [Ibe22].

Aplikační jádro Hexagonal Architecture lze považovat za „port“ (rozhraní), jako je port displeje nebo port USB. Vnější vrstva aplikace pak vytvoří „adaptér“, který se zapojí do portu, takže pokud by tam byl databázový port, adaptér by se do tohoto portu „zasunul“ a zajistil připojení ke konkrétní databázi.

4.1.2 Onion Architecture

Onion Architecture byla poprvé prezentována v roce 2008. Tato architektura dále rozvíjí myšlenku definice „jádra“ aplikace s několika dalšími vrstvami, které ho obklopují [Dam21].



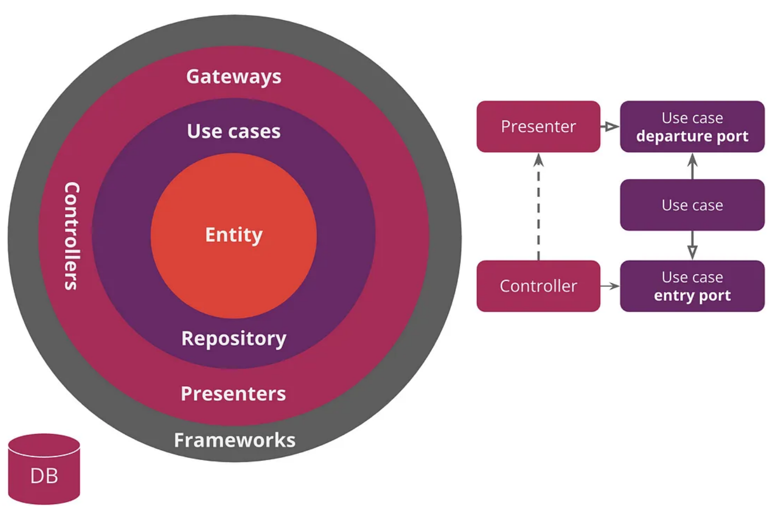
Obrázek 4.5: Příklad Onion Architecture[Tho23].

Jádro ***Domain Model*** reprezentuje entity a jejich „business rules“. Tato vrstva nemá žádné závislosti a představuje základní blok DDD. V další vrstvě ***Domain Services*** se chová jako „middle-man“ (servisní vrstva), kde udržujeme databázové migrace a objekty databázových kontextů, které jsou založeny na business pravidlech z vyšší vrstvy a vytváří tím jasné rozhraní pro zápis a čtení dat. Samotná implementace je však ponechána až nižším vrstvám. Následující vrstva ***Application Services*** se skládá z implementace „use cases“, které jsou nezbytné pro aplikaci a tvoří její základ. Poslední vrstvou okolo jádra je ***User Interfaces***, která definuje „business process“ aplikace - „Infrastructure“. V této vrstvě se nachází uživatelské rozhraní, externí infrastruktura apod. Tato architektura je navržena tak, aby umožňovala snadné škálování a údržbu aplikace [Kha22].

Nicméně, Onion Architecture může být komplikovaná a obtížná na implementaci, zejména pokud aplikace obsahuje velké množství funkcionalit.

4.1.3 Clean Architecture

Nejnovější architektura prezentovaná v roce 2012. Clean Architecture je architektura, která se zaměřuje na oddělení jednotlivých vrstev aplikace a udržování závislostí mezi nimi v minimu. Tato architektura je založena na principu ***Dependency Inversion***[2] a ***Single Responsibility***[3], což umožňuje snadné testování aplikace a zlepšuje její přehlednost a flexibilitu. Tato architektura vychází z návrhu Onion Architecture. Hlavním pravidlem této architektury je, že závislosti musí směřovat z externího kruhu k vnitřnímu a ne naopak (viz obr. 4.6).



Obrázek 4.6: Příklad Clean Architecture[Las22].

Clean Architecture mění parafrázi „Application Services“ na „Use Cases“ a staví je do popředí aplikace, takže je zde zcela jasné, co aplikace skutečně dělá. Podle obrázku 4.6 je zde nejvíce vnitřní vrstva ***Entity***, která reprezentuje business logiku. Zde je to místo, kde lze aplikovat DDD přístup a implementovat Agregáty[4], entity, „value objects“ a rozhraní služeb. Tato vrstva musí být izolována od zbytku aplikace. V další vrstvě ***Use cases*** představuje rozhraní s „okolním světem“ a stále je izolovaná od krajních vrstev, jako například databáze. V předposlední vrstvě rozhraní adaptérů (také známá jako ***Infrastructure*** layer) lze nalézt implementaci aplikace, tj. repository, databáze a podobně. Nejvíce krajní vrstva je pak složena z nástrojů jako jsou webové frameworky nebo třeba i samotný databázový systém [Ibe22].

[2]Princip inverze závislostí je princip návrhu, který říká, že vysokoúrovňové moduly by měly záviset na abstrakcích spíše než na konkrétních implementacích.

[3]Princip Single Responsibility říká, že „modul by měl být odpovědný jednomu a pouze jednomu aktérovi - ten nebo co vyžaduje danou změnu“.

[4]Agregát je entita, která má nějaké další entity, které nemohou existovat bez tohoto určitého agregátu.

Podobně jako Hexagonal Architecture a Onion Architecture, tato architektura externalizuje implementační detaily toho, jak se připojí k vnějšímu světu (např. uživatelské rozhraní, databáze atd.) až k okrajům architektury.

4.2 Architektonické vzory

Tato sekce se zaměřuje na analýzu čtyř různých softwarových návrhových vzorů, které jsou často používány v moderním softwarovém vývoji: CQRS pattern[23i], Specification pattern[Cat20], Event Sourcing pattern[Ozk21] a Repository pattern[23ar]. Podobně jako u předchozí sekce architektur, zde se jedná o principy, které by se při vývoji měly dodržovat. Nicméně v rámci jedné architektury můžu uplatnit různou kombinaci těchto vzorů.

4.2.1 CQRS pattern

CQRS (Command Query Responsibility Segregation) pattern je návrhový vzor, který se zaměřuje na oddělení zápisových a dotazovacích operací v aplikaci. Tento vzor říká, že každý dotaz na data by měl být zpracován samostatně, a to bez ohledu na to, zda se jedná o dotaz na čtení nebo zápis dat.

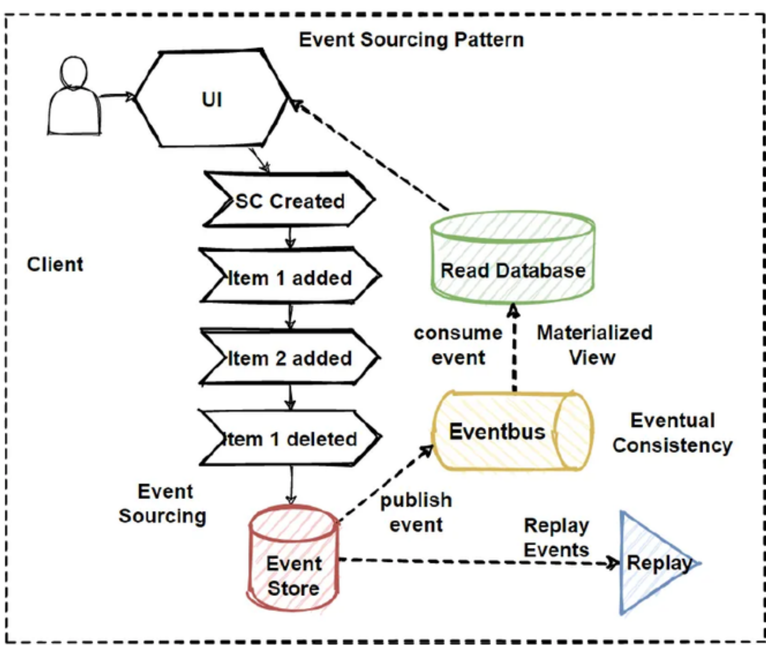
Výhodou CQRS patternu je, že umožňuje snadnou optimalizaci a škálování aplikace, protože oddělení dotazovací a zápisové části aplikace umožňuje optimalizovat a škálovat každou část samostatně. Další výhodou tohoto vzoru je, že umožňuje snadnou implementaci event-driven architektury a asynchronního zpracování dat.

Je potřeba mít na paměti, že oddělení dotazovací a zápisové části může vést k větší náročnosti na synchronizaci dat mezi těmito částmi.

4.2.2 Event Sourcing pattern

Event Sourcing pattern je návrhový vzor, který se zaměřuje na ukládání stavu aplikace jako sérii událostí (eventů), které popisují všechny změny stavu v aplikaci. Data musí být podle toho tak i ukládána do databáze. Zpětně se pak načítají události, které aplikace rekonstruuje do chtěného stavu celého aplikace (viz obr. 4.7). Namísto ukládání aktuálního stavu aplikace by měly být ukládány pouze události, které popisují všechny změny stavu. Princip technologie Git[5] jsou stejné a dá se to snadněji představit v této závislosti.

[5] Git je distribuovaný systém správy verzí, který sleduje změny v jakékoli sadě počítačových souborů, obvykle se používá pro koordinaci práce mezi programátory, kteří spolupracují na vývoji zdrojového kódu během vývoje softwaru.



Obrázek 4.7: Event Sourcing pattern vizualizace [Ozk21].

Hlavní výhodou tohoto patternu je, že umožňuje snadné načtení stavu aplikace z minulosti. Toto chování může být užitečné například v grafických nebo jiných editorech, kdy je možné se vrátit zpět v historii a aplikovat změny, resp. vrátit změny, které byly provedeny. Je zároveň velmi složitý pro vlastní implementaci a vyžaduje vysokou úroveň abstrakce. Zároveň, při použití tohoto vzoru je potřeba mít na paměti, že ukládání všech událostí může vést k větší náročnosti na výkon aplikace.

4.2.3 Specification pattern

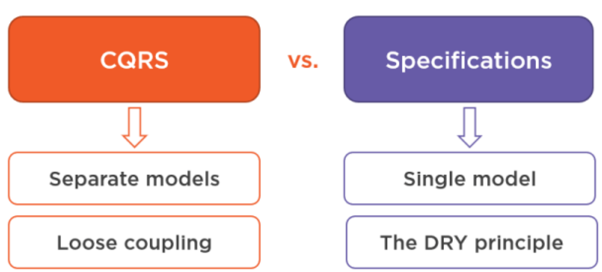
Specification pattern je návrhový vzor, který se zaměřuje na zjednodušení komplexních dotazů na data v aplikaci. Tento vzor říká, že každý dotaz by měl být vyjádřen jako sada specifikací, které definují požadavky na data. Tuto sadu specifikací si můžeme v objektově orientovaném programování a konkrétně v C# představit jako sadu tříd, které se mohou řetězit pomocí extension metod. Vytvoří tím programovou datovou strukturu, která definuje všechna data potřebná k znalosti sestavení dotazu.

Výhodou tohoto vzoru je, že umožňuje snadné testování aplikace, protože specifikace jsou samostatné a mohou být snadno testovány. Zároveň tak tyto specifikace je možné definovat na úrovni doménové vrstvy, kde definujeme různá doménová pravidla a není nutné tyto dotazy definovat na úrovni databáze.

S mnoha výhodami jsou zde i možná rizika kde Specification pattern může být

složitý na implementaci a vyžaduje vysokou úroveň abstrakce. Kromě toho, při použití tohoto vzoru je potřeba mít na paměti, že sestavování specifikací může vést k větší náročnosti na výkon aplikace.

Tento návrhový vzor se však rozchází s návrhovým vzorem CQRS (viz obr. 4.8). Zatímco Specification pattern obhajuje jednu doménu pro čtení i zápis a zachovat tím princip DRY (Don't Repeat Yourself), tak na druhé straně se CQRS pattern toto snaží rozdělit [Kho18]. Tyto rozdíly vychází o mnoho více najevo v kombinaci s Repository vzorem.



Obrázek 4.8: Rozdíl CQRS a Specification vzoru [Kho18].

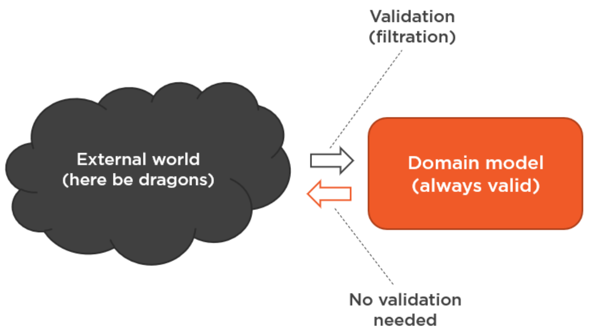
4.2.4 Repository pattern

Tento návrhový vzor se zaměřuje na oddělení logiky pro práci s daty od zbytku aplikace. Každá operace s daty by měla být prováděná pomocí repository objektu a poskytuje jednotné rozhraní pro práci s daty.

Toto řešení může vést ke zbytečné abstrakci, které zvyšuje složitost kódu, pokud není správně implementováno. Výhodou je však umožnění snadného oddělení logiky pro práci s daty, a to zlepšuje modularitu celého systému a závislostí v kódu.

4.2.5 Validace & Always-Valid Domain Model

Always-Valid Domain Model říká, že všechna data v doménovém modelu by měla být vždy platná a konzistentní s doménovými pravidly. Vyžaduje však pečlivou implementaci a dodržování doménových pravidel.



Obrázek 4.9: Grafické znázornění nutného bodu validace v Always-Valid Domain Model vzoru [Kho21].

4.3 Komunikační způsoby zvolených technologií

.NET umožňuje vytvoření soketové komunikace pomocí jeho interních knihoven. Avšak, nabízí také vysokoúrovňové rozhraní, které je velmi populární - SignalR.

4.3.1 SignalR

SignalR je open-source knihovna pro vývoj webových aplikací, která umožňuje vytvářet interaktivní aplikace s reálným časem pomocí technologie WebSockets. SignalR poskytuje programátorům jednoduché rozhraní pro vytváření serverových push notifikací a umožňuje klientům okamžitě reagovat na změny na serveru.

Tato technologie zároveň umožňuje snadnou škálovatelnost pomocí Redis backplates[23au] nebo Azure služeb. To je velkou výhodou zvláště, pokud by systém měl být nasazený globálně po světě a udržován v cloudových službách [23e].

4.4 Výběr zvolených řešení

Pro výběr architektury je nutné zmínit, že potřebujeme, aby systém byl snadno udržitelný a s tím se pojí i snadná modularita (výměna technologií), případně škálovatelnost. Postupem času se může projevit jiná technologie jako vhodnější. Toto se může stát i během budoucí produkce, kdy bude nutné vyměnit databázový systém za jiný, například z důvodu výkonnosti, nového efektivnější systému na trhu apod. Toto není výhrada jen konkrétně mého systému, ale jedná se o obecně vhodné vlastnosti pro každý velký systém.

Clean Architecture (CA) organizuje kód do různých vrstev, kde každá vrstva má svoji zodpovědnost. Hexagonal Architecture (HA) organizuje kód okolo jádra funkcionalit aplikace. Podle výše uvedených informací a mých zkušeností chci pro tento systém zvolit CA z důvodu toho, že CA se snaží udržet doménovou vrstvu a aplikační vrstvu separátně od zbytku implementace. Na druhé straně, HA se snaží použít také Separation of Concerns, ale nijak konkrétně nespecifikuje, jak tento proces má být proveden, do jakých vrstev. CA je v tomto daleko více specifičtější a díky návrhu, který uvádí, je v kombinaci s DDD vhodnou kombinací [Ibe22]. Plánuji tedy použít DDD, kde oddělení domény od zbytku kódu aplikace je jen vedlejším efektem tohoto hlavního cíle, ale je to hlavní součástí CA, se kterou se vhodně tyto architektury budou doplňovat.

Konkrétní výběr jednotlivých návrhových vzorů nelze jednoznačně vybrat na začátku, ale postupem vývoje. V další kapitole 5 budu tyto návrhové vzory používat a bude zde podrobněji popsáno, jak je aplikovat a k jakému účelu.

Návrh řešení a popis implementace

5

Tato kapitole popisuje návrhová řešení v rámci celého systému, implementaci konkrétních částí systému a komunikaci v mezi těmito částmi. V této kapitole také zmíním různé problémy, se kterými jsem se během vývoje setkal a bylo je nutné vyřešit. Velmi důležitou částí, kromě samotné implementace architektury, je také způsob implementace trasování a zobrazení trasovacích dat v uživatelské aplikaci. Speciální sekcí je zde práce s technologií Mapbox, kde popisuji práci s trasováním dat mobilních zařízení a zároveň práci s Mapbox.

5.1 Architektura systému

Tato sekce popisuje architektonický návrh celého systému a jeho implementaci. Pro vyšší přehlednost, tento popis dále podrobně rozdělím do dvou menších sekcí. V první (5.2) z těchto sekcí se budu zabývat serverovou částí, ke které se uživatelské aplikace mohou připojovat. V druhé (5.3) popíši principy, které jsem uplatnil na straně klientské aplikace a způsoby komunikace se serverem.

Na začátku by bylo vhodné upozornit na důležité rozdíly mezi ***solution***[1] a ***project***[2]. Tyto termíny jsou v .NET zásadní a budu je velmi často používat.

Se začátkem vývoje se dalo nad celým systémem uvažovat jako nad párem jednoduchých aplikací, kde jedna by představovala server a druhá klientskou aplikaci. Postupným rozvojem systému a požadavků na systém jsem byl nucený celý systém začít členit do menších částí a zvýšit tím udržitelnost, ale zároveň tím vzrostla i složitost projektu.

[1] Solution v .NET se dá přeložit jako „řešení“. V jiném programovacím jazyce bychom to standardně nazvali projektem. Zde to má podobný význam, jedná se o ucelené řešení jednoho nebo více projektů (`project`).

[2] Projekt se dá v .NET představit jako „package“ (balík), který je možná běžnější v jiných programovacích jazycích. Může představovat executable aplikaci nebo třeba jen knihovnu tříd. Mezi projekty na sebe mohou vytvářet závislosti v rámci jednoho `solution`.

Jedno z prvních rozhodnutí o vyšší úrovni členění bylo nutné vyřešit ihned při startu implementace, a to byl způsob komunikace mezi klientskými aplikacemi a serverem (více v 5.4). Jednoduchý způsob by bylo udržet komunikaci pouze dotazování se na námi vytvořený API server. Ovšem, takové řešení by vyžadovalo, aby všechny klientské aplikace „připojené" na server fungovali a řídili se sami. Nebyl bych schopný vyřešit situaci (nebo velmi obtížně) v případě, kdy potřebuji upozornit jiné uživatele, na základě akce, která se stala vyvoláním jiného uživatele. Bylo tedy jasné, že budu muset zachovat komunikaci pomocí API, ale zároveň použít komunikaci pomocí soketů, které mi umožní udržovat neustále informaci o připojených klientech. Pomocí takového řešení jsem schopen již zmíněný příklad realizovat.

Obrázek 5.1: Grafický přehled všech řešení v rámci systému včetně projektů, ze kterých se skládají a jejich závislostí.

Z pohledu udržitelnosti celého systému jsem chtěl celý projekty udržet co nejvíce přehledný. Míchání implementace API serveru a soketové komunikace se nemusí zdát jako problém při malém či středním rozsahu celého projektu. V narůstající velikosti se však implementace může začít plést a promíchávat navzájem. Z pohledu optimalizace a dostupnosti služby poskytující uživateli by bylo vhodné tyto způsoby komunikace oddělit, jelikož každá z nich bude poskytovat jiný druh informací, pro který se klientské aplikace budou dotazovat.

Podobná rozhodnutí následovala a postupně se z celého systému stalo 7 samostatných řešení (solutions) s dohromady celkem 16 projekty v rámci celého systému. Pro jednodušší představu jsem vytvořil jednoduchou grafickou reprezentaci (viz obr. 5.1), kde lze vidět všechna řešení, z jakých projektů se skládají. Více informací bude uvedeno v následujících sekcích.

5.2 Serverová architektura

Architektura serveru je konstruována pomocí principů Domain-Driven Design (DDD) a Clean Architecture (CA). První náznaky těchto zvolených principů lze již vidět na obr. 5.1, kde lze vidět jmenovité rozdělení projektů do tří základních projektů: Api, Core a Infrastructure. K tomuto rozdělení se dostanu později v této sekci.

Hlavním rozdělením serverové části je rozdělení podle služby, kterou server nabízí. Totiž, jedním z prvních rozhodnutí byla nutnost navrhnout způsob komunikace. Je nutné zachovat oba již zmiňované způsoby, kterými jsou webové REST API a soketová komunikace. Pro tyto účely proto vznikly 2 samostatné aplikace, se kterými uživatelské aplikace mohou komunikovat. Projekty, představující executable aplikaci mají příponu `.App`. V tomto případě jde o dva projekty `Server.Api.App` (dále jen API server) a `Server.Engine.App` (dále jen Engine server), kde API server vytváří jen a pouze koncové body (end-points) pro možnou komunikaci a Engine server vytváře prostředí pro komunikaci pomocí soketů.

Mezi mnoho zmiňovanými požadavky byla snadná škálovatelnost a udržitelnost softwaru. Kdyby v budoucnu vyšel nový výkonnější databázový systém, který by významně zvýšil výkonnost a odezvu pro uživatele, chtěl bych takovou změnu propagovat. Aby to bylo možné, je nutné dodržovat zmíněné zásady DDD a CA.

Na obrázku 5.2 lze vidět strukturu jednoho z řešení, které v této sekci budu dále popisovat. V rámci tohoto obrázku jde vidět, že je řešení rozdělené do další tří projektů s příponou reprezentující (viz výše) jejich účel. Jedná se o základní rozdělení pomocí principů DDD. Účel a organizaci kódu popíši v tabulce 5.1.

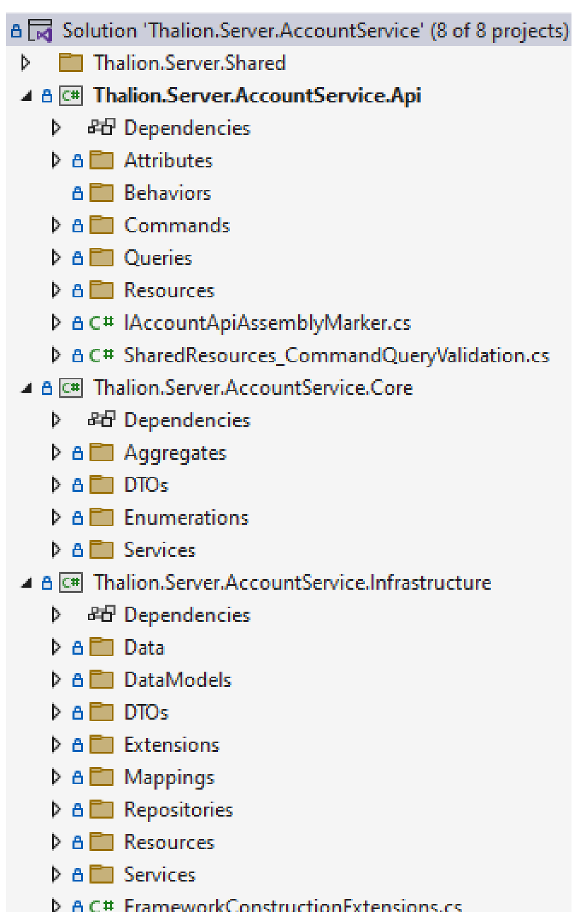
Tabulka 5.1: Obsah a význam projektů rozdělených dle DDD.

projekt	**vysvětlení**
`Core`	Z pohledu architektury tento projekt představuje doménovou vrstvu (***Entity***). Tuto vrstvu reprezentují doménové modely (agregáty) entit, Repository pattern rozhraní rozhraní jednotlivých agregátů a jiná rozhraní definující doménovou logiku a pravidla dané služby. Součástí tohoto projektu jsou: • `Aggregates`: obsahují implementace doménové logiky, její entity a rozhraní. • `DTOs`: obsahují implementace nezbytných objektů pro přenášení dat v dané službě. • `Services`: implementace rozhraní speciálních „funkčních“ služeb pro použití v rámci dané služby. Může se jednat například o často používané funkce. Snaží se zachovávat principy DRY.
`Api`	Architektonicky tento projekt představu aplikační vrstvu (také známou jako ***Use cases***). V rámci tohoto projektu je definované rozhraní pro komunikaci s „okolním světem“, v případě tohoto sytému se jedná o zbylé části aplikace. Pomocí tohoto rozhraní lze ovládat funkce nabízené a definované touto službou (tj. `AccountService`). Hlavní součástí tohoto projektu jsou: • `Commands & Queries`: rozhraní pro komunikaci/ovládání služby podle návrhového vzoru CQRS. • `Resources`: definice zajišťující globalizaci a lokalizaci této části dané služby.

(tabulka pokračuje na další stránce)

Tabulka 5.1 *(pokračování z předchozí stránky)*

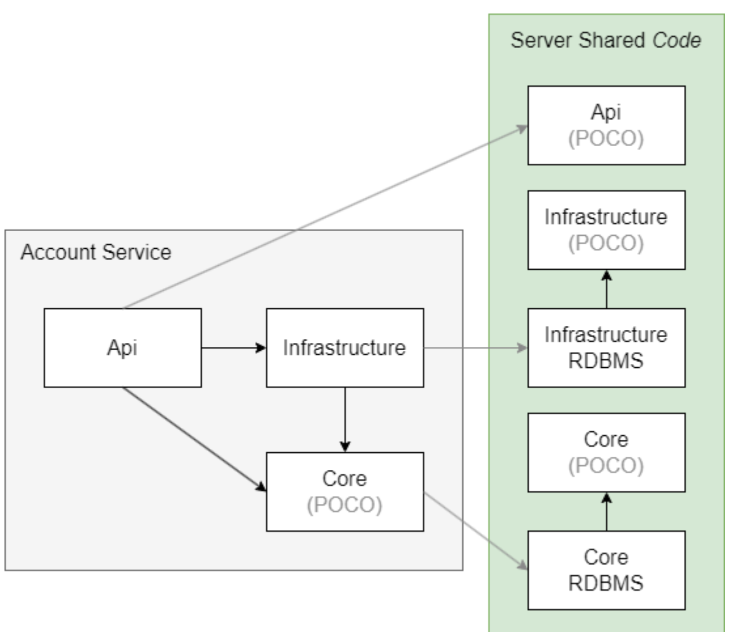
projekt	**vysvětlení**
`Infrastructure`	Zde se nachází všechny implementace aplikace. V architektuře by se jednalo o vrstvu rozhraní adaptérů, jinými slovy ***Infrastructure***. Tato vrstva obsahuje implementace databázových kontextů, migrace dat, datových modelů databáze, Repository implementace a již zmíněné „funkční“ služby.



Obrázek 5.2: Struktura řešení `Server.AccountService`.

Z tabulky 5.1 vyplynulo mnoho nových informací jako prezence návrhového vzoru CQRS apod. Než se k popisu těchto použitých vzorů dostanu, je nutné vysvětlit důležitost návrhu DDD/CA (Domain Driven Design + Clean Architecture) a jakým způsobem navrhnout členění kódu a funkcí v systému serverů.

Již jsem zmínil, že z návrhu je nutná existence dvou serverů, kde každý má svoji funkci. Jeden pro soketovou komunikaci s uživatelskými aplikacemi a druhý jim poskytující API. Oba tyto servery budou s vysokou pravděpodobností muset přistupovat ke stejným datům, do stejné databáze. Oba dva budou pracovat s uživateli. To je hlavní a první funkcionalita, která musí být součástí obou dvou serverů. Z pohledu udržitelnosti a snadné správy kódu rozhodně nechci, aby docházelo k porušování zásad ***DRY***[3]. Z tohoto důvodu je nutné navrhnout v systému serverů pořádek, který dokáže sdružovat společné funkcionality do jednoho nebo více spolu pracujících projektů. Tyto projekty lze následně použít jako balík (package), který lze importovat a následně používat ve vyvíjených aplikacích (obou zmíněných serverech). Rozhodl jsem se tedy vytvořit samostatně fungující řešení, které lze připojit k aplikacím dle potřeby. Každé takové řešení nazývám `Service` a již jedna z implementovaných zde byla zmíněna jako příklad - `AccountService` (viz obr. 5.3).



Obrázek 5.3: Projektový návrh řešení `Server.AccountService` včetně závislostí mezi projekty a závislostmi k řešení `Server.Shared`.

Abych striktně zachoval principy DDD/CA, tak každá definovaná služba (`Service`, např. uváděná `AccountService`) je vytvořena jako samostatné řešení nezávislé na aplikaci. Na obrázku 5.3 lze vidět závislostní diagram projektů v rámci jedné služby. Projektová struktura je fixní pro každou službu a v tomto příkladu pracuji se službou `AccountService`. Řešení je složeno z projektů zmíněných v tabulce 5.1.

[3]Don't Repeat Yourself je princip vývoje softwaru, jehož cílem je omezit opakování softwarových vzorů, nahradit je abstrakcemi nebo použít normalizaci dat, aby se zabránilo nadbytečnosti.

Zároveň si je možné všimnout, že služba je závislá na projektech z jiného řešení - `Server.Shared`. Tyto projekty tvoří samostatné řešení `Server.Shared`, které je součástí každé implementované služby a přináší ji knihovnu základních tříd pro každý z projektů rozdělených dle DDD/CA. Jinými slovy se jedná o sdílenou funkcionalitu (kód) používanou ve všech službách. Díky tomuto zachovávám DRY.

Dále je z obrázku zjevné, že zde jsou různé varianty. Například je zde projekt `Server.Shared.Core` a `Server.Shared.Core.RDBMS`, který je na něm závislý. První zmíněný projekt vytváří obecnou knihovnu funkcionalit, druhý zmíněný s příponou „RDBMS" rozšiřuje tento balík o rozhraní pro práci s relačními databázovými systémy, v případě .NET je tato implementace provedena pomocí Entity Frameworku. Díky tomuto řešení je možné vytvořit rozhraní pro práci s různými datovými uložišťmi. Na obrázku si lze povšimnou klíčového slova „POCO", které znamená, že by daný projekt neměl být závislý na jiném projektu ani žádné externí knihovně [23al]. Jedinou výjimkou jsou samotné `Core` projekty v rámci každé služby, kde je nezbytná jedna závislost a to na `Server.Shared.Core` (nebo jeho rozšiřující verzi `RDBMS`, případně v budoucnu i jinou) projektu nesoucí základní funkcionalitu.

5.2.1 Monolit vs. Microservices

Služby tedy vytvářejí jakousi knihovnu funkcí, kterou nabízejí aplikaci, která je používá. Nicméně se může zdát, že tento způsob organizace projektů připomíná architekturu Microservices. Toto návrhové rozhodnutí je použito záměrně.

Mým cílem je vytvořit monolitickou aplikaci z důvodu možné složitosti implementace a návrhové složitosti z pohledu komunikace mezi jednotlivými službami, pokud bych vybral cestu Microservices. Nejde pouze o složitost, ale možné zvýšení latence serveru které jde stěží odhadnout, když je práce teprve ve vývoji. I když jsou servery plánovány jako monolit, tak jsem si do budoucna nechtěl zamezit případnou změnu a optimalizaci. Proto přes důmyslné abstraktní konstrukce jsem vytvořil strukturu, kterou jsem rozdělil jednotlivé související funkcionality do jednotlivých (již zmíněných) služeb a samotná aplikace tyto služby používá.

Jediným rozdílem je, že místo síťové komunikace, komunikace probíhá přímo v implementaci konkrétní aplikace. V případě zmíněných služeb, by komunikační rozhraní představoval projekt `Api` z obrázku 5.3. Z takového návrhu je možné, kdykoli změnit architekturu monolitu na Microservices s minimálními nutnými změnami v kódu. Api a Engine server tyto služby mohou jednoduše přidat pomocí rozhraní, které jim každá z těchto služeb nabízí. V kódu 5.1 lze vidět metodu, která slouží jako

extension metod pro `IServiceCollection`, pomocí které mohu řetězově zavolat již konkrétní metody jednotlivých služeb (`AddServiceInfrastructure`).

Pokud se podíváme konkrétně do jednotlivých služeb, tak právě zmíněná metoda `AddServiceInfrastructure` je vstupním bodem, která integruje nezbytně nutné závislosti do celého systému aplikace (viz 5.2). Tuto metodu nabízí rozhraní každé služby a v rámci této metody mohu definovat všechny nutné závislosti pro správnou funkčnost jednotlivé služby. V aplikaci následně nemusím řešit žádné závislosti, jen je potřeba připojit tuto metodu pomocí extension řetězení na builder aplikace.

Zdrojový kód 5.1: Výpis kódu metody pro integraci služeb do aplikace.

```
namespace Server.Engine.App.DI;

public static IServiceCollection AddProjectServices(
  this IServiceCollection services,
  string dbConnectionString, bool dbInMemory = false)
{
  // Add shared services used among the project services
  services.AddSharedEventStore(dbConnectionString, dbInMemory);
  services.AddSharedLocalization();
  services.AddSharedRepositories();

  // --- bind services here ---

  AccountService.Infrastructure.FrameworkConstructionExtensions
    .AddServiceInfrastructure(services, dbConnectionString, dbInMemory);
  WaypointService.Infrastructure.FrameworkConstructionExtensions
    .AddServiceInfrastructure(services, dbConnectionString, dbInMemory);

  // Return collection for chaining
  return services;
}
```

Zdrojový kód 5.2: Kód metody konkrétní služby pro integraci do aplikace.

```
namespace Server.AccountService.Infrastructure;

public static IServiceCollection AddServiceInfrastructure(
  this IServiceCollection services,
  string dbConnectionString, bool dbInMemory = false)
{
  if (!dbInMemory && dbConnectionString == null)
    throw new ArgumentException("Invalid database connection parameters!");

  // Add database context
  services.AddDatabasePersistence(dbConnectionString, dbInMemory);
  services.AddRepositories();
  // We are using Identity database context in this service,
  // lets add Identity then
  services.AddIdentity();

  // Return collection for chaining
  return services;
}
```

Rozhodnutí realizovat monolitickou aplikaci je jednodušší pro implementaci. Ovšem, přináší s sebou jednu možnou nevýhodu, která může být v jisté podobně i výhodou. V architektuře Microservices je každá služba samostatná aplikace. Každá služba si však musí držet vlastní databázový kontext, vlastní datové modely a doménovou logiku nad nimi. V tomto bodě přichází hlavní nevýhoda, která tímto rozbíjí efektivní využití relačních databází. Implementace systému zahrnuje dvě hlavní služby, jednu již zmíněnou `AccountService`, která nabízí funkcionalitu ohledně správy uživatelů a následně druhou službu `WaypointService`, která nabízí funkce pro práci s mapou. Může zde nastat příklad relace mezi bodem na mapě nebo přímo pozicí uživatele, která je uložena v databázi pod správou služby starající se o mapy a potom samotný uživatel uložený v databázi pod správou účtů. Jelikož se jedná o různé databázové kontexty, tak nelze spoléhat na relace v databázovém modelu.

Toto není lehké návrhové rozhodnutí a je nutné upřednostnit požadavky systému, kterými jsou udržitelnost a škálovatelnost. Tyto požadavky převažují a po uvážení zachování principů DRY a dlouhodobého vývoje, kdy bude funkcionalit jen přibývat, tak se nezdálo rozumné se snažit aplikaci tvořit jako jeden plnohodnotný celek, ale spíše funkcionality třídit.

5.2.2 Organizace a tok dat

S ohledem na to, jak je vytvořená infrastruktura jednotlivých služeb a jakým způsobem jsou aplikace těmito službami vytvářeny, potom musím vymyslet sofistikovaný a robustní způsob, jak předávat data mezi jednotlivými vrstvami systému.

Základním prvkem jsou tzv. data transfer objekty, které z principu návrhu jsou ***immutable***[4]. Pomocí těchto objektů lze snadno přenášet data mezi zmíněnými vrstvami. Pomocí sofistikovaných rozhraní lze vynutit pouze užití takovýchto modelů, které označím příslušným rozhraním.

Další důležitou vlastností pro zachování udržitelného kódu je zachovat co nejnižší úroveň ***Loose Coupling***[5] a již také zmíněný princip Dependency Inversion. Kdybychom si představili jednoduchou aplikaci s `Controller` třídou zpracovávající požadavky z uživatelského rozhraní, tak zde budeme potřebovat závislost na každou službu, která je k zapotřebí vykonání takových požadavků. Pokud by se jednalo o získání informací o uživateli z databáze, potřebovali bychom databázový kontext,

[4] Immutable objekt je takový objekt, kterému nelze změnit jeho stav po tom, co je vytvořený.

[5] Systém, který se označuje Loose Coupled je takový, kde jeho části/komponenty jsou na sobě velmi málo závislé nebo vůbec. Respektive, komponenty se navzájem co nejméně ovlivňují.

atp. Zde přichází na scénu ***Mediator pattern***[6]. Zjednodušeně řečeno se jedná o obalový systém okolo všech závislostí. Při startu aplikace všechny chtěné závislosti v aplikaci přidáme do rozhraní Mediátoru. Následně v jednotlivých místech, kde by závislosti standardně vznikaly, nechám pomocí ***Dependency Injection***[7] parametrizovat pouze definovanou třídu sběrnice služeb (proměnná `_serviceBus` - viz kód 5.3).

Zdrojový kód 5.3: Metoda obsluhjící požadavek z klientské strany pro aktualizaci uživatelských dat.

```
namespace Server.Engine.App.Hubs;

public async Task RequestUserData()
{
  var response = new SocketResponse<SendUserDataDto>();

  // Check if context data are valid ...
  if (!Guid.TryParse(Context.UserIdentifier, out var uid))
  {
    response.MakeErrorResponse(null, null, null);
    await Clients.Caller.SendUserData(response);
    return;
  }

  //
  // Required context data are valid here
  //

  var reqRes = await _serviceBus.Send(new GetUserQuery(uid));
  if (reqRes.Successful)
  {
    response.MakeSuccessfulResponse(
      reqRes.Message,
      reqRes.Errors.ToStringsOrNull(),
      new SendUserDataDto(
        reqRes.Data!.Nickname
      ));
  }
  else
  {
    response.MakeErrorResponse(
      reqRes.Message,
      reqRes.Errors.ToStringsOrNull(),
      null);
  }

  await Clients.Caller.SendUserData(response);
}
```

Metoda ukázaná v 5.3 slouží jako obslužná rutina pro požadavek ze strany uživatelské aplikace, při dotázání o aktualizaci uživatelských dat. Metoda ověří, zda se jedná o uživatele a pomocí sběrnice služeb pošle registrovaný požadavek `GetUserQuery`

[6] Návrhový vzor Mediator zapouzdřuje, způsob interakce sady objektů. Mediátor podporuje volné připojení tím, že brání objektům v tom, aby na sebe explicitně odkazovaly.

[7] DI je programovací technika, díky které je třída nezávislá na jejích závislostech.

do adekvátní služby ke zpracování. Na základě vrácených hodnot a obalové třídy požadavku ServiceResult<T>, která nese získaná data a informace o stavu požadavku, lze sestavit finální odpověď, kterou lze rozeslat soketovou komunikací zpět uživatelské aplikaci.

Zdrojový kód 5.4: Ukázka struktury rozhraní v Mediator patternu

```
namespace Server.AccountService.Api.Queries.UserAggregate;

public record GetUserQuery(Guid UserId)
  : IRequest<ServiceResult<UserDto>>;

internal class GetUserQueryHandler : QueryHandlerBase<GetUserQuery, UserDto>
{
  ...
}
```

Pokračuji-li ve stejném příkladu, tak potom když se aplikace dostane k samotnému zpracování požadavku volaného pomocí rozhraní mediátoru, tak tento požadavek je reprezentovaný jen a pouze jednoduchou immutable třídou nesoucí požadovaná data nutná pro zpracování tohoto požadavku (viz 5.4).

Ke každé takové třídě však připadá ještě `handler`, který slouží jako obsluha požadavku. Tento handler představuje třída `GetUserQueryHandler`, která dědí z abstraktní třídy `QueryHandlerBase` vytvářející interní rozhraní Mediátoru. Zároveň této abstrakci je nutné předat 2 generické parametry, kde prvním je reprezentující třída typu požadavku (zmíněná výše) a druhým parametrem je návratové DTO, který ponese získaná data.

Následně v ukázce 5.5 lze vidět část obsahu třídy `GetUserQueryHandler` včetně samotné metody `Handle` sloužící jako obsluha vyvolaného požadavku. Zde je vidět vyhodnocení procesu získání požadovaných dat službou pro získání dat uživatele `IApplicationUserService`.

Tento popis zahrnuje základní informace o organizaci toku dat na serveru. Konkrétní informace přenosu dat a samotných data flow diagramů jsou popsány v sekci 5.4, kde zmíním i způsob komunikace nejen na serveru, ale i mezi serverem a klientskými aplikacemi, včetně reálného užití obou dvou typů server: Api a Engine.

Zdrojový kód 5.5: Obslužná metoda Handle nacházející se ve třídě z předchozího příkladu - GetUserQueryHandler.

```
private readonly IApplicationUserService _applicationUserService;
private readonly IMapper _mapper;
private readonly IUserRepository _repository;

...

public override async Task<ServiceResult<UserDto>> Handle(
  GetUserQuery query, CancellationToken cancellationToken)
{
  ServiceResult<UserDto> res = new();

  var userResult = await _applicationUserService
    .GetUserAsync(query.UserId);
  if (userResult.Failed)
    res.Failed()
      .WithMessage(userResult.Message)
      .WithErrors(userResult.Errors);
  else
    res.Successful()
      .WithData(_mapper.Map<UserDto>(userResult.Data));

  return res;
}
```

5.2.3 Získávání a ukládání dat

Přístup k datům a uložišti závisí na použité implementaci `Core` a `Infrastructure` vrstvy. Na předchozím obrázku 5.3 bylo možné vidět řešení `Server.Shared`, které tyto projekty nabízí s možným rozšířením, jako je například ***RDBMS***. Díky navržené struktuře těchto projektů lze řešení libovolně rozšiřovat a vytvářet knihovny pro různé typy databází.

V daném případě si však vystačím s relačními databázemi, konkrétně PostgreSQL. Tento databázový systém nabízí i možnost rozšíření PostGIS, který s sebou přináší funkce pro ukládání a vyhledávání v geolokačních datech.

Nyní je velmi důležité navrhnout takový sofistikovaný způsob, aby celá tato architektura plnila svůj zamýšlený účel, čili schopnost snadno vyměnit databázový systém za jiný, bez nutnosti zasahovat do kódu doménové logiky.

V následujících podsekcích se pokusím představit několik řešení a pomocí nich dospět k použitelnému řešení implementace Repository patternu.

Naivní přístup

Naivní přístup by mohl být snadnou implementací Repository patternu, kde repository jednotlivých agregátů bude nabízet základní funkcionalitu (CRUD[8]) a případné speciální funkce lze implementovat později v samotném repository dle potřeby (například požadavek na čtení dat z databáze se specifickým filtrem, který nelze zobecnit). Toto řešení je efektivní a elegantní pro aplikace malého rozsahu. Mnou implementovaný systém je však implementovaný pomocí architektury DDD/CA a z toho principu je již rozdělen do několika projektů: Api, Core a Infrastructure.

Toto naivní řešení s sebou přináší dva zásadní problémy:

- Doménová logika musí být částečně implementována v Infrastructure vrstvě.
- Zvyšováním velikosti projekt také stoupá složitost udržování kódu.

Podle architektonické návrhu rozhraní používaných repository definuje v projektu `Core` a samotnou implementaci ponechám projektu `Infrastructure`. Ovšem, pokud chci tuto architekturu správně dodržet, tak jsem nucen implementaci jednotlivých metod repository zanechat právě v `Infrastructure` vrstvě. To znamená, že veškerá doménová logika je definovaná ve vrstvě `Use cases` a sestavování dotazů na data je implementováno samostatně ve vrstvě, kde by měla být pouze implementace spjatá se samotnými technologiemi, v tomto případě databázový kontext. Tím by ale vznikla závislost, která se rozporuje s návrhem celého systému.

Druhým problémem je následně narůstající složitost a možná nepřehlednost implementace v samotných repository. Se zvyšující se velikostí systému budou pravděpodobně narůstat požadavky na specifické metody v repository. Nejen, že to může zvyšovat složitost přehlednosti, ale dokonce začít porušovat principy DRY.

Lepší řešení pomocí Specification patternu

Výše zmíněné nedostatky naivního řešení lze snadno vyřešit aplikováním tzv. Specification patternu. Jedná se o složitou strukturu pro implementaci a vyžaduje vysokou úroveň abstrakce, nicméně díky tomuto dokáži definovat sestavování dotazů (specifikace) ve vrstvě doménové logiky. Použitím se musí změnit vstupní parametry repository metod, kde vstupem bude objekt specifikace, nikoli výrazy nebo přímo data. V rámci `Infrastructure` vrstvy mám veškeré informace o tom, abych dokázal dotaz do databáze sestavit, přesně jak si to doménová vrstva přeje.

[8] Create Read Update Delete, jedná se o základní nabízené operace.

Nyní když máme oba zmíněné problémy naivního řešení opravené, tak se dostáváme do bodu, kdy začínáme řešit jiné problémy. Dojdeme-li k představě, že bych byl doopravdy nucen změnit databázový systém na jiný, tak rozhodně nechci být nucený zasahovat do jakékoli doménové logiky. Příkladem, přidání nebo odebrání uživatele ze skupiny uživatelů je záležitost doménové logiky a implementace tohoto procesu by měla být správně vytvořena v projektu `Core` představující vrstvu `Use cases` a tím pádem kompletně nezávislou na implementaci konkrétní technologie.

V relační databázi máme však každou tabulku reprezentovanou nějakým datovým modelem, třídou v kódu. Z vlastní zkušenosti není zcela jednoduché najít správnou odpověď na rozdíl mezi datovým a doménovým modelem a mnoho webových zdrojů tyto modely směšuje. Ovšem, každý model plní jinou funkci, i když jsou v mnoha případech považovány za jeden a ten samý model.

Doménový model představuje objekt velmi podobně jako to dělá objektově orientované programování, tj. každý objekt má své atributy a své vlastnosti ve formě metod. Doménový model na tom není jinak a nabízí veškerou funkcionalitu. V případě uživatele si lze představit doménový model `User`, který má vlastnosti jako `AddToGroup` a `RemoveFromGroup`. Zde nastává rozpor s datovým modelem, který by správně měl obsahovat jen a pouze atributy, které zcela kopírují strukturu databázové tabulky.

Pokud začneme nad tímto problémem uvažovat, tak velmi rychle lze docílit k názoru, že se o zásadní problém nejedná a struktura doménového a datového modelu by se téměř nelišila, a proto je možné je považovat jako jeden. Ovšem toto je zásadní nepravda. Jednoduchým příkladem může být atribut hodnoty data a času. S velkou pravděpodobností tento atribut budeme ukládat pomocí jiného typu než typ, se kterým budeme pracovat v implementaci doménové logiky. Pokud bychom ho naivně ukládali jako textový řetězec a následně s ním chtěli pracovat, tak to práci výrazně znepříjemní. Jednoduchým řešením by bylo vytvoření nového atributu, který by fungoval pro doménovou logiku. V tento moment však začneme do kódu tohoto modelu zanášet značně rychle narůstající nepřehlednost a velmi náročnou udržitelnost. Samotný problém nepřehlednosti nemusí být tak fatální, ale výrazně to bude zvyšovat chybovost programátora, která může nastat při manipulaci s daty.

Druhým problémem s tímto spojeným je výše zmíněný příklad s nutností změnit databázový systém. Pokud by k tomu došlo, tak je programátor nucen zasáhnout a provést změny přímo v samotném datovém/doménovém modelu a tím provádí změny v doménové logice, která se změnou databázového systému vůbec nesouvisí.

Shrnu-li všechny tyto informace, tak lze docílit toho, že Specification pattern není

zase tak vhodný pro aplikaci. Je pravda, že se rozchází v principech s CQRS patternem, ale i přes výše uvedené závažné problémy se jedná o velice sofistikovaný vzor, pokud je správně implementovaný. Vše se odvíjí od požadavků na systém. Z vlastní zkušenosti se setkávám s převažujícím množstvím řešení, kdy je datový a doménový model slučován do jednoho, jak výše popisuje. A pokud si lze tyto problém připustit, tak je Specification pattern více než užitečný. Ovšem, taková řešení je vhodné uplatňovat pro projekty střední velikosti. Pro projekty malé velikosti se může jednat o zbytečně zatěžující způsob implementace a pro projekty velkého rozsahu je to na uvážení a požadavcích.

Řešení s rozdělením doménového a datového modelu

Toto řešení jsem shledal relativně obtížným. Je velmi obtížné takové řešení vyhledat v online zdrojích pro možnou inspiraci nebo napodobení. Z důvodu obtížnosti implementace, nebývá často realizováno.

Je nutné na začátek zmínit zásadní problémy tohoto řešení. Používáme-li Entity Framework nebo framework tomu podobný je dobře si uvědomit, jak přesně fungují „Tracking Queries"[23av] a jestli tuto funkcionalitu zcela potřebujeme. Z důvodu rozdělení doménového a datového modelu, o tuto funkci frameworku přijdeme z důvodu nutnosti přemapování datových modelů na doménové a naopak. Framework potom nedokáže udržet informace o používaných objektech.

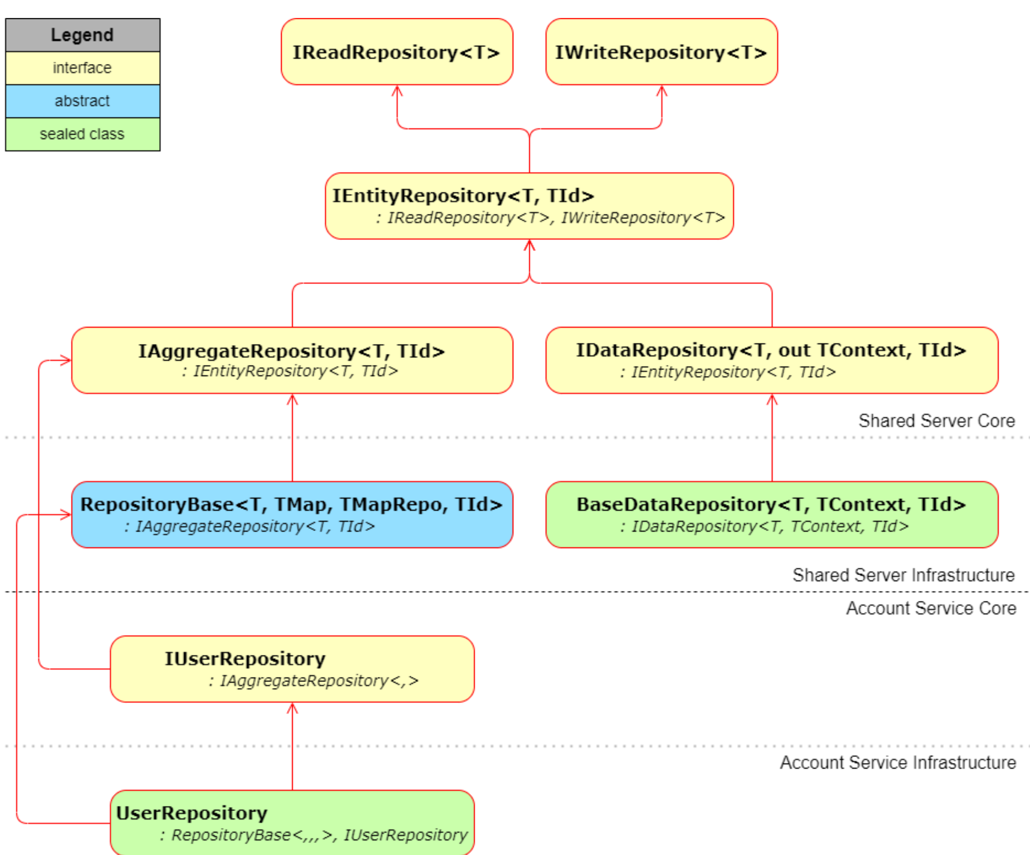
Další zásadní problém nemusí být zcela zjevný, ale pokud toto řešení chci uplatnit, tak jsem nucen zcela ustoupit od řešení pomocí Specification patternu. Je to z toho důvodu, že doménové modely nemají informace o navigačních atributech, které slouží například pro udržování informací o vazbách z jiných tabulek. S touto informací se však dostávám k problémům, které s sebou přinášelo naivní řešení.

Při vývoji jsem v tomto bodě začal postupně chápat, proč mnoho vývojářů jde směrem sloučeného doménového a datového modelu i přes značná rizika. Nicméně takový směr byl zásadně v rozporu s návrhem systému, kterého bych chtěl dosáhnout. Dlouho jsem se snažil tento problém vyřešit a jedno ze zdánlivých řešení bylo pomocí knihovny AutoMapper[23f] pomocí rozšiřujících metod `Project`. Jedná se o projekci přemapovaných objektů přímo do interního sestavování dotazů, jako by tomu bylo standardně s datovými modely. Toto mělo dva nedostatky, kde jedním je již zmíněný problém nemožnost použití příkazů `SQL JOIN` (a jeho dalších variant) a druhým problém vytvářel nutnost vždy udržovat stejné atributy v doménovém modelu, jako tomu je v datovém modelu.

Kombinovaný vlastní přístup k řešení

Dlouho dobu jsem se snažil tyto problémy vyřešit, ale nakonec jsem skončil u úsudku, že je nutné si vybrat z méně horší varianty. Jelikož jsem nebyl s tímto faktem spokojený, tak jsem přišel s vlastním řešením, které do implementace zavádí několik pravidel. Tato pravidla lze však snadno řídit pomocí sofistikované úrovně abstrakce pomocí generičnosti kódu.

První úkol bylo vytvoření repository struktury, která by se dala použít napříč projektovým rozdělením pomocí DDD/CA a zároveň fungovala pro systém s rozděleným doménovým a datovým modelem. Na obrázku 5.4 lze vidět celý návrh repository struktury. Z obrázku lze vidět dědičnost jednotlivých repository úrovní, kde nejnižší vrstvu tvoří definice základních metod pro zápis a čtení, tj. IWriteRepository<T> a IReadRepository<T>, kde T je generický typ pro identifikátor v daném repository.



Obrázek 5.4: Návrh řešení kompozice repository s příkladem na User agregátu.

Na obrázku 5.4 lze také vidět abstraktní třídu `RepositoryBase`, kde její použití nemusí být zcela jednoznačné. Jedná se o repository ke kterému přistupují doménové modely, nikoli datové. Tuto informaci lze snadno vyčíst z názvů rozhraní, od kterých tyto repository dědí. Generické parametry této třídy tvoří: typ doménového modelu, typ datového modelu mapovaný k danému doménovému modelu, rozhraní pracující s danými typy dat a datový typ identifikátoru. Pro snadnější ilustraci je zde vidět náhled implementace 5.6.

Zdrojový kód 5.6: Náhled implementace základu pro doménové repository.

```
1  namespace Server.Shared.Infrastructure.RDBMS.Relational.Domain;
2
3  public abstract class RepositoryBase<T, TMap, TMapRepo, TId>
4    : IAggregateRepository<T, TId>
5    where T : class, IAggregateRoot<TId>
6    where TMap : class, IDataEntity<TId>
7    where TMapRepo : IDataRepository<TMap, IUnitOfWork, TId>
8    where TId : struct
9  {
10   ...
11
12   public virtual async ValueTask AddAsync(T entity)
13   {
14     if (entity == null)
15       throw new ArgumentNullException(
16         $"Value of '{nameof(entity)}' cannot be null!");
17
18     var dataEntity = Mapper.Map<TMap>(entity);
19     await PersistenceRepository.AddAsync(dataEntity);
20   }
21
22   ...
23 }
```

Dále je možné si všimnou, že na obrázku 5.4 zcela chybí implementace datových repository ze strany služeb. Takové řešení není ani nutné, jelikož veškeré sestavování probíhá v úrovni doménového repository, kde lze kombinovat CRUD metody dostupné z `BaseDataRepository`. Pro samotnou práci s daty nám CRUD operace stačí a je pouze na úrovni doménových repository, jak tento dotaz sestaví. Je však třeba si uvědomit fakt, že v doménové úrovni dochází k přemapování jednotlivých modelů, a proto zde pracuji s doménovými i datovými modely (viz 5.6). Výstupem z doménových repository může být však pouze doménový model. Náhled implementace 5.7 ukazuje zmíněnou práci pouze s CRUD operacemi nad databázovým setem konkrétních dat, pro která daný repository je vytvořený. Z toho principu je zcela zbytečné rozdělovat logiku správy mapování a přístupu do databáze. Pro práci s daty nám tedy stačí jedna pevně daná třída a není nutné žádných specialit.

Zdrojový kód 5.7: Náhled implementace datového repository.

```
namespace Server.Shared.Infrastructure.RDBMS.Relational.Data;

internal sealed class BaseDataRepository<T, TContext, TId>
  : IDataRepository<T, TContext, TId>
  where T : class, IDataEntity<TId>
  where TContext : DbContext, IUnitOfWork
  where TId : struct
{
  private readonly DbSet<T> _dbEntities;

  ...

  public async ValueTask AddAsync(T entity)
  {
    if (entity == null)
      throw new ArgumentNullException(
        $"Value of '{nameof(entity)}' cannot be null!");

    await _dbEntities.AddAsync(entity);
  }

  ...
}
```

Zdrojový kód 5.8: Rozhraní repository pro User agregát.

```
namespace Server.AccountService.Core.Aggregates.UserAggregate;

/// <summary>
///     Specific repository for its aggregate root entity.
/// </summary>
public interface IUserRepository : IAggregateRepository<User, Guid>
{ }
```

Zdrojový kód 5.9: Ukázka implementace User agregát repository.

```
namespace Server.AccountService.Infrastructure.Repositories;

/// <summary>
///     Implementation of <see cref="User"/> aggregate root repository.
/// </summary>
internal class UserRepository
  : RepositoryBase<
      User,
      ApplicationUser,
      IDataRepository<ApplicationUser, AccountDbContext, Guid>,
      Guid>,
    IUserRepository
{
  public UserRepository(
    IDataRepository<
      ApplicationUser, AccountDbContext, Guid> persistenceRepository,
    IMapper mapper)
    : base(persistenceRepository, mapper)
  { }
}
```

Návrh struktury je nyní jasný. Je tedy nutné tento návrh aplikovat na reálný příklad, kterým může být uživatel. Ukázka kódu 5.8 vytvoří základní rozhraní pro možnou práci s repository. Kdykoli bych v kódu pracoval s repository, budu používat toto rozhraní a z toho důvodu dědí od `IAggregateRepository`, aby s sebou mělo přístupné všechny základní vlastnosti celé repository struktury. V rámci toho rozhraní lze definovat i vlastní metody specifické jen pro agregát uživatele. Pro představu lze vidět i implementaci tohoto rozhraní v ukázkovém kódu 5.9.

Nyní se vrátím zpět ke zmiňovaným problémům z naivního řešení. Již jsem naznačoval, že je nelze zcela eliminovat, ale vymyslel jsem řešení, jak je redukovat. Specification pattern nelze použít, ale jen do jisté míry. Rozhodl jsem se tedy implementovat Specification pattern, kterým nelze sestavit dotazy přes více než jednu databázovou tabulku. To se může zdát neužitečné. Ale pro jednoduché dotazy, toto řešení odstraní nutnost vytvářet nespočet vlastních metod v `IUserRepository`, protože lze využít `IReadRepository`, které používá specifikace.

Příkladem může být filtrování uživatele podle jeho jména a příjmení. Za normální situace bych musel vytvořit speciální metodu v `IUserRepository` pro filtrování podle jména, další podle příjmení a poslední podle obou. Toto však platí, pokud je dotaz složitý a vyžaduje získání dat i z jiných tabulek. Ovšem pro jednoduché dotazy bez nutnosti spojování lze vytvořit specifikaci na úrovni doménových modelů a tím pádem redukovat 3 metody do jedné obecné pro zpracování specifikace.

Nelze tedy plně odstranit všechny překážky, ale lze je do jisté míry značně redukovat, a pokud se bude dodržovat návrh pomocí specifikací pro jednoduché dotazy, tak potom lze udržet repositáře pouze s nezbytně nutnou implementací vlastních metod. Ve výsledku zde mám řešení, které je zcela nezávislé na databázovém systému. Jedinou vadu na kráse, kterou toto řešení přináší je samotná definice složitých dotazů, které musí být definované v Infrastructure vrstvě. Ale díky použití částečného patternu Specifikací lze tuto nutnost značně zredukovat.

5.2.3.1 Event Store

Součástí RDBMS knihovního rozhraní je také implementace tzv. uložiště akcí, které je založené na Event Sourcing patternu. Toto řešení nabízí přizpůsobený databázový kontext a speciální repository. Tyto repository nejsou zobrazeny ve výše uvedených diagramech z důvodu samostatné hierarchie rozhraní. Nabízené funkce však v této práci nejsou plně využity, nicméně možnost je použít zde je a v budoucnu funkce tohoto uložiště budou více než užitečné, například v souvislosti se správou výzev, kdy bude vhodné se vracet a sledovat své statistiky v historii.

5.2.4 Správa geolokačních dat

Již v této kapitole je zmíněna technologie PostGIS, kterou jsem si vybral pro ukládání geolokačních dat. Na rozdíl od možných alternativ jako je MongoDB, tak PostGIS funguje jako rozšíření pro PostgreSQL a i z toho důvodu vyžaduje o něco vyšší péči při nasazení a integraci do projektu.

PostGIS podobně jako jiné technologie toho typu, nabízí dva způsoby, jak data ukládat [Bas18]:

- ***Geometry*** - data jsou reprezentována s předpokladem, že existují na kartézské rovině.
- ***Geography*** - s daty je zacházeno, jako kdyby tvořili body na povrchu Země, tj zakřivený povrch.

Všechna tato data spadají ke standardu ***EPSG Geodetic Parameter Dataset***, který je veřejným registrem geodetických údajů, prostorových referenčních systémů, zemských elipsoidů, transformací souřadnic a souvisejících měrných jednotek, založený členem European Petroleum Survey Group (EPSG) v roce 1985. Většina geografických informačních systémů (GIS) a knihoven GIS používá kódy EPSG jako Spatial Reference System Identifiers (SRIDs) a definiční data EPSG pro identifikaci souřadnicových referenčních systémů, projekcí a provádění transformací mezi těmito systémy [23m].

Mezi nejpoužívanější systémy patří ty s kódovým označením EPSG 4326 - WGS 84, který pracuje s daty jako se souřadnicovým systémem zeměpisné šířky a délky založený na těžišti Země, který je mimo jiné používaný Globálním Pozičním Systémem (GPS). Druhým velmi používaným je EPSG 3857, jde o Mercator projekci mapy. Tento systém se používá zejména v nástrojích pro projekci mapy jako jsou například Google Maps. V mém případě se zcela nabízí první zmíněný systém, který zároveň pracuje s mnou používaným systémem GPS.

Tyto informace je zcela nezbytné definovat v implementaci při definici databázové struktury. V názorném výpisu 5.10 lze vidět uplatnění výše uvedených systémů a kódových označení. Zároveň je zde možné se povšimnout konkrétního indexování datového sloupce `Location`, které je nezbytné pro efektivní vyhledávání v datech a PostGIS využívá efektu indexování pro rychlé vyhledávání pomocí geohashů. Pro příklad jsem uvedl ukázku 5.11, která demonstruje vyhledávání všech bodů v okolí jednoho specifického bodu o rozsahu daného rádiusu. Data jsou následně během tohoto procesu mapována do příslušných DTO, aby přemapování nemuselo být prováděno v paměti aplikace.

Zdrojový kód 5.10: Ukázka implementace konfigurace databázového modelu pro ukládání bodů mapy.

```
namespace Server.WaypointService.Infrastructure.Data.Configurations;

internal class WaypointConfiguration
  : IEntityTypeConfiguration<WaypointDataModel>
{
  public void Configure(EntityTypeBuilder<WaypointDataModel> builder)
  {
    builder.ToTable("Waypoints");

    // Add PostGIS extension annotation
    builder.HasAnnotation("PostgresExtension:postgis", "");

    // Configure the Location property
    builder.Property(o => o.Location)
      .IsRequired()
      .HasColumnType("geometry(Point,4326)")
      .HasDefaultValueSql("'POINT(0 0)'::geometry");
    // Create a GIST index on the Location property
    builder.HasIndex(e => e.Location).HasMethod("GIST");

    // Configure the Type property
    builder.Property(o => o.Type)
      .IsRequired();
  }
}
```

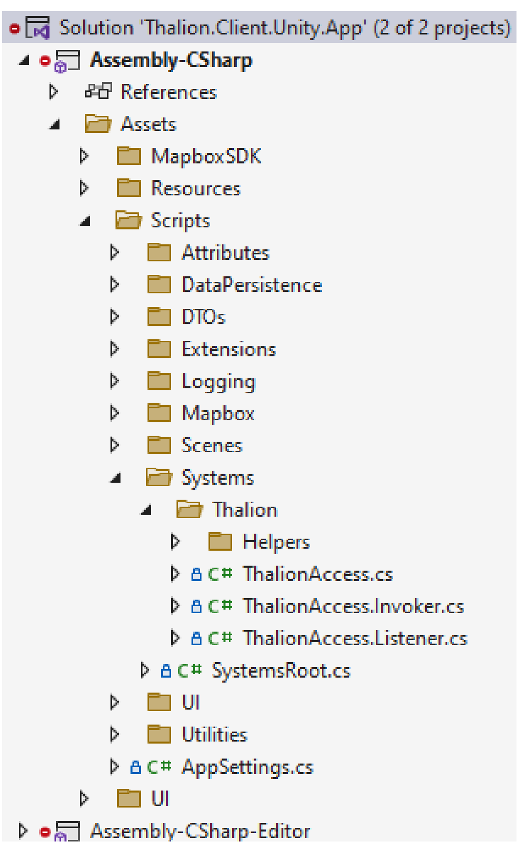
Zdrojový kód 5.11: Ukázka implementace metody pro čtení datz z databáze v okruhu stanoveného bodu a nastaveného rádiusu.

```
namespace Server.WaypointService.Infrastructure.Repositories;

public async Task<ICollection<WaypointDto>> ReadAllInRadiusAsync(
  Location centerPoint, int radius)
{
  var center
    = new Point(centerPoint.Longitude, centerPoint.Latitude) {SRID = 4326};

  var queryable = PersistenceRepository
    .GetQueryableBySqlRaw($@"
      SELECT *
      FROM ""WaypointService"".""Waypoints""
      WHERE ST_DWithin(""Location""::geography,
        ST_SetSRID(ST_Point({center.X}, {center.Y}), 4326)::geography,
        {radius})
    ");

  var waypoints
    = await Mapper.ProjectTo<WaypointDto>(queryable.Select(o => o))
      .ToListAsync();

  return waypoints;
}
```

5.3 Architektura klientské aplikace

Tato sekce popisuje infrastrukturu klientské části systému, kterou tvoří klientská aplikace vyvíjená pomocí herního enginu Unity. Oproti serverové části se zde jedná o velmi jednoduchou architekturu, kde převážná část je dána používaným standardem Unity [23h].

Unity je velmi mocný nástroj, zejména pro vývoj her. Ale i když je toto jeho primární účel, tak se dá využít i jinými způsoby a od různých realistických vizualizací až po mobilní aplikace. Velká část práce se vyvíjí přímo v samotném editoru Unity, který poskytuje bohaté možnosti editace, od editoru animací až po skriptování. Pokud je tedy nutné vytvářet vlastní programovou logiku, je nezbytné vytvořit takovou infrastrukturu, která bude zachovávat udržitelnost projektu. V rámci Unity projektu lze vidět 5.5 náhled struktury projektu. Samotné C# skripty se nacházejí ve složce `Scripts`. Ostatní složky vytvářejí objekty vztažené k editoru Unity.



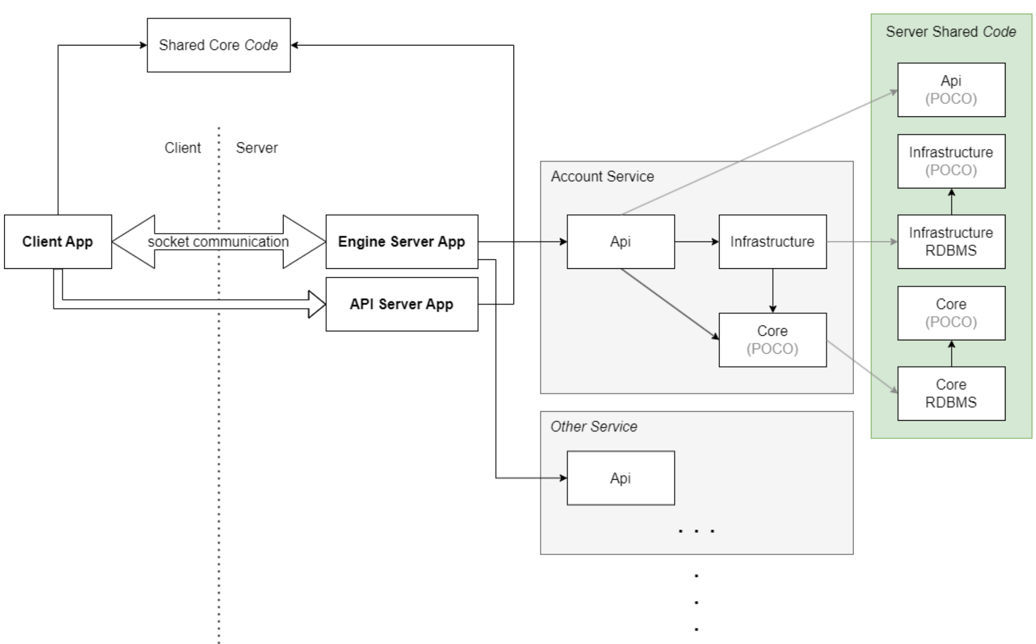
Obrázek 5.5: Struktura projektu klientské aplikace.

Jednotlivé skripty jsou tříděny dle scén, které se v aplikaci nachází, tj. například „main menu“ a „game“. Každý skript je následně vytvářen v příslušné podsložce projektu s názvem dané scény. Skripty, které však nepřiléhají k žádné konkrétní scéně pak tvoří zbytek struktury. Abych konkrétní popis této struktury zavedl konkrétně k problematice tématu práce, potom je zde nejdůležitější třída `ThalionAccess`, která je rozhraním pro komunikaci se serverem. Více však zmíním v další sekci 5.4.

5.4 Komunikace a přenos dat v systému

V této kapitole jsem již zmínil, že je velmi důležité a nezbytné vytvořit takové serverové rozhraní, které bude nabízet komunikaci pomocí REST API pro klientské aplikace, tak i možnost klientských aplikací se k serveru připojit pomocí soketové komunikace.

Již v sekci 5.1 na obrázku 5.3 byla vidět projektová infrastruktura na straně serveru. Nyní tuto strukturu rozšířím o návrh komunikace mezi serverem a klientskými aplikacemi na obrázku 5.6. Tento obrázek zobrazuje diagram, kde je možné si všimnout již zmiňovaných služeb, ale tentokrát zapojených do celého systému. Rozhraní pro komunikaci na straně serveru vytváří samotné aplikace serverů, kde existují dvě zmíněné serverové aplikace poskytující jedna REST API (Api) a druhá soketovou komunikaci (Engine). Tyto aplikace mají závislost pouze na takových službách, které doopravdy používají.



Obrázek 5.6: Diagram struktury závislostí celého systému.

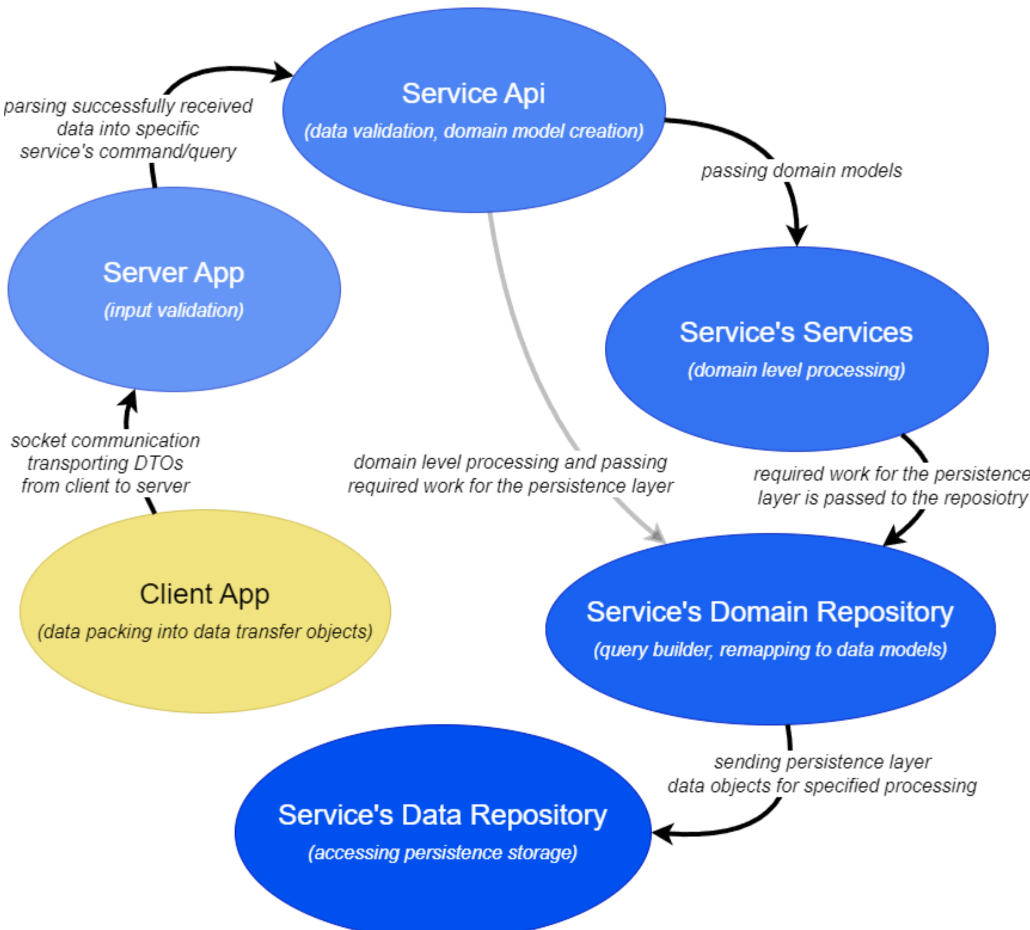
Je dobré si povšimnou projektu Shared.Core, který je sdílený mezi aplikacemi serveru a klientskou aplikací. Jedná se o značnou výhodu, že aplikace klienta je taktéž vyvíjena v programovacím jazyce C#, a proto jsem schopný vytvořit DLL knihovnu, kterou jsem schopný dodat do klientské aplikace. Tato knihovna obsahuje především veškeré objekty DTO potřebné ke komunikaci mezi klientem a serverem a celkově tak vytváří rozhraní usnadňující přehlednost komunikace.

Soketovou komunikaci je nutné autorizovat pouze jednou a následně je spojení trvale udržováno. Bylo by možná rozumnější nejdříve uživateli předat autorizační klíč pro soketové připojení. Ve chvíli připojení si již můžeme být jisti, že je uživatel autorizován pod svým účtem. Pro tyto účely je REST API skvělou volbou a zároveň tím odlehčíme serveru pro soketovou komunikaci veškerou správu autorizací.

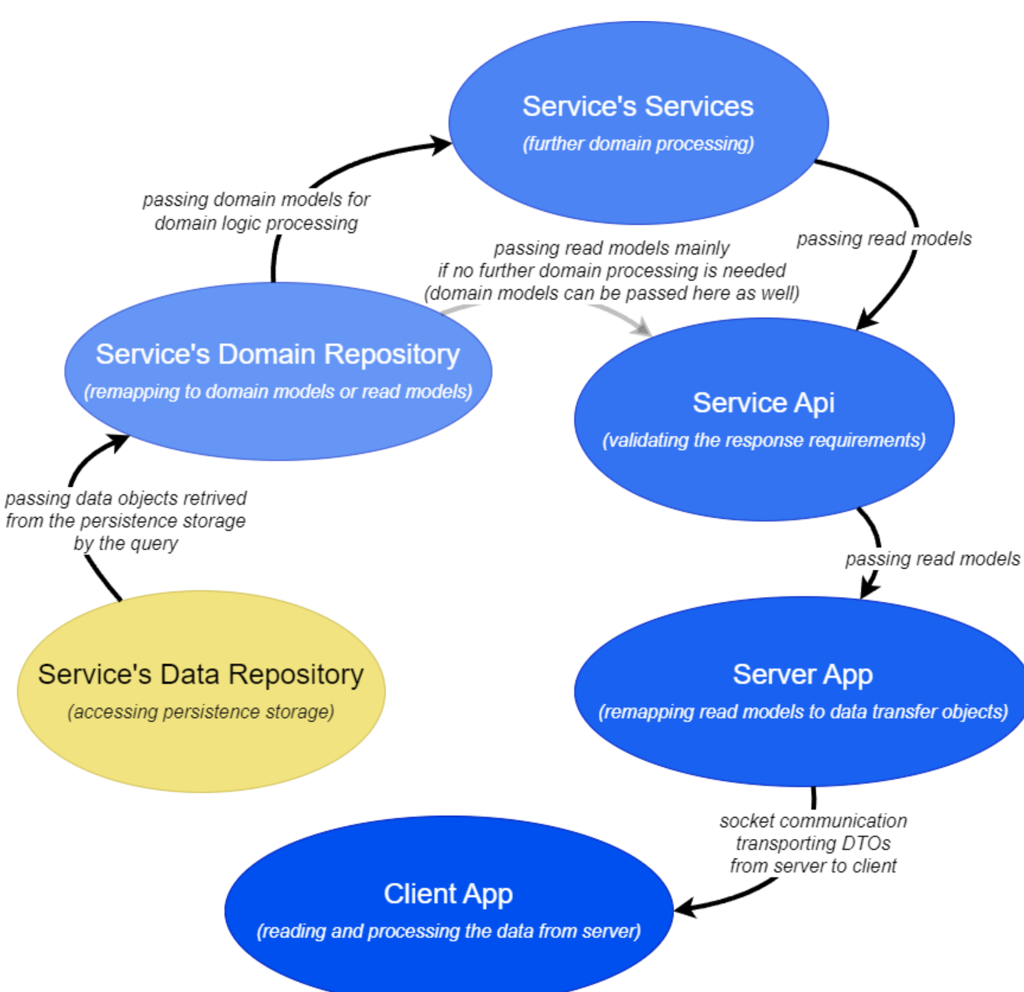
Příklad zahájení komunikace ze strany klienta může být následující:

1. Uživatel zadá své přihlašovací údaje do aplikace a spustí proces přihlašování.
2. Požadavek (řádně ošetřená data) se vyšle na Api server, kde dojde k jeho zpracování.
3. Při úspěšném ověření uživatelských údajů, Api server navrátí uživateli jednorázový autorizační klíč.
4. Uživatelská aplikace při obdržení tohoto autorizačního klíče vyšle požadavek pro připojení na Engine server a k požadavku přiloží tento autorizační klíč.
5. Engine aplikace rozpozná uživatele a vytvoří mezi aplikací a serverem trvalé připojení.
6. Uživatelská aplikace při úspěšném navázání spojení přepne aplikaci do autorizované oblasti.

Abych toto řešení více přiblížil, tak jsem vytvořil jednoduché data flow diagramy, na kterých lze proud dat jednoduše vidět. Na obrázku 5.7 lze vidět komunikaci, která vytváří klientský požadavek pro získání dat ze serveru a tento požadavek bude vyžadovat přístup k datům z databáze. Druhý příklad 5.8 je přesně opačný a to takový, kde jsou data získány z databáze. Obrázek pak znázorňuje cestu těchto dat až k návratu do klientské aplikace.



Obrázek 5.7: Data Flow diagram přenosu dat ve směru klient->server.



Obrázek 5.8: Data Flow diagram přenosu dat ve směru server->klient.

5.5 Zobrazování polohy zařízení a implementace Mapbox SDK

Tato sekce popisuje způsoby implementace klientské aplikace se zaměřením na integraci mapy a dotazování se na mapové API, kde aplikace získává data o aktuální poloze, kde se daný uživatel (zařízení) nachází.

S rozhodnutím použití Unity pro vývoj, tak zde padla otázka, jakým způsobem integrovat zobrazení mapy. Samotné získání polohy se nejevilo jako velký problém, jelikož poloha je snadno získatelná ze zařízení, na kterém aplikace běží. Jelikož se bude jednat pouze o mobilní telefony Android, případně iOS v budoucnu, tak to nevytváří ani žádné omezení pro takový způsob získávání polohy.

Mapbox nabízí SDK pro Unity, které již poskytuje základní možnosti interakce a vykreslování mapy přímo v Unity. Během snahy nasadit toto řešení jsem byl vystaven problému, který byl nutný vyřešit ještě před samotným použitím SDK. Jedná se o samotnou verzi Unity. Vývoj Unity jde rychle dopředu a od nové verze 2019 lze vývoj provést v jiném rendering systému, než je Built-In. Takovým je URP, který je směřovaný a optimalizovaný pro mobilní hry[23ay]. Kvůli následnému ořezu mapy okolo hráče je nutné jej použít z principu, že nabízí technologii Shadergraph[23at], kterou Unity poskytuje, protože Built-In rendering systém jej nepodporuje.

Při prvním sestavení testovacího dema aplikace jsem měl dvě identické verze, jednu vytvořenou v URP a druhou v Built-In rendering pipeline. I přes značné širší možnost nastavení URP jsem nedokázal docílit takového výkonu aplikace jako jsem získal při Built-In. Pro představu se jednalo o rozdíl až nízkých stovek FPS[9], kdy URP bylo schopno dosáhnout pouhých 25 FPS při vykreslování mapy. Tento výsledek mě velice zaskočil a při další zkoumání jsem na technickém fóru Unity objevil diskuse s podobnými problémy. Při hlubším rozboru tohoto problému jsem nabyl informací, že je URP vhodné pro zařízení vyššího výkonu, a především s výkonnou grafickou kartou. Při širším použití mohou nastat problémy, jako tomu bylo při testování na mém zařízení. A to z návrhu použití pro širokou veřejnost se jistě takto omezovat nechci.

Při návrhu jsem měl tedy velice těžké rozhodnutí, kdy jsem musel zanechat použití Shadergraph a zůstat u standardního renderovacího systému Unity. Nový UI Toolkit byl přístupný od verze 2021, ale až o rok později plně funkční, a proto jsem vývoj započal v LTS verzi 2022 [23ax]. Toto řešení nové verze s sebou nabízelo další

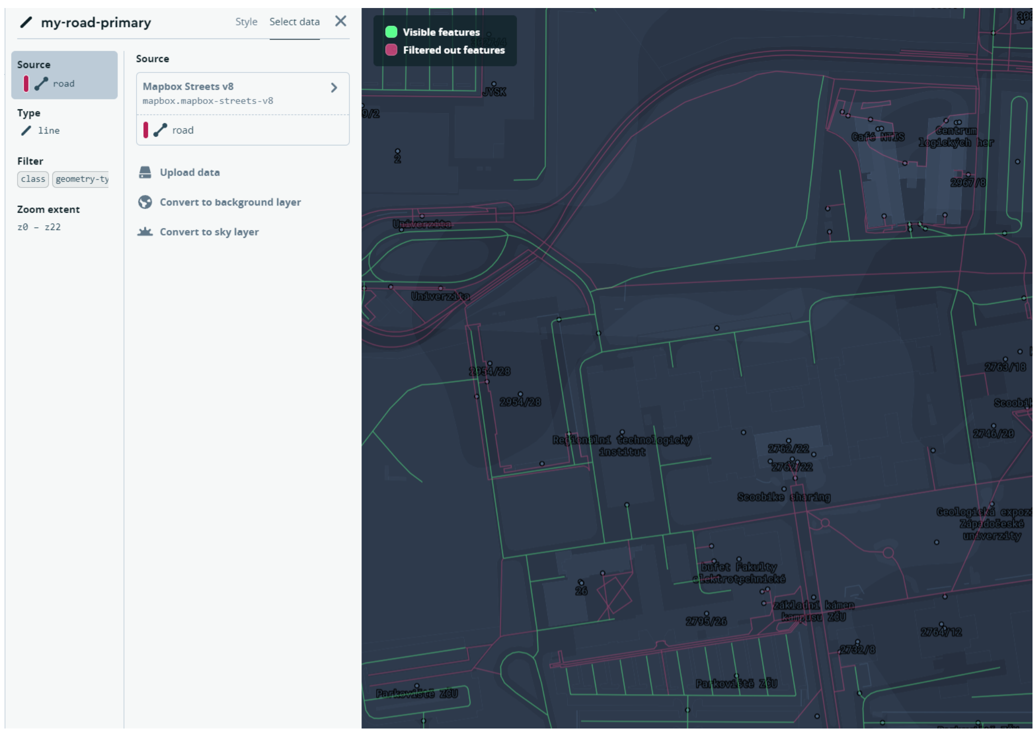
[9] Frame per Second

výhody jež byla podpora pro Shadergraph v Built-In renderovacím systému, který se dal k projektu připojit jako rozšíření. V době návrhu (tj. rok 2022) o takové možnosti moc informací nebylo. Na základě toho vytvořil první testovací demo, kde jsem ověřil funkčnost aplikace na zařízení s podobným výkonem, jako při použití URP.

Základní návrh o použitých verzích jednotlivých technologií je nyní dán a dále popis rozdělím do menších podsekcí, kde každá bude řešit konkrétní problematiku.

Zajímavostí a zároveň velkou problematikou, kterou v jedné ze sekci popisuji, je použití verze Mapbox 2.1.1, která je nejnovější a zároveň 4 roky stará[23z].

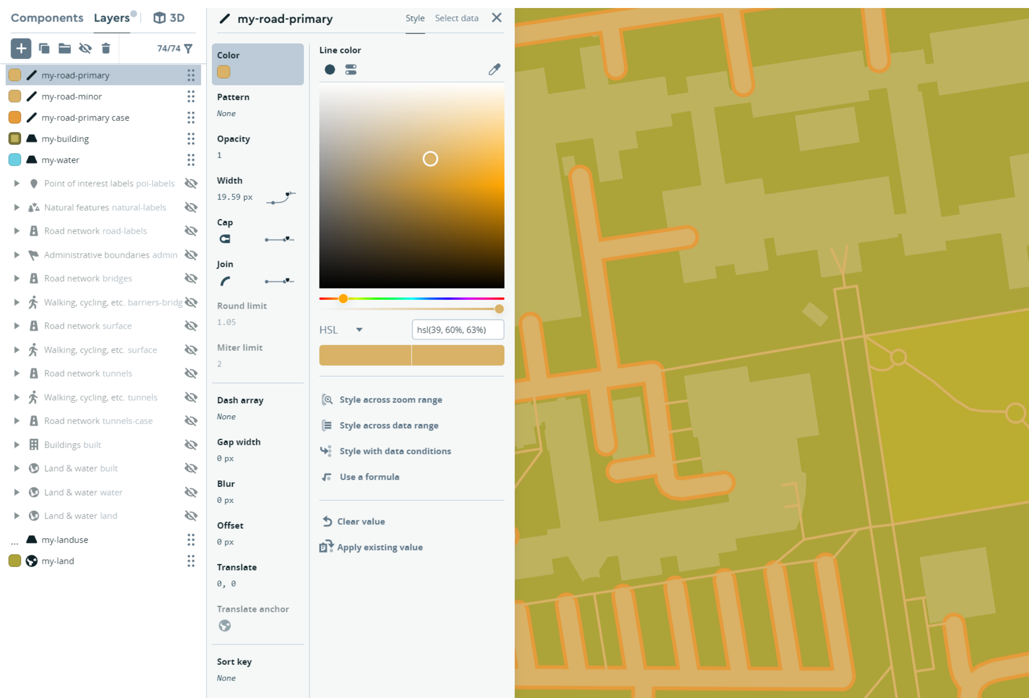
5.5.1 Získávání geolokačních dat



Obrázek 5.9: Náhled Mapbox map editoru s vlastních výběrem dat.

Získávání dat v dohledu hráče je z velké části zajištěno sw pomocí dostupného Mapbox SDK. Mapbox SDK je open-source a je možné si kód přizpůsobit dle svých potřeb. SDK již zajišťuje řešení pro získání mapových dat pomocí jejich vlastního API. Získaná data jsou následně přemapována do dlaždic, které mohou reprezentovat mapu. V mé implementaci používám textury stylu, který jsem si definoval pomocí Mapbox editoru (viz obr. 5.10). Pomocí tohoto editoru lze také upravit,

která data stahovat. Editor je velmi podrobný a umožňuje vysokou míru vlastního přizpůsobení. Vytvořil jsem vlastní šablonu, která vybírá pouze nezbytně nutná data pro zobrazování mapy. Na obrázku 5.10 lze vidět definovaný vlastní styl, ale na předchodím obrázku 5.9 lze vidět téměř stejnou výseč mapy se stejnými vybranými daty, ale v jiném náhledu a to výběru dat.



Obrázek 5.10: Náhled Mapbox map editoru s vlastním stylem.

Ve chvíli, kdy jsou data reprezentovaná pomocí objektů Unity, tak je lze vykreslit do prostoru. Zde je několik možných návrhu, které s sebou SDK přináší. Základním a jednoduchým způsobem je vykreslování v mřížce kolem hráče. Druhým způsobem je vykreslování na základě úhlu kamery a prostoru, který zaměřuje. Druhý způsob umožňuje efektivnější vykreslování a nevykresluje dlaždice, které uživatel stejně nevidí. Následně je možné vytvořit jednoduchou cache, která objekty udrží načtené v prostoru, ale neviditelné, aby se při otáčení kamery nemusely znovu vykreslovat a načítat nová data.

5.5.2 Oprava chyb a zpětná vazba k vývojářům

Mapbox SDK nabízí nejnovější verzi 2.1.1, která je ovšem publikovaná v roce 2019, ve kterém zároveň Unity prezentovalo nový renderovací systém URP. Toto hrálo také velkou roli, kdy jsem zvažoval renderovací systém a Mapbox SDK jej nepodporovalo. Mimo to, v novějších verzích Unity (tj. 2022, ve které pracuji) je nefunkční

modul SDK pro rozšířenou realitu. Je nutné tento fakt vzít v úvahu. Nicméně pro stanovené plány to není omezující podmínka. V návrhu jsem s tímto musel počítat a při analýze jsem tento fakt také řešil.

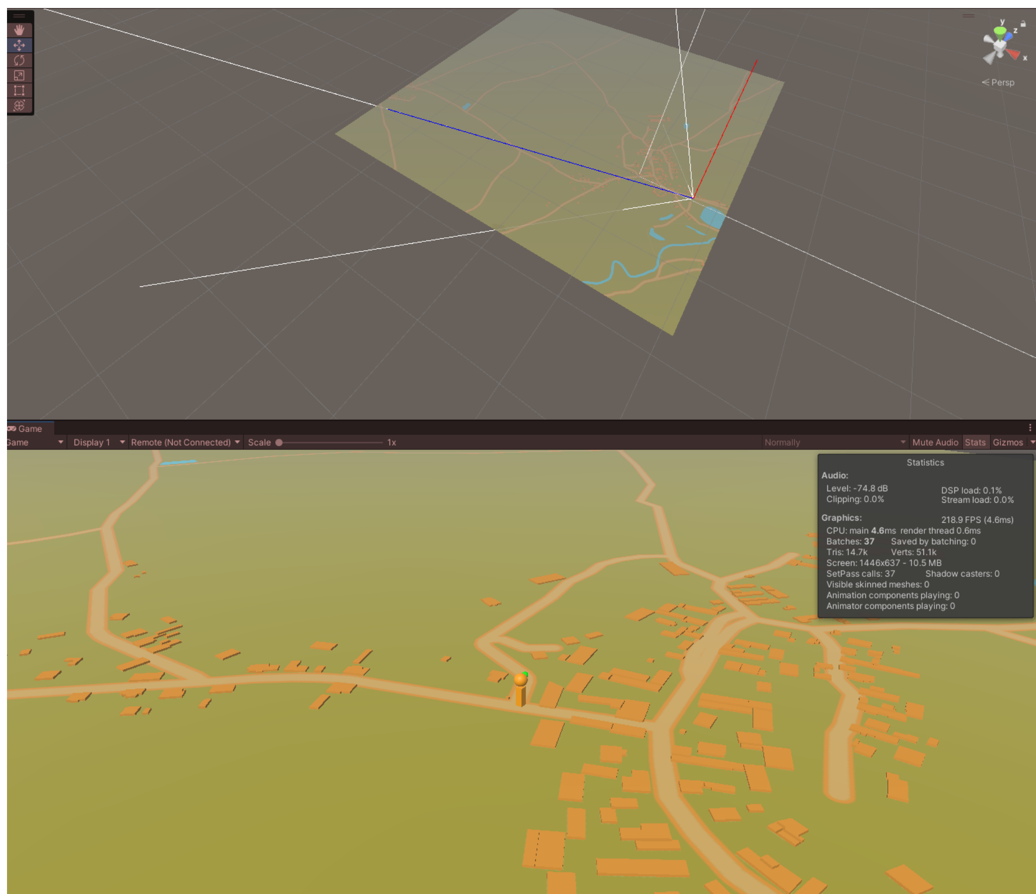
V předchozí sekci 5.5.1 jsem popisoval způsob získávání dat a tento popis zakončil návrhy vykreslování mapy, které nabízí Mapbox SDK. Po delším zkoumání a práci s tímto SDK jsem postupně začal objevovat nedostatky, které SDK s sebou přináší.

Prvním nedostatek mě potkal ihned při tvorbě prvního dema, které nejdříve z blíže nespecikovaných důvodů zamrzlo a přetížilo Unity editor, kdy bylo nutné ukončit proces. Bylo nutné projít a analyzovat kód SDK pro bližší pochopení funkčnosti. Výsledkem jsem docílil nalezení příčiny, která mě značně překvapila. Jednalo se o chybu, která nastávala při výpočtu vykreslování jednotlivých dlaždic na mapu a Mapbox ani žádná diskusní fóra tento problém neřešila. Při použití metody pro vykreslování dlaždic mapy ve výseči mapy se však nijak neuvažovalo nad situací, kdy uživatel nastaví kameru na horizont. V tomto případě se algoritmus pokoušel načítat „nekonečné množství dlaždic" a tím i velké množství API dotazů. To vysvětlovalo kompletně zahlcené internetové připojení v daný moment.

Pro vyřešení tohoto problému jsem vytvořil vlastní službu poskytující výpočet pro vykreslení, který předává informace dále pro API, která data jsou nutná načítat. Řešení jsem zakládal na pdoobných principech, které byly řešeny v Mapbox SDK. Vlastním řešením jsem následně docílil vykreslování mapy pod úhlem kamery a pouze v předem vybraném počtu dlaždic, které lze jedním směrem vykreslit. Toto aplikované řešení lze vidět na obrázku 5.11, kde kamera směřuje daným směrem a je vykreslován pouze nezbytně nutný počet dlaždic.

Při pokračování vývoje a testování schopností SDK, tak jsem dále narazil na další chyby, které se v tomto SDK vyskytovaly. Jednou z dalších chyb, kterou SDK trpělo, je zasekávaný pohyb během rotace kamery. Během dalšího zkoumání jsem zjistil, že proces pro ukládání dat do cache paměti není nijak zvlášť optimalizovaný. Proto jsem celý proces upravil, optimalizoval a tím odstranil potíže.

Když těchto potíží začalo přibývat, rozhodl jsem se vytvořit krátký článek na oficiální Mapbox vývojové platformě a prezentovat řešení, kterými tyto chyby opravit. Všiml jsem si, že vývoj stále pokračuje, ale pomalu a nelze říci, kdy očekávat vydání další verze s opravenými chybami. Nicméně již po několika málo dnech, tento článek dosáhl několika sdílení s odkazem na něj pro řešení problémů jiných uživatelů [22a].



Obrázek 5.11: Náhled do editoru Unity s pohledem na mapu vykreslenou pomocí Mapbox SDK.

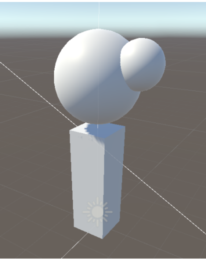
5.5.3 Pohyb po mapě a výseč pro hráče na mapě

Samotné vykreslení mapy není však jedinou nutnou záležitostí. Je nutné vyřešit ještě pohybu postavičky, která reprezentuje hráče čili našeho uživatele aplikace. Mapbox SDK dodává pouze primitivní skripty pro pohyb na mapě, a proto jsem se rozhodl vytvořit svůj vlastní, který bude vhodný pro mé konkrétní užití. Také bylo potřeba nově implementovat pohyb kamery. Pro pohyb kamery je nutné dát možnost uživateli otáčet kameru kolem své postavičky a schopnost přibližovat kameru pomocí standardních gest na dotykové obrazovce.

Abych dodal trochu dynamiky do tohoto pohybu, rozhodl jsem se přidat pro kameru setrvačnost, která z lineární pohybu vytvoří jemný rozjezd a dojezd kamery z původního místa do cílového. Samotné trasování postavičky na mapě byla o něco složitější úloha, kdy bylo nezbytné zachovávat směr, kterým postavičky směřuje. Zároveň, je nutné navrhnout rozumný způsob, jak dlouho má být postavička pře-

souvána během jednoho pohybu. Čas trvání nemůže být konstantní a musí se odvíjet od vzdálenosti, kterou postava ujde a která může být variabilní.

Aby bylo možné vůbec rozpoznat směr postavičky (kterým míří), tak jsem vytvořil testovací objekt s velmi jednoduchou napodobeninou nosu, který udává směr pohybu (viz obr. 5.12). Skript zajišťující pohyb a rotaci postavičky jsem umístil do zmíněného článku, který jsem přiložil jako bonus pro zájemce řešící podobné návrhové problémy, které SDK přináší [22a].



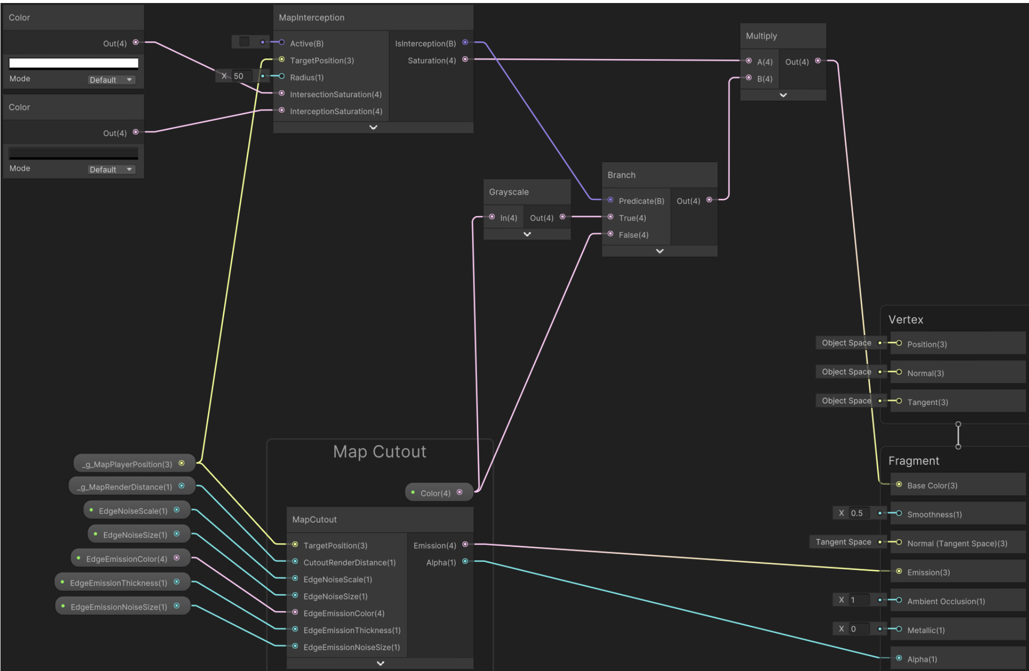
Obrázek 5.12: Používaná mapová postavička pro vývoj v Unity.

Pro herní účely této aplikace by bylo velmi vhodné umožnit vývojáři pracovat s okrajem mapy. Nyní hráč vidí prakticky až do okraje zobrazovacího pole. Nicméně, pokud se uživatel rozhodne mapu přiblížit s náklonem do dálky, tak s velkou pravděpodobností narazí na vizuální chybu, která představuje viditelný ostrý přechod z mapy do ničeho. Toto by bylo vhodné v budoucnu doplnit speciálním efektem nebo jednoduchým přechodem do ztracena.

Aby byl takový efekt možný, je nutné jednoho celistvého objektu, který lze oříznout pomocí masky. Mapu jsem však takovým způsobem nevytvářel. Nejsem si jistý, zda by bylo možné takového řešení s jedním objektem mapy dosáhnout, protože se nyní snažím mapu vykreslovat po dlaždicích, z důvodu efektivního načítání dat. V tomto případě je nutné pro každou dlaždici spočítat výseč, která se dynamicky bude stále přepočítávat v závislosti na poloze postavičky hráče.

Pro takové účely se naprosto hodí již zmiňovaná technologie v Unity - Shadergraph. Pomocí této technologie lze aplikovat speciálně programově řízený materiál, který vylepší daný objekt. Na základě polohy hráče lze dynamicky měnit materiál tohoto objektu (jedné dlaždice mapy). Cílem je implementovat takový materiál, aby byl viditelný v daném rádiusu a neviditelný ve zbytku. Implementace a používání rozhraní Shadergraph není zcela triviální a vyžaduje hlubší znalosti o tvorbě materiálů. Pro jednoduchou ilustraci jsem do této práce vložil několik náhledů, jak implementace v tomto editoru vypadá.

Základním objektem je zde Fragment (viz obr. 5.13), který vytváří vstupní pole hodnot, kterými lze daný materiál objektu modifikovat. Samotné modifikace jsou ve zbytku grafického editoru Shadergraph implementované ve formě funkcí. Ty lze vidět na obrázku 5.13 pod označením „Map Cutout“.

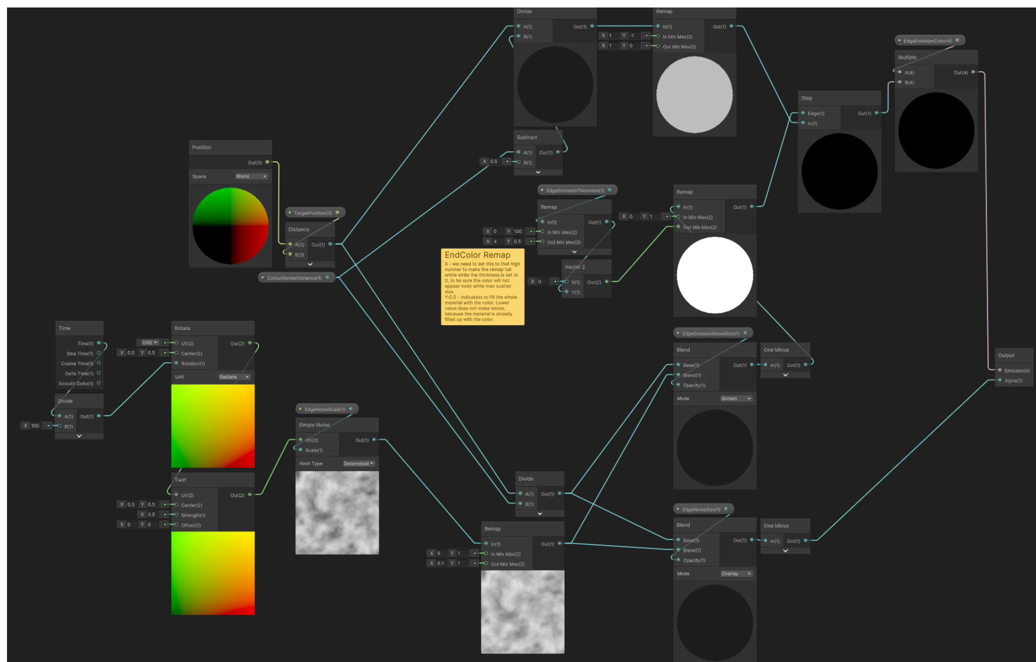


Obrázek 5.13: Pohled na Shadergraph rozložení zajišťující efekt výseče.

Tato zmíněná funkce je již poněkud složitější, a proto nebudu zacházet do hlubších detailů a zůstanu pouze u náhledu, který lze vidět na obrázku 5.14. Tato implementace vytváří efekt rozplývajícího se kouře u okraje mapy, který lze vidět na jednom z dalších obrázků, například obrázek 5.16. Pomocí základních operací lze pracovat s daty, které dynamicky poskytuje Unity. Na základě získávané polohy (viz obr. 5.15) s touto polohou pracuji, přepočítávám relativní vzdálenost k postavičce hráče a na těchto faktech následně vytvořím pouze tu část vyseče, která se má zobrazit na konkrétní dlaždici, kde je tento materiál aplikovaný.

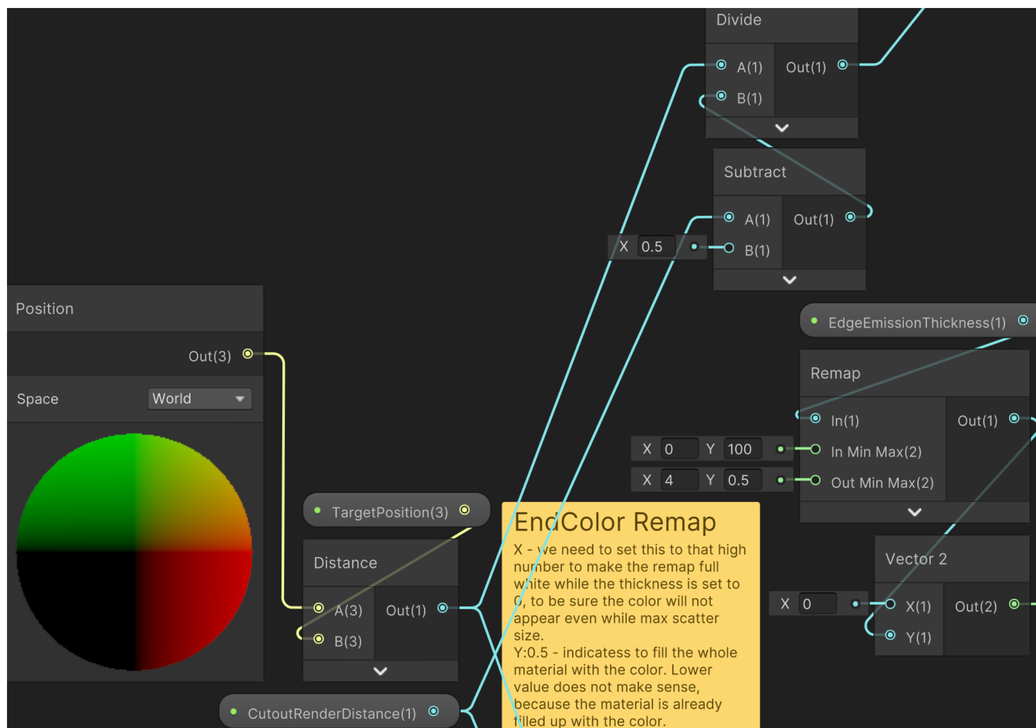
Následně, přesunu-li se z implementační části více do té prezentační, tak pro uživatele vytvořím efekt, kde okraj mapy mizí do ztracena pomocí speciálního efektu. Tento efekt lze snadno měnit a hlavně dynamicky upravit za běhu aplikace.

Na obrázku 5.16 lze vidět pohled do editoru, kde probíhá vykreslení mapy na základě pozice kamery do prostoru. Toto vykreslení zároveň již obsahuje aplikace zmiňované výseče. Na snímku je část mapy bez vykreslených budov. Je to tak záměrně zachycené v rychlém pohybu přes mapu a demonstruje to, že se mapa načítá po částech. Základní vrstvu tvoří texturový podklad, ale veškeré objekty, které mapu tvoří 3D se načítají až následně.

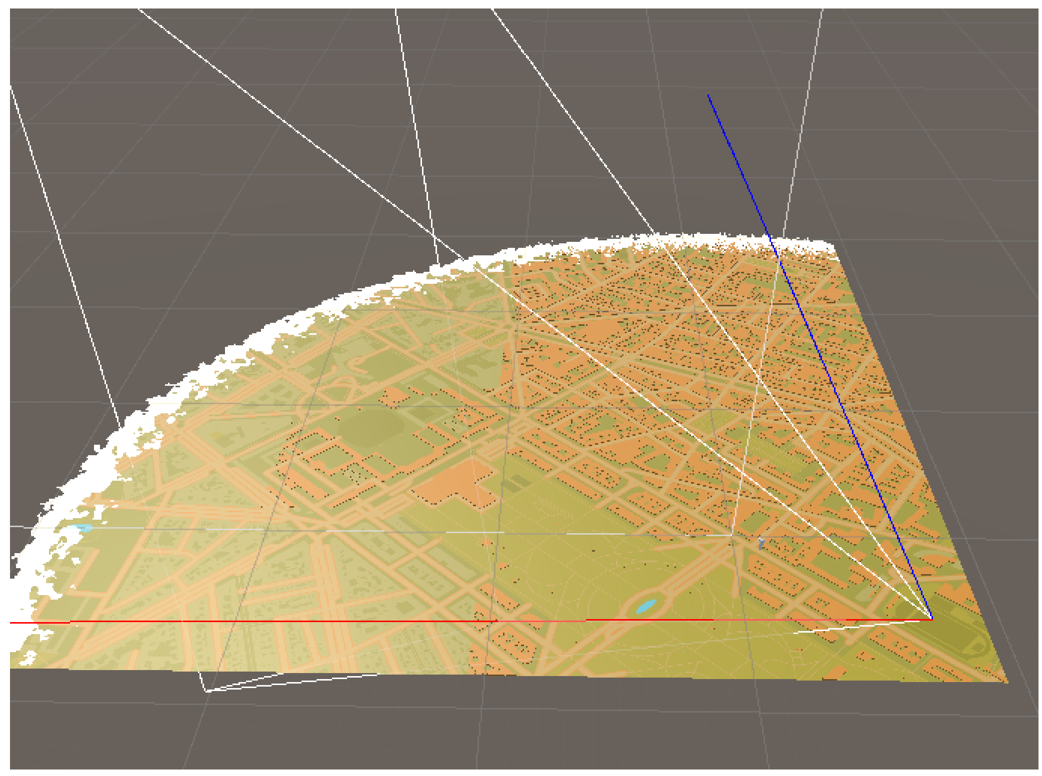


Obrázek 5.14: Rozložení prvků grafu pro tvorbu mapové výseče v okolí hráče.

Pohled zobrazený na obrázku 5.16 je tentýž, který lze vidět na obrázku 5.17. Zde však s tím rozdílem, že je vidět přesný pohled uživatele na jeho zařízení.



Obrázek 5.15: Detailní pohled na graf výseče, zaměřený na část práce s pozicí.



Obrázek 5.16: Aplikace výseče na výkres mapy podle pozorovacího úhlu uživatele.

Obrázek 5.17: Pohled uživatele s aplikovanou výsečí v prostředí emulátoru.

5.5.4 Synchronizace se serverem, vykreslování dat

Tato sekce popisuje implementaci zaměřenou na zobrazování mapových informací na mapě.

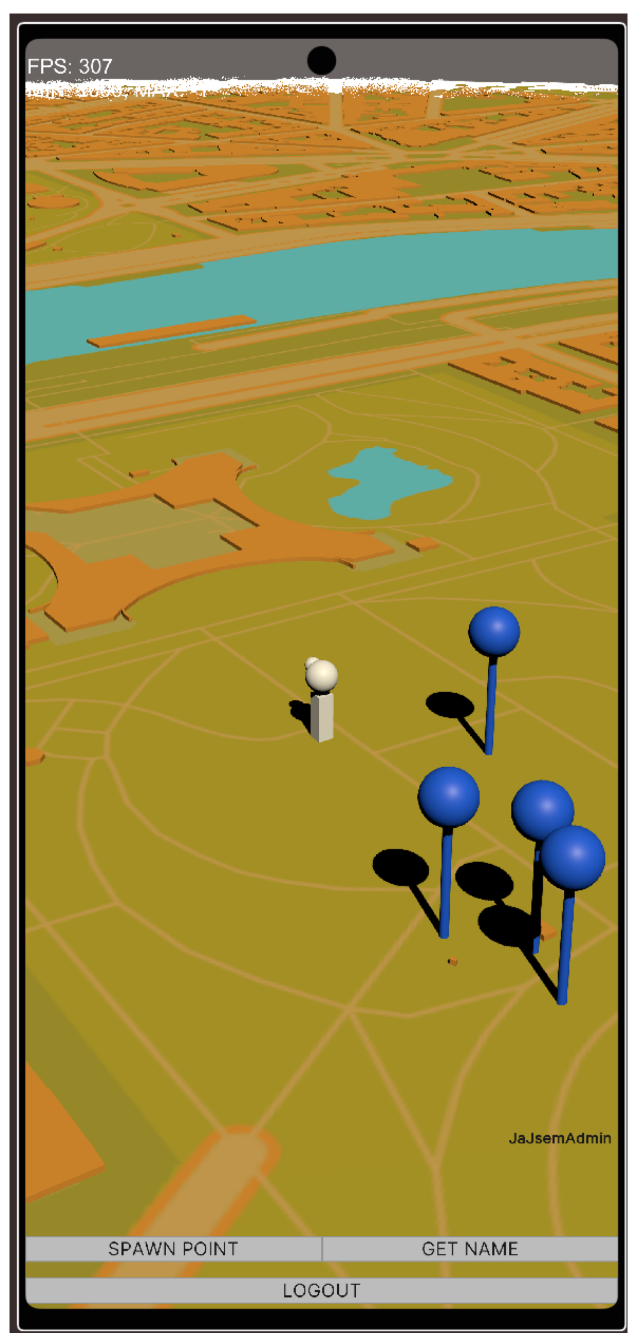
Systém musí být schopen sdílet pozici uživatelů mezi sebou. Lokace hráčů jsou ukládány do databáze, kde jsou pravidelně aktualizovány na základě jejich aktivity.

Pro jednodušší demonstraci místo samotných hráčů budu zobrazovat body mapy. Jedná se prakticky o to samé. V uvedeném příkladu se vyhledávají a zobrazují body v okruhu 200 metrů kolem aktuální pozice hráče. Na obrázku 5.18 lze vidět několik již existujících bodů aplikovaných na mapě v blízkosti hráče. Na základě pozice hráče však tyto body nemusí být již viditelné. Pokud se hráč vzdálí příliš daleko, tak se tyto body přestanou zobrazovat a místo nich bude skenována nová lokace a pravděpodobně se zobrazí i jiné body, pokud v dané lokalitě nějaké existují.

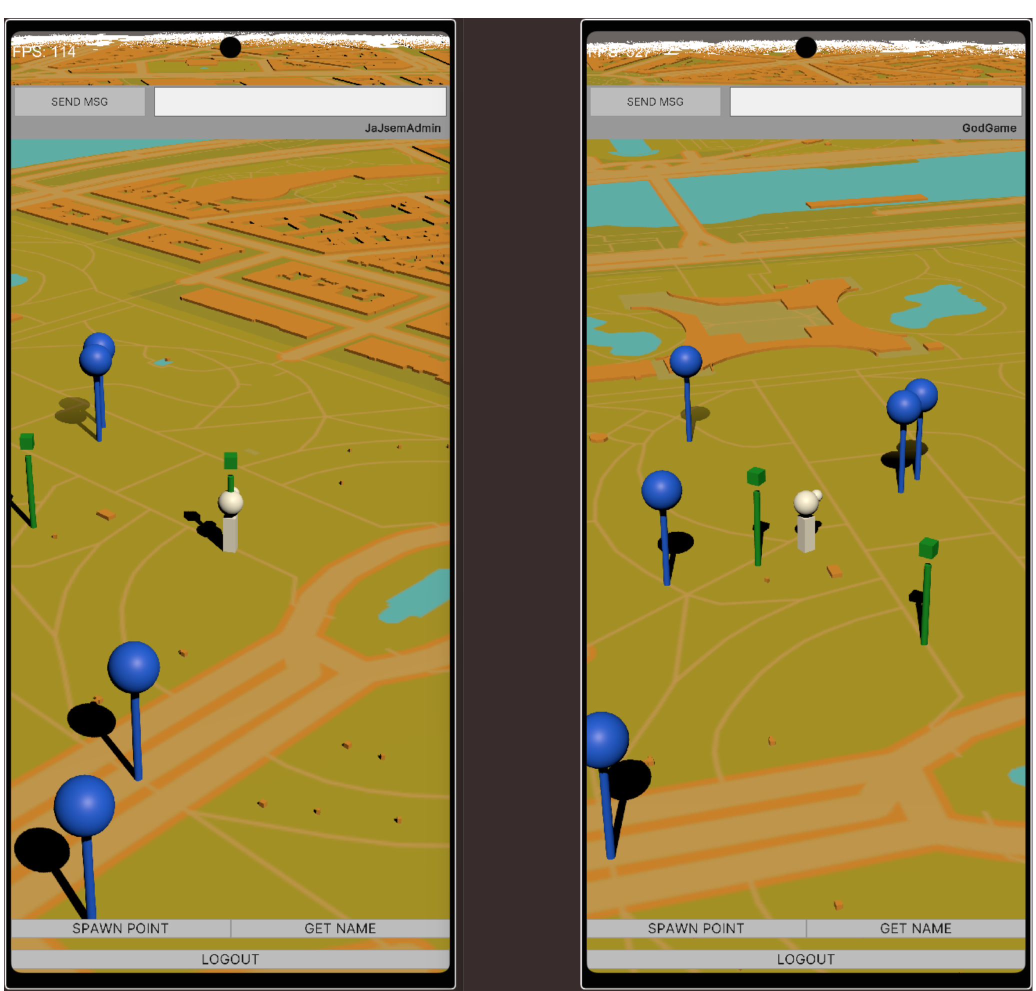
Celý proces získávání a ukládání dat vychází z předchozích sekcí, které hovoří o toku dat celým systémem. Zároveň je velmi důležitý způsob ukládání dat, aby bylo efektivní tato data získávat dostatečně rychle.

Na obrázku 5.18 lze vidět vykreslené body mapy, které uživatel vytvořil. Na dalším obrázku 5.19 lze vidět náhled dvou hráčů, kteří jsou v těsné blízkosti. Na tomto obrázku lze vidět vykreslené body sdílené mezi oběma hráči, kteří je mají oba ve vykreslovací vzdálenosti. Tyto body jsou však modré. Jsou zde i body, které jsou označeny zeleně. Tyto body značí lokaci ostatních hráčů ve vykreslovací oblasti jednotlivých hráčů.

Komunikace se serverem probíhá na principu změny lokace. Pokud hráč (uživatel) změní lokaci a ujde z aktuální vykreslovací vzdálenosti, tak se klientská aplikace začne dotazovat serveru na aktualizaci dat v okolí. Pokud však jsou data stále stejná, nedochází k aktualizaci. V takovém případě je aktualizace prováděna v pravidelných časových intervalech. Pokud k aktualizaci dojde, tak server zároveň vygeneruje kontrolní součet těchto dat, aby bylo možné pro klientskou aplikaci při další požadavku upřesnit hledaná data, která se snaží nahradit.



Obrázek 5.18: Ukázka z aplikace pro synchronizaci dat z databáze mezi klienty.



Obrázek 5.19: Ukázka z aplikace pro synchronizaci dat z databáze mezi klienty.

5.6 Implementované funkce a vlastnosti systému

Finální verze této práce nabízí mnoho funkcí. Pokusím se v bodech vypsat všechny možné funkce a vlastnosti, kterých systém nabývá.

Funkce systému

- Dva různé přístupy komunikace: REST API a soketové rozhraní.
  - Pro každé toto komunikační rozhraní je vyhrazena zvláštní samostatná serverová aplikace, která tuto komunikační službu poskytuje.
  - Soketové rozhraní implementované pomocí SignalR.
  - REST API nabízející uživatelskou autorizaci a jiné získávání dat, např. vytváření testovacích dat.
  - Soketová komunikace nabízí přihlášení na základě tokenů.
- Projektové Clean Architecture rozdělení datových entit, doménových entit a jejich mapování pro perzistentní datové uložiště.
- Testovací prostředí Swagger pro REST API.
- CQRS abstrakce pro podporu vzoru Mediatora pro možné rozdělení projektů a jejich modularizace.
- Lokalizace pro podporu více jazyků.
- Implementace vzoru Event Sourcing jako nadstavba databázového uložiště systému.
  - Uložiště dat může být snadno změněno díky sofistikovanému abstraktnímu rozhraní.
- Uživatelská autorizace pomocí ASP.NET Core Identity.
- Správa uživatelů pomocí ASP.NET Core Identity.
- ASP.NET Core MVC Razor komponenty pro testování s výpomocí Sweagger na straně API serveru.
- Vlastní poskytování a generování JWT tokenů pro autorizaci pomocí soketové a možné jiné komunikace.
- Specification pattern nabízející rozhraní pro zachování principů DRY.

- Podpora ukládání geolokačních dat do perzistentního uložiště Postgres.
- Ukládání indexovaných dat pomocí PostGIS.
- Data jsou vyhledávána na základě geohashů.
- Vyhledávání mapových dat v okruhu kolem uživatele nebo jiných možných bodů na přesnost deseti metrů.
- Připojení k Mapbox API pro získávání geolokačních dat.
- Vykreslovací vzdálenost na základě pohybu a úhlu kamery.
- Pozice uživatelského zařízení je trasována, v jeho okruhu jsou renderována příslušná data.
- Podpora získávání aktuální lokace zařízení pomocí Android API (24).
- Automatizace buildu klientské aplikace pro tvorbu závislostí.
- Dynamická komunikace se serverem pomocí soketové komunikace.
- Komunikace se serverovým REST API a zároveň i API pro získávání mapových dat z Mapbox.
- Speciální nastavitelné výseče mapy pomocí Shadergraph.
- Použití herního enginu Unity s možnostmi 3D grafiky.

Vlastnosti systému

- Docker podpora. Celý systém (s vyjímkou klientský aplikací samozřejmě) lze spustit ve vlastní síti v rámci Docker kontejnerů.
- Velký důraz byl kladen na testování, proto architektonické rozhodování systému bylo zásadně tvořené pro snadnou testovatelnost a mock objektů. Testy jsou snadno vytvořitelné a udržovatelné pro každou část systému.
- Podpora DDD pomocí agregátů, doménových a datových modelů.
- Modularita aplikace pro možný vývoj Micro-Services.
- Loose Coupling, kde komponenty systému lze snadno nahradit bez zásahu do jiných částí systému.
  - Například databázový systém.

- Systém dokáže v jeden okamžik obsloužit tisíce souběžných uživatelských požadavků. Robustnost je důležitá role tohoto systému.
- Systém je vyvíjen s podporou rozšiřitelnosti do cloudových služeb jako je Azure a podporou Redis backplane pro škálovatelnost do šířky.
- Systém nabízí abstraktní kostru, která umožňuje velmi snadnou implementaci logiky uživatelských funkcí a tím i snadnou rozšiřitelnost systému.
- Testovací scénáře pohybu v rámci editoru Unity.
- Tok mapových dat lze upravit na míru systému.
- Synchronizace se serverem počítá s možnými výpadky sítě.
  - Při výpadku má uživatel stanovené časové okno pro znovu připojení.
  - S uživatelem je v systému dle toho náležitě manipulováno.

Vlastností systému z pohledu uživatele

- Systém nabízí uživatelský systém.
- Vykreslení mapových dat ve 3D zobrazení aplikace.
  - Data jsou z API mapována do herních objektů Unity, které vytvářejí dlaždice pro tvorbu mapové plochy.
- Jednoduché rozhraní pro interakci s aplikací.
  - Uživatelské rozhraní přihlášení.
  - Základní rozhraní aplikace: generování nových vlastních bodů mapy a uživatelská část.
- Animace a orientace dle pohybu po mapě.
- Ovládání a rotace kamery na 3D mapě.
- Uživatelé mohou ve svém okolí vytvářet nové lokační body sdílené mezi všemi uživateli systému.
- Zobrazení lokace ostatních uživatelů v dosahu.
- Uživatelé mohou zasílat zprávy ostatním uživatelům v dosahu.

Návrh rozšíření systému

6

Tato kapitola popisuje několik možných rozšíření systému. Cílem bylo sestavit systém pro trasování uživatelů a demonstrovat funkční celek, který dokáže trasovat jejich zařízení. Je zde mnoho funkcí, které mohou být uvažovány dále pro implementaci. Zejména, pokud vezmeme v úvahu dlouhodobý směr tématu práce, který popisuji v kapitole 2.

Aktuální systém nabízí robustní řešení pro tvorbu různých typů aplikací založených na trasování zařízení svých uživatelů. Systém je připravený především pro použití v herním průmyslu, a to hlavně z důvodu použití Unity jako technologie pro vývoj klientské aplikace.

Rozšíření uživatelské služby

Systém nabízí mnoho různých modulů, kterými je možné systém rozšířit. Jedním takovým modulem může být vytvoření produkčního rozhraní uživatele s dalšími funkcemi. Například ve formě seznamu přátel a možnosti rozhraní pro lepší výměnu textových zpráv mezi sebou. To navazuje na další modul založený na původní myšlence výzev, pro kterou je tato práce směřována. Vytvoření možnosti vyzvat jiného hráče na výzvu a vytvoření prostředí, kde uživatelé tyto výzvy mohou spravovat v grafickém prostředí.

Dalším chtěným rozšířením tohoto systému by byla možnost vidět okolní hráče, nikoli pomocí zobrazení na mapě, ale pomocí uživatelského rozhraní. Kdyby byl uživatel v lokalitě s vysokou herní aktivitou, potom by pro něj nemuselo být zcela přehledné vykreslovat všechny objekty na mapě.

Každý uživatel by také měl mít přístupný svůj vlastní zpětný záznam úkolů, které splnil. Toto se zejména odráží na způsobu postupu, kde každý hráč bude reprezentován pomocí svého vytvořeného avatara. Tohoto avatara bude moci dále vylepšovat

tím, jak bude hrou postupovat a vyhrávat jednotlivé výzvy ostatních hráčů.

Důležitou částí bude vytvoření administračního a moderátorského prostředí uživatelů, tzn. vytvoření speciálních rolí například pro znepřístupnění přístupu po stanovenou dobu, případně zamezení pouze textové komunikace apod.

Zajímavým a dalším možným rozšířením může být například zpětné sledování pohybu uživatele. Pomocí toho by bylo pro uživatele možné zjistit, zda se setkal v minulosti se svými přáteli či nikoli. Do systému je možné jednoduše implementovat mnoho podobných rozšíření jako je toto. Rozhodnutí o jejich implementaci závisí na požadavcích z pohledu nabízené služby.

Technické rozšíření

Po technické stránce je zde také několik možností, jak systém dále rozšiřovat. Pro budoucí vývoj je pravděpodobné využití No-SQL databází, které je možné využít díky navržené architektuře. Tyto databázové systémy nabízejí trochu odlišný přístup než relační. Pro jejich implementaci však není nutné řešit architektonický návrh, jelikož takový přístup systém již podporuje. Nicméně je nutné vytvořit vstupy a abstraktní infrastrukturu pro práci s takovými systémy.

Důležitým bodem je optimalizace, kterou bych rád dále prováděl zejména na straně klientské aplikace. Prakticky se jedná o principy, které během vývoje a následně produkce stále trvají. Zde se nachází možná zlepšení na straně efektivnějšího použití paměti cache. Nynější implementace do cache paměti ukládá data z mapového API, ale ještě nepracuje se samotnými daty, které generuje přímo uživatel. Synchronizace a šetření přenosu dat je tak na jednom z hlavních bodů priorit pro další rozšiřování a zlepšování systému.

Neposledním bodem je samotné testování. Architektura systému vytváří dokonalé prostředí pro unit testing, zejména pro možnost testování pomocí mock objektů. Bylo by vhodné pokrýt všechny kritické části systému, které operují s uživatelskými daty. Jelikož se jedná o velmi obecnou aplikaci a systém je z převážné části sestaven z abstraktních struktur, tak zde kromě samotné malé prezentační části není takové množství kódu, kde se s těmito daty manipuluje.

Na základě vyčtených možných rozšíření je možné vidět, že se z převážné většiny jedná o rozšíření, které směřují k naplnění uživatelské služby. Po technické stránce je systém prakticky kompletní.

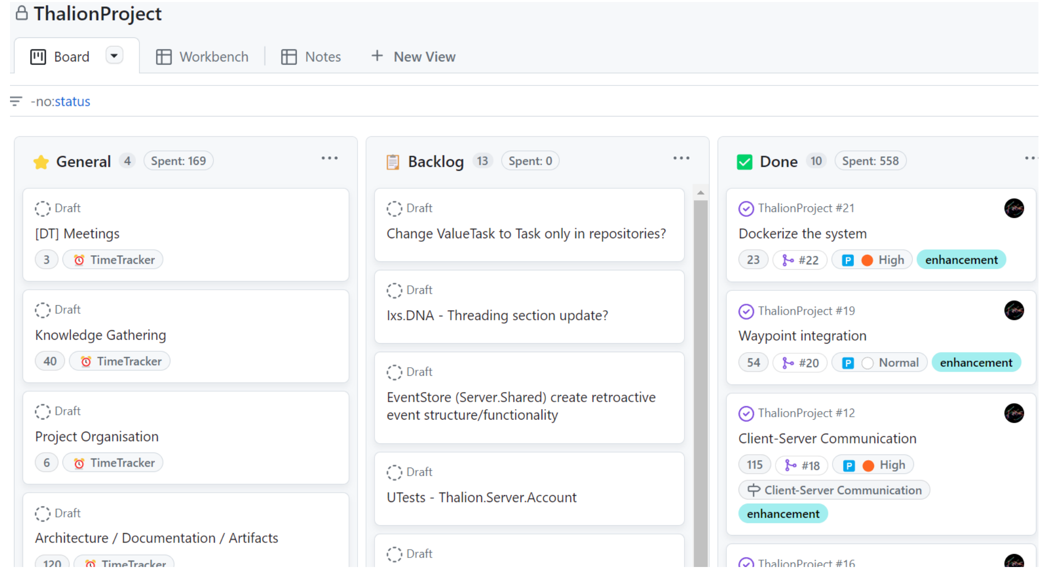
Organizace projektu & Testování

7

Tato kapitola popisuje organizaci projektu, jak byl celý projekt plánován, použití technologie Docker pro celý systém a samotné testování funkcionalit.

7.1 Plánování

Navrhovaný systém je složitější na implementaci, jak je možná zjevné z předchozích kapitol. Tento fakt jsem musel vzít v úvahu při samotném plánování práce. V dnešní době nám mnoho ulehčuje technologie Git, která pomáhá verzovat zdrojové soubory. Nicméně je mnoho dalšího, co dokáže v tomto procesu plánování pomoci.



Obrázek 7.1: Náhled do trasovacího rozhraní úkolů této práce.

Není nutné zbytečně složitého plánovacího systému, jelikož ve vývoji pracuji sám. Kdyby byl tento proces příliš složitý, tak by to mohlo mít opačný efekt nebo při nejmenším jen značně zpomalovat vývoj. Proto je vhodné vytvořit jen nezbytně nutná opatření, která zjednoduší tento vývoj. Git nabízí funkcionalitu možnou vytvářet větve různých verzí programu. Díky tomu dokáži vyvíjet a třídit vyvíjenou funkcionalitu separátně od funkčního celku a případně se vrátit. Pokud však nepoužívám žádný speciální nástroj, tak si tyto větve musím vytvářet ručně. To s sebou nese možná rizika, a to zejména z důvodu pojmenování těchto větví, které jsou má zodpovědnost. Jsou zde nástroje, kterým se také říká „issue tracker". Pomocí takového nástroje dokáži spravovat úkoly na jednom místě a zautomatizovat tvorbu nových úkolů (issues) a také je uzavírat a spojovat do jiných větví. Náhled takového trasování úkolů lze vidět na obrázku 7.1, který zobrazuje také aktuální zpracovávanou práci ke konci vývoje. Je možné si také všimnout celkového stráveného času udávaného v čistých pracovních hodinách, kde jeho hodnota činí 727 hodin v době zaznamenání.

7.2 Dockerizace

Nasazení celého systému nelze předpokládat, že bude prostřednictvím IDE[1], ale spíše pomocí sofistikovaného řešení, které umožní spouštět systém bez nutných závislostí na prostředí. Pro takové řešení lze využít službu Docker[231], která námi definové služby dokáže hostovat jako tzv. „kontejnery".

Vytvoření tzv. „Dockerfiles" nebylo zcela jednoduché, protože je zde velké množství závislostí pro spuštění. Do budoucna jsou tyto závislosti zamýšleny jako balíky pro import pomocí správce balíků Nuget, ale to by v tento moment vyžadovalo vlastní server. Pro nynější účely a efektivnější vývoj jsou tyto závislosti vedeny jen jako projekty referencované na sebe.

Bylo nutné vytvořit celkem 3 kontejnery, kde dva tvořili serverové aplikace pro REST API a Engine soketové komunikace. Třetím pak byla samotná databáze používaná v systému pro uchovávání dat.

Zdrojový kód 7.1: Ukázka health-check pro databázový kontejner.

```
1 healthcheck:
2   test: ["CMD-SHELL", "pg_isready -U postgres"]
3   interval: 5s
4   timeout: 5s
5   retries: 5
```

[1] Integrated Development Environment

Pro sestavení jsem následně použil docker-compose, který vytvoří infrastrukturu na základě vytvořených obrazů jednotlivých aplikací. Důležitou věcí bylo správné nastavení portů a jejich otevření do externího prostředí docker sítě. Zároveň tak bylo důležité ještě vytvořit „health-check" pro databázový kontejner, z důvodu nutnosti informace o možném dalším spuštění jednotlivých serverových aplikací, které jsou na tomto kontejneru závislé. Tato definice je provede v docker-compose souboru, viz ukázka 7.1.

7.3 Testování

Testování je nedílnou součástí dnes již snad každého systému. Během vývoje byl jeden z hlavních cílů návrhu takový, aby umožňoval snadnou implementaci jednotkových testů a tím i snadnou testovatelnost funkcionalit celého systému.

Zejména „mockování" je důležitou praktikou pro testování pomocí jednotkových testů. Celý projekt je implementován pomocí principů Dependency Injection, která z vlastní definice výrazně napomáhá ke snadnější výměněně různých objektů, které jsou do konkrétního objektu vkládány pomocí konstruktorových parametrů.

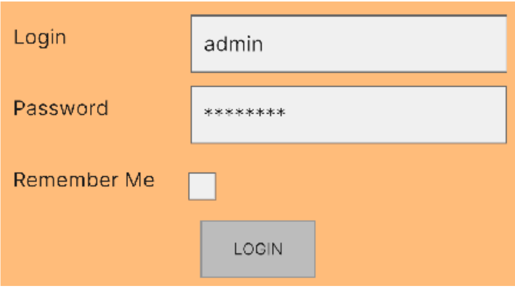
Nejedná se však o jedinou tuto vlastnost, která by udělala mé testování jednodušší, ale v sekci 5.1 hovořím o různých způsobech implementace a kombinaci architektur v rámci serverové části systému. Způsob implementace, který jsem následně zvolil výrazně usnadňuje testování zejména ze strany perzistentního uložiště. Mezitím, co jiné aplikace používají kombinaci doménového a datového modelu (viz zmíněná kapitola), tak v mém případě je to jinak. Snažím se tuto implementaci rozdělit. Dokáži tak držet oddělené implementační primitiva systému pro doménovou logiku a datové modely s implementací konkrétního databázového systému.

Tato architektura je díky tomuto řešení velice členitá. Pokud bych stál o to nahradit pouze samotnou databázi jinou, tak robustní abstraktní struktura (definovaná v téže sekci) nabízí jednotné rozhraní pro práci s daty na úrovni přechodu datové a doménové vrstvy architektury.

Zaměřím-li se dále na klientskou část, tak zde je testování o něco složitější. Je to zejména z důvodu, že pracuji s Unity, které samozřejmě má možnosti testování pro mechaniky vytvářené v editoru, ale není již tak zcela přímočaré jako tomu je u jednotkových testů.

Nechtěl jsem však ignorovat možná testování, na které může v budoucnu dojít, ale zároveň jsem nechtěl vytvořit systém, který by nenabízel žádné testy samotné

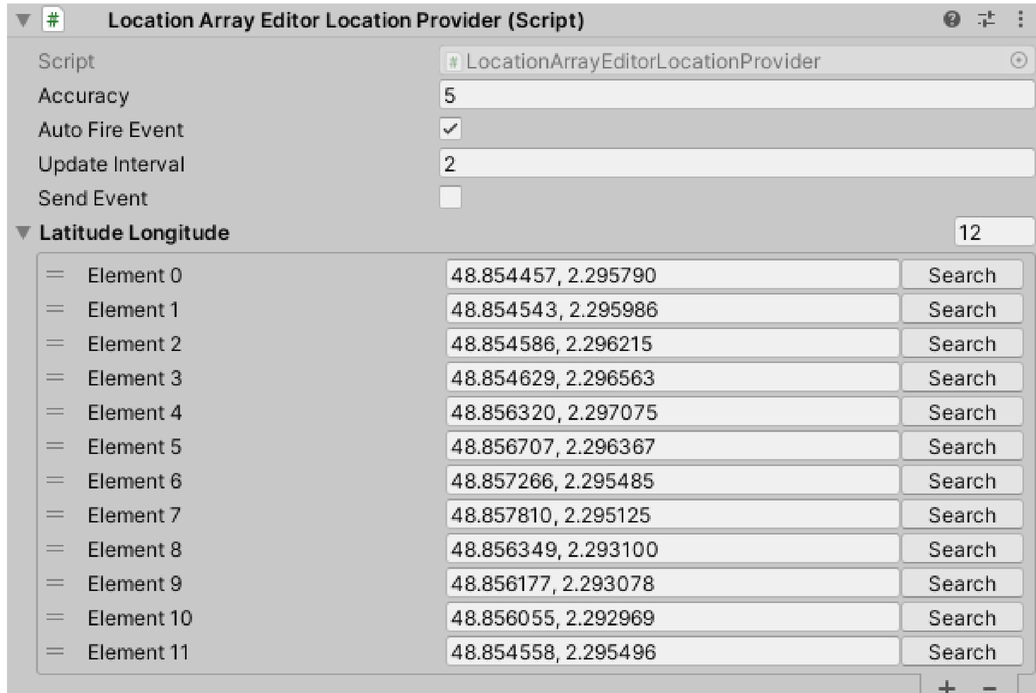
aplikace. Vytvořil jsem tedy testovací scénáře, které lze v rámci editoru Unity přehrávat a interaktivně je vyzkoušet. Je zde několik možností, jak takový testovací scénář vytvořit. Každý z nich však začíná tak, že je nutné, aby se uživatel přihlásil do systému. Pro tyto případy je zde vytvořeno několik testovacích uživatelů s různým oprávněním. Pro jednoduchost je zde uveden pouze jeden a to `admin` (viz obr. 7.2). Pokud je uživatel úspěšně přihlášen, tak je přepnut do scény s mapou. Samotný testovací scénář započne automaticky. Uživatel může využít tohoto scénáře a vkládat do mapy vlastní body a testovat správnost načítání dat v závislosti na vzdálenosti a výseči.



Obrázek 7.2: Přihlášení do systémuv klientské aplikaci.

Testovací scénáře lze vytvářet několika způsoby. Existují dva skripty, které lze ovládat pomocí Isnpector menu v rámci Unity editoru. Oba dva způsoby lze vidět na obrázcích 7.3 a 7.4. V prvním případě lze vidět definici vytyčených bodů cesty, kterou má scénář procházet. V druhém případě následně lze přiložit soubor s možným podrobnějším a složitějším zadáním pro daný testovací scénář.

V rámci tohoto testování jsem se především zaměřil na testování v odlehlých lokalitách, které mohou být méně zmapované a také na oblasti v blízkosti severního pólu, kde může být přesnost snížena. Takovými lokalitami jsou například Helsinki nebo Reykjavik s porovnáním s Paříží. Pro jednoduchost lze v rámci uživatelského rozhraní Unity vyhledávat body přímo na mapě.



Obrázek 7.3: Ukázka možného vstupu pro testovací scénář.

Editor Location Provider Location Log (Script)	
Script	EditorLocationProviderLocationLog
Accuracy	1
Auto Fire Event	✓
Update Interval	0.1
Send Event	
Location Log File	Helsinki

Obrázek 7.4: Ukázka možného vstupu pro testovací scénář.

Dalším druhem testování bylo zátěžové testování na robustnost serveru. Nastavit správné podmínky tohoto testování může být obtížné a nelze v tuto dobu vědět, na jakých strojích doopravdy serverové aplikace poběží. Testování jsem prováděl na svém osobním počítači.

Parametry tohoto stroje jsou následující: Operační systém Windows 11, Procesor Intel i7-1165G7, Grafická karta NVIDIA GeForce MX330, Paměť 16 GB 3200 MHz.

Servery jsem vystavil zátěži spočívající v různé velikosti paralelně volaných požadavků, které server musel obsloužit. Na obrázku 7.5 lze vidět náhled z jednoho z výsledků těchto zátěžových testů. Z tohoto výsledku lze vypozorovat latenci jednotlivých požadavků, které server zpracoval. Jedná se o zátěž, kdy server musel čelit zpracování tisíce požadavků v jeden časový moment. V rámci testování jsem se pokusil dojít až na samotnou hranici, kdy server přestane stíhat zpracovávat požadavky a zpracování požadavku bude zamítnuto. Tuto hranici jsem objevil při překročení deseti tisíc paralelně vyslaných požadavků. Avšak stále velké množství požadavků bylo zpracováno s odezvou v rámci jednotek sekund. Zpracovávání bylo limitováno dostupným hardwarem. Pro testování jsem použil Apache JMeter[23c].

Jeden z hlavních bodů, kterého jsem chtěl při budování tohoto systému docílit, tak je možnost škálovatelnosti celého systému. Vyvíjel jsem systém tak, aby bylo možné toto škálování zařídit pomocí Redis backplane[23au] nebo vývojem v Azure[23au].

Sample #	Start Time	Thread Name	...	Sample Time(ms)	Status
966	22:30:02.404	Thread Group 1-778	...	3147	✓
967	22:30:02.066	Thread Group 1-446	...	3485	✓
968	22:30:02.482	Thread Group 1-859	...	3070	✓
969	22:30:02.414	Thread Group 1-774	...	3138	✓
970	22:30:02.556	Thread Group 1-920	...	2996	✓
971	22:30:02.385	Thread Group 1-763	...	3167	✓
972	22:30:02.462	Thread Group 1-834	...	3090	✓
973	22:30:02.459	Thread Group 1-824	...	3093	✓
974	22:30:01.889	Thread Group 1-259	...	3663	✓
975	22:30:01.874	Thread Group 1-253	...	3678	✓
976	22:30:01.892	Thread Group 1-261	...	3661	✓
977	22:30:02.457	Thread Group 1-830	...	3096	✓
978	22:30:02.510	Thread Group 1-885	...	3043	✓
979	22:30:02.467	Thread Group 1-808	...	3086	✓
980	22:30:02.482	Thread Group 1-862	...	3071	✓

Obrázek 7.5: Ukázka výpisu ze zátěžových testů.

Závěr 8

Cílem této diplomové práce bylo prozkoumat možné způsoby trasování, analyzovat trh takových aplikací, navrhnout a implementovat vlastní řešení pro mobilní zařízení se systémem Android, které bude umožňovat práci s geolokačními daty a tím tedy trasování těchto zařízení.

V první řadě bylo nutné, abych vytvořil návrh, který má smysl a byl by nějakým způsobem konkurence schopný s ostatními návrhy aplikací, se kterými jej budu porovnávat. Nejednalo se o lehký úkol a také tato analýza trvala dlouhou dobu, než jsem se dostal k samotné realizaci. Během tohoto vývoje jsem analyzoval jednotlivé typy aplikací, které měly společné nebo podobné vlastnosti jako můj navrhovaný systém.

Během další fáze příprav a vývoje tohoto systému bylo nutné vybrat vhodné technologie, kterými tento systém realizovat.

Po výběru technologií byl nutný návrh a další analýzy způsobu implementace. Navrhnutý systém, který vznikl musí být schopen dlouhodobé údržby a robustnosti a to především pak pro správu velkého množství uživatelů a škálovatelnosti nejen do výšky, ale především do šířky. Bylo tedy nutné detailně projít různé návrhové vzory a architektonické způsoby. Z vlastní analýzy jsem nedošel k žádné architektuře, která by byla „dokonalá". Z toho důvodu bylo potřeba složitějších rozhodnutí, které jsou v práci popsány.

Po úspěšném návrhu, vhodným pro popisovaný systém, jsem započal implementaci, kterou zde v této práci popisuji. Implementace je rozdělena do hlavních dvou částí. První tvoří serverová část a druhou klientská aplikace a zároveň tak zobrazení map, na kterých lze pozorovat trasovaná data. Samotná implementace mě provázela nemálo nástrahami, kterým jsem musel čelit a to především ze strany použití rozhraní, pomocí kterého jsem získával mapová data. Používaná technologie obsahovala chyby, které mi znemožňovaly požadované implementace a byl jsem nucen

je řešit. Výsledkem nebylo však pouhé vyřešení, ale také nápomoc při dalším vývoji této technologie ve formě komunikace s vývojáři a ostatními přispívajícími v jejich online open-source komunitě.

Výsledkem této práce je pak funkční a stabilní serverový systém nabízející sofistikované a robustní architektonické řešení pro systémy velkého rozsahu a jeho mobilní část. Aplikace nabízí rozhraní pro trasování uživatelů v blízkém okolí a komunikaci s nimi. Aplikace je zároveň vytvořena pomocí rozhraní pro tvorbu vysoce upravitelných 3D map. Tato práce přinesla funkční systém, ale také možný vzor po stránce práce s mapami, trasováním dat uživatelů, správou geolokačních dat a v neposlední řadě také vzor architektonického návrhu. Celý systém dohromady vytváří prostředí pro velmi jednoduchou tvorbu obsahu různých aplikací s podobnými funkcemi.

Instalace & Uživatelská příručka

Instalace

Všechny soubory potřebné pro spuštění systému lze nalézt ve složce `build`, která se nachází v kořenové složce `Aplikace_a_knihovny` v přiložené adresářové struktuře této práce.

Zde se nachází soubor `docker_build.bat`, který spouští vytváření nových Docker obrazů (images) systému, které jsou bezprostředně nutné pro chod systému. Pro veškeré operace s dockerem byl pro vývoj použit Docker Engine verze v23.0.5, a proto bych doporučil použít tuto verzi nebo novější.

Po úspěšném vytvoření těchto obrazů lze spustit soubor `docker_compose.bat`, který provede samotnou kontejnerizaci (vytvoří Docker kontejnery) na základě vytvořených obrazů. Aby toto bylo však možné, je nutné mít naistalovaný SSL certifikát pro podporu HTTPS. Pro případnou instalaci je nutné provést následující příkazy A.1.

Zdrojový kód A.1: Příkazy pro instalaci SSL certifikátu.

```
1 dotnet dev-certs https
2   -ep %USERPROFILE%\.aspnet\https\aspnetapp.pfx
3   -p 123456789
4
5 dotnet dev-certs https --trust
```

Pro použití těchto příkazů je nutné instalace .NET 6.0 SDK[23b]. První příkaz vygeneruje certifikát na zmíněné cestě (Windows). Příkaz používá parameter `-p`, kterým lze specifikovat heslo tohoto certifikátu. To lze libovolně změnit, ale je nutné adekvátně upravit `docker-compose.yml` soubor. Další příkaz jej přidá mezi ověřené certifikáty na hostovaném systému. Více informací lze nalézt přímo na oficiálních stránkách .NET dokumentace[23k].

V tomto bodě je databáze prázdná a je nutné vytvořit vstupní data pro možnou práci. Serverová API aplikace se spustila na vaší lokální adrese, tj. `127.0.0.1:9721`. Po zadání této adresy a včetně portu by měla být navrácena odpověď „OK!“. Pro inicializaci dat lze přejít na adresu: `127.0.0.1:9721/seed`.

V tento moment je serverová část připravena, aby se k ní začali připojovat klienti. Pro spuštění klientské aplikace lze použít přiložený soubor `apk`, který nainstaluje na vaše mobilní zařízení. Druhou možností je otevřít projekt v Unity editoru, kde lze emulovat mobilní zařízení. Je nutné zmínit, že je minimálně nutná verze Android API je 24, tj. Android verze 7.0 nebo vyšší.

Pro instalaci je nutná verze Unity `2022.1.16f1`. Načtený projekt lze zahájit scénou `MainMenu`. V rámci této scény je přihlašovací formulář. Po úspěšném přihlášení se dostanete do uživatelského prostředí mapy.

Uživatelská příručka

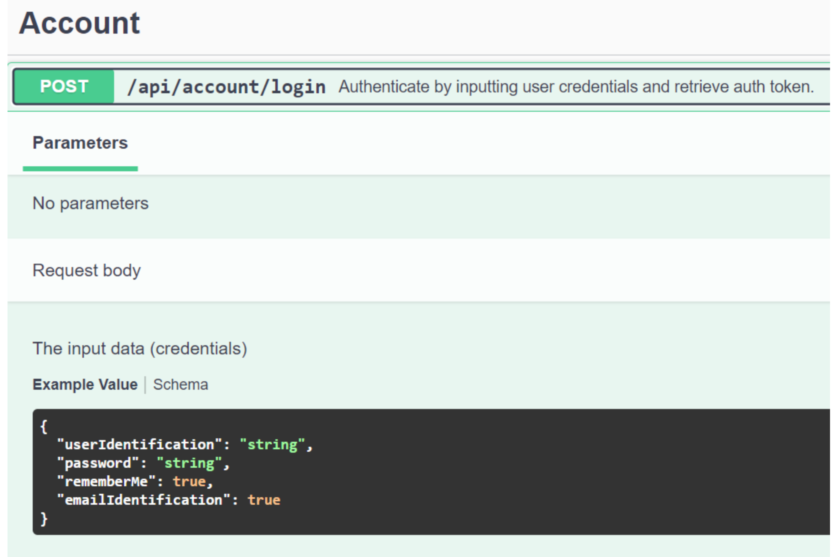
Konfigurace a testování

Server nabízí v development módu rozhraní Swagger (viz obr. A.1), kterým lze snadněji testovat definované REST API v rámci Api serverové aplikace. Dále nabízí možnost vygenerovat vstupní data, pokud je databáze prázdná a to pomocí adresy `/seed`.

V klientské části lze fungovat pomocí definovaných testovacích scénářů, pokud se jedná a produkční verzi aplikace na mobilní zařízení. V ideálním případě, pokud se jedná o Unity editor, tak lze testovací scénáře přizpůsobit a vytvořit si vlastní scénáře.

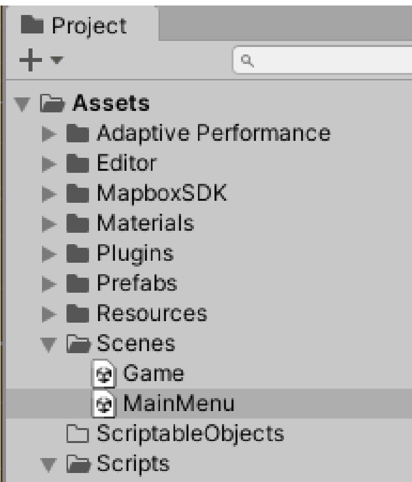
Začneme-li spuštěním v editoru Unity, je nutné spustit `MainMenu` scénu (viz obr. A.2), kde začíná hlavní smyčka celé aplikace a při spuštěném serveru je možné použít přihlašovací formulář a přihlásit se. Po úspěšném přepnutí do scény mapy lze vyvolávat vlastní body dle svého uvážení. V případě editoru Unity lze upravit na herním objektu `LocationProvider` hodnoty jednotlivých skriptů a upravit si tak testovací scénář dle vlastní vůle.

Při startu aplikace má uživatel možnost se přihlásit. Jeden z možných testovacích účtů, které lze použít je `admin` s heslem `Th123/*-`, který je vygenerován při prvním nasazení. Po přihlášení je uživatel přepnut do nastaveného testovacího scénáře.



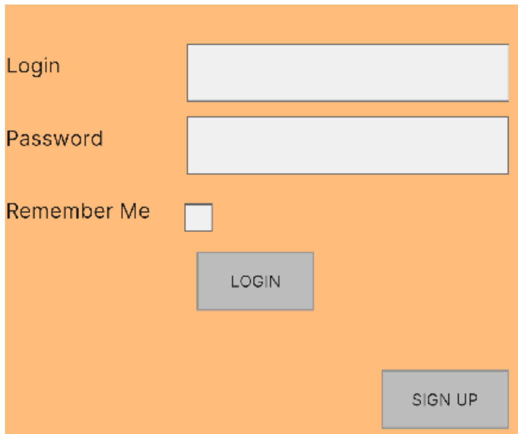
Obrázek A.1: Náhled do rozhraní Swagger.

V případě spuštění aplikace na reálném zařízení je uživateli zobrazena mapa dle jeho reálné polohy GPS. Ovládání lze provádět pomocí následujícího menu A.5 a případně komunikovat s ostatními uživateli v okolí (200 metrů) pomocí menu A.4.

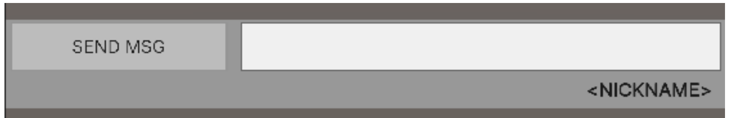


Obrázek A.2: Náhled do hierarchie projektu klientské aplikace.

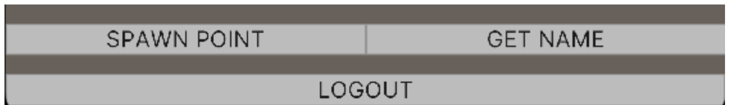
Vypracovaný systém je spíše technologicky zaměřený, a proto zde není mnoho možností interakce. Z toho důvodu bych doporučil projít si implementační část této práce a pochopit samotný kód, případně se jím inspirovat pro vlastní práci či rozšíření této práce.



Obrázek A.3: Hlavní obrazovka s přihlášením uživatele.



Obrázek A.4: Rozhraní pro chat s uživateli v okolí.



Obrázek A.5: Ovládací panel uživatele.

Uživatelský přístup

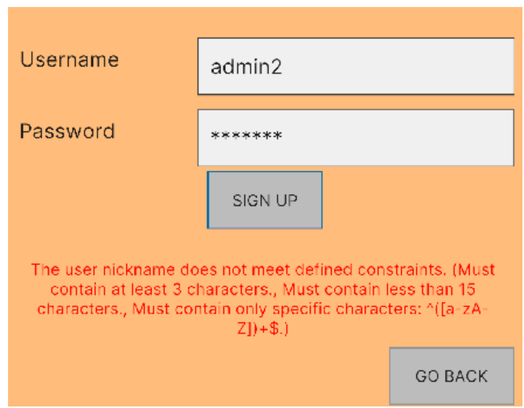
Na úvodní obrazovce Vás aplikace přivítá prostřednictvím přihlašovacího formuláře (viz obr. A.3). V tomto bodě je nutné se přihlásit. Pro toto přihlášení lze použít již předem „seedované“ uživatele. Uživatelé jsou uvedeni v následující tabulce A.1. Také lze vytvořit svého vlastního uživatele na obrazovce registrace. Na tuto obrazovku se lze dostat od přihlašovacího formuláře (viz obr. A.3) pomocí tlačítka „SIGN UP“, které vás po stisknutí přesměruje (viz obr. A.6).

Tabulka A.1: Seznam předem definovaných uživatelů.

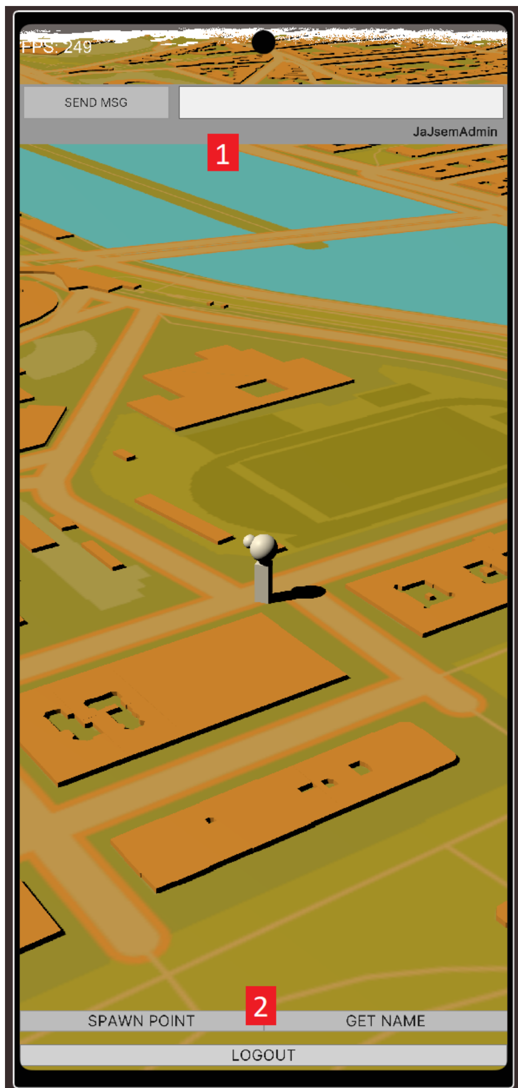
uživatelské jméno	**heslo**
`admin`	`Th123/*-`
`tomas`	`Th123/*-`
`honza`	`Th123/*-`

Po úspěšném přihlášení do systému je uživatel přesměrován na obrazovku samotné mapy, kde se také nacházejí ovládací prvky aplikace. Takovou obrazovku lze vidět na obrázku A.7, kde se nachází zmíněné ovládací prvky s označením.

Pod označením čísla 1 se nachází rozhraní pro zasílání textových zpráv ostatním uživatelům ve vašem blízkém okolí. Zároveň se v tomto rozhraní nachází informace o uživateli, který je přihlášen. Rozhraní pod označení číslem 2 vytváří základní uživatelské rozhraní. Zde je možné se odhlásit z aplikace, vytvářet nové body na mapě nebo aktualizovat své informace o uživateli.



Obrázek A.6: Obrazovka registrace uživatele.



Obrázek A.7: Uživatelské ovládací prvky.

Obsah přiloženého souborového archivu

Popis adresářové struktury a obsahu přiložených dat:

- `Text_prace` - Zdrojové soubory a výsledné PDF této práce.
- `Aplikace_a_knihovny`
  - `build` - Obsahuje nutné soubory pro běh aplikace.
    - `Dockerfiles` - Obsahuje docker soubory pro sestavení obrazu nezbytných částí systému.
    - `docker-compose.yml` - Hlavní docker soubor pro spuštění Docker kontejnerů.
    - `docker_build.bat` - Sekvence příkazů pro vytvoření obrazů dockeru.
    - `docker_compose.bat` - Sekvence příkazů pro spuštění kontejnerů na základě jejich obrazů.
    - `docker_clean.bat` - Sekvence příkazů pro smazání kontejnerů.
  - `Thalion.Client` - Zdrojové soubory klientské části.
  - `Thalion.Server` - Zdrojové soubory serverové části.
  - `Thalion.Shared` - Zdrojové kódy sdílené mezi klientem a serverem.
  - `app.apk` - Soubor APK pro instalaci klietnské aplikaci na zařízení s OS Android.
- `Readme.txt` - Popis souborové struktury.

Bibliografie

[20a] *.NET*. 2020-05-05. Dostupné také z: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet.

[23a] *.NET MVC*. 2023-06-17. Dostupné také z: https://dotnet.microsoft.com/en-us/apps/aspnet/mvc.

[23b] *.NET SDK Download*. 2023-05-31. Dostupné také z: https://dotnet.microsoft.com/en-us/download.

[23c] *Apache JMeter*. 2023-05-31. Dostupné také z: https://jmeter.apache.org/.

[23d] *ArcGIS official website*. 2023-05-10. Dostupné také z: https://www.arcgis.com/index.html.

[23e] *ASP.NET Core SignalR hosting and scaling*. 2023-05-12. Dostupné také z: https://learn.microsoft.com/en-us/aspnet/core/signalr/scale.

[23f] *AutoMapper*. 2023-05-15. Dostupné také z: https://automapper.org/.

[Bas18] BASKIN, Jacob. *PostGIS Performance Showdown: Geometry vs. Geography*. 2018-08-01. Dostupné také z: https://medium.com/coord/postgis-performance-showdown-geometry-vs-geography-ec99967da4f0.

[23g] *Benchmark Games*. 2023-05-10. Dostupné také z: https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[23h] *BEST PRACTICES FOR ORGANIZING YOUR UNITY PROJECT*. 2023-05-15. Dostupné také z: https://unity.com/how-to/organizing-your-project.

[Cat20] CATAMAK, Adem. *Specification pattern*. 2020-09-08. Dostupné také z: https://medium.com/c-sharp-progarmming/specification-design-pattern-c814649be0ef.

[Coh21] COHEN, Jason. *How to Share Your Location in Google Maps*. PC Magazine, 2021-12-06. Dostupné také z: https://www.pcmag.com/how-to/how-to-share-your-location-in-google-maps.

[23i] *CQRS pattern*. 2023-06-17. Dostupné také z: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs.

[20b] *CSS*. 2020-05-05. Dostupné také z: https://www.w3.org/Style/CSS/Overview.en.html.

[Dam21] DAMTOFT, Eric. *Onion vs Clean vs Hexagonal Architecture*. 2021-11-30. Dostupné také z: https://medium.com/@edamtoft/onion-vs-clean-vs-hexagonal-architecture-9ad94a27da91.

[23j] *Dart official webosite*. 2023-05-09. Dostupné také z: https://dart.dev/.

[23k] *Docker Compose Hosting Documentation*. 2023-05-31. Dostupné také z: https://learn.microsoft.com/en-us/aspnet/core/security/docker-compose-https?view=aspnetcore-6.0.

[23l] *Docker official website*. 2023-05-24. Dostupné také z: https://www.docker.com/.

[Dom23] DOMINIK BARTOSZEWSKI Adam Piorkowski, Michal Lupa. *The Comparison of Processing Efficiency of Spatial Data for PostGIS and MongoDB Databases*. 2023-05-10. Dostupné také z: https://edisciplinas.usp.br/pluginfile.php/5530294/mod_resource/content/1/BDAS19_DBAPML_online.pdf.

[Dzi23] DZIUBA, Anna. *Mapbox vs. Google Maps vs. OpenStreetMap APIs: Finding the Perfect Fit for Your Next App*. 2023-04-28. Dostupné také z: https://relevant.software/blog/choosing-a-map-amapbox-google-maps-openstreetmap/.

[20c] *Electron*. 2020-05-05. Dostupné také z: https://www.electronjs.org/.

[20d] *Electron: The best*. 2020-05-05. Dostupné také z: https://www.intertech.com/Blog/why-major-companies-are-using-electron-to-build-cross-platform-apps/.

[23m] *EPSG Geodetic Parameter Dataset*. 2023-05-15. Dostupné také z: https://en.wikipedia.org/wiki/EPSG_Geodetic_Parameter_Dataset.

[Eva03] EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 2003-08-22. Dostupné také z: https://www.amazon.com/-/en/Evans-Eric-ebook-dp-B00794TAUG/dp/B00794TAUG/ref=mt_other.

[Fan23] FANCHI, Christopher. *Flutter vs. React Native: Which is Better in 2023*. 2023-05-08. Dostupné také z: https://backendless.com/flutter-vs-react-native-which-is-better-in-2023.

[23n] *Find My Kids Google Play Store*. 2023-05-07. Dostupné také z: https://play.google.com/store/apps/details?id=org.findmykids.app.

[23o] *Flutter official webosite*. 2023-05-09. Dostupné také z: https://flutter.dev/.

[23p] *FollowMee official website*. 2023-05-06. Dostupné také z: https://www.followmee.com/.

[23q] *Geofences*. 2023-05-11. Dostupné také z: https://developer.android.com/training/location/geofencing.

[23r] *Glympse official website*. 2023-05-07. Dostupné také z: https://glympse.com/.

[23s] *GNSS*. 2023-05-10. Dostupné také z: https://www.gps.gov/systems/gnss/.

[Goo23a] GOOGLE. *Share a map or directions with others*. 2023-05-06. Dostupné také z: https://support.google.com/maps/answer/144361.

[Goo23b] GOOGLE. *Share your real-time location with others*. 2023-05-06. Dostupné také z: https://support.google.com/maps/answer/7326816.

[23t] *Google Maps Gaming Services*. 2023-05-10. Dostupné také z: https://developers.google.com/maps/documentation/gaming.

[23u] *Google Maps pricing*. 2023-05-09. Dostupné také z: https://mapsplatform.google.com/pricing/.

[20e] *HTML*. 2020-05-05. Dostupné také z: https://developer.mozilla.org/en-US/docs/Web/HTML.

[20f] *Chromium*. 2020-05-05. Dostupné také z: https://www.chromium.org/.

[Ibe22] IBER, Dyarlen. *Hexagonal Architecture and Clean Architecture (with examples)*. 2022-08-11. Dostupné také z: https://dev.to/dyarleniber/hexagonal-architecture-and-clean-architecture-with-examples-48oi.

[23v] *Ingress Prime official website*. 2023-05-07. Dostupné také z: https://www.ingress.com/.

[20g] *Ionic*. 2020-05-05. Dostupné také z: https://ionicframework.com/.

[20h] *Java*. 2020-05-05. Dostupné také z: https://www.java.com/en/.

[20i] *Javascript*. 2020-05-05. Dostupné také z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[Kay15] KAY, David. *How to Follow A Public Group / Tag on Glympse*. 2015-08-05. Dostupné také z: https://www.youtube.com/watch?v=A1pxHPMbDWc.

[Kha22] KHAN, Sardar Mudassar Ali. *Onion Architecture In ASP.NET Core 6 Web API*. 2022-07-07. Dostupné také z: https://www.c-sharpcorner.com/article/onion-architecture-in-asp-net-core-6-web-api/.

[Kho18] KHORIKOV, Vladimir. *CQRS vs Specification pattern*. 2018-11-06. Dostupné také z: https://enterprisecraftsmanship.com/posts/cqrs-vs-specification-pattern.

[Kho21] KHORIKOV, Vladimir. *Specification Pattern vs Always-Valid Domain Model*. 2021-07-27. Dostupné také z: https://enterprisecraftsmanship.com/posts/specification-pattern-always-valid-domain-model/.

[20j] *Laravel*. 2020-05-05. Dostupné také z: https://laravel.com.

[Las22] LASO, Javiera. *What is domain-centric architecture?* 2022-03-13. Dostupné také z: https://jlasoc.medium.com/what-is-domain-centric-architecture-e030e609c401.

[23w] *Life360 official website*. 2023-05-06. Dostupné také z: https://www.life360.com/.

[23x] *Location Based Game*. 2023-05-10. Dostupné také z: https://docs.mapbox.com/unity/maps/guides/location-based-games/.

[23y] *Magic Streets official website*. 2023-05-07. Dostupné také z: https://magic-streets.com/.

[23z] *Mapbox changelog*. 2023-05-21. Dostupné také z: https://docs.mapbox.com/mapbox-unity-sdk/docs/05-changelog.html.

[23aa] *Mapbox official website*. 2023-05-10. Dostupné také z: https://www.mapbox.com/.

[22a] *Mapbox Unity SDK (v2.1.1) issue solving*. 2022-10-16. Dostupné také z: hhttps://gist.github.com/Frixs/fa3b1bec1f7626c05c4c3c6ff1475618#gistcomment-4337586.

[23ab] *MapMyRun Google Play*. 2023-05-10. Dostupné také z: https://play.google.com/store/apps/details?id=com.mapmyrun.android2.

[23ac] *MapMyRun official website*. 2023-05-06. Dostupné také z: https://www.mapmyrun.com/.

[23ad] *mLite - lighter version of mSpy*. 2023-05-06. Dostupné také z: https://play.google.com/store/apps/details?id=com.mspy.lite.

[23ae] *MongoDB geoNear function*. 2023-05-10. Dostupné také z: https://www.mongodb.com/docs/manual/reference/operator/aggregation/geoNear/.

[23af] *MongoDB official website*. 2023-05-10. Dostupné také z: https://www.mongodb.com/.

[23ag] *mSpy official website*. 2023-05-06. Dostupné také z: https://www.mspy.com.

[23ah] *mSpy pricing*. 2023-05-10. Dostupné také z: https://www.mspy.com/discount.html.

[23ai] *MUNZEE official website*. 2023-05-07. Dostupné také z: https://www.playmunzee.com/.

[20k] *NodeJS*. 2020-05-05. Dostupné také z: https://nodejs.org/en/.

[23aj] *OpenStreetMaps*. 2023-05-10. Dostupné také z: https://www.openstreetmap.org/.

[23ak] *Orna official website*. 2023-05-07. Dostupné také z: https://playorna.com/.

[Ozk21] OZKAYA, Mehmet. *Event Sourcing Pattern in Microservices Architectures*. 2021-09-08. Dostupné také z: https://medium.com/design-microservices-architecture-with-patterns/event-sourcing-pattern-in-microservices-architectures-e72bf0fc9274.

[20l] *PhP*. 2020-05-05. Dostupné také z: https://www.tutorialspoint.com/php/index.htm.

[23al] *Plain old CLR object*. 2023-05-13. Dostupné také z: https://en.wikipedia.org/wiki/Plain_old_CLR_object.

[23am] *Pokémon GO official website*. 2023-05-07. Dostupné také z: https://pokemongolive.com/.

[23an] *PostGIS official website*. 2023-05-10. Dostupné také z: https://postgis.net/.

[23ao] *PostGIS ST_DWithin function*. 2023-05-10. Dostupné také z: https://postgis.net/docs/ST_DWithin.html.

[23ap] *PostgreSQL official website*. 2023-05-10. Dostupné také z: https://www.postgresql.org/.

[23aq] *React Native official website*. 2023-05-09. Dostupné také z: https://reactnative.dev/.

[20m] *React.js*. 2020-05-05. Dostupné také z: https://reactjs.org/.

[23ar] *Repository pattern in .NET*. 2023-02-21. Dostupné také z: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design.

[23as] *Resources Game official website*. 2023-05-07. Dostupné také z: https://resources-game.ch/en/.

[Rut22] RUTKOWSKI, Rob. *GNSS systems comparison*. 2022-01-25. Dostupné také z: https://blog.bliley.com/the-differences-between-the-5-gnss-satellite-network-constellations.

[23at] *SHADER GRAPH*. 2023-05-21. Dostupné také z: https://unity.com/features/shader-graph.

[23au] *SignalR scale-out*. 2023-05-31. Dostupné také z: https://learn.microsoft.com/en-us/aspnet/core/signalr/redis-backplane?view=aspnetcore-6.0.

[22b] *Socket Programming in The .NET Framework Using C#*. 2022-03-25. Dostupné také z: https://www.section.io/engineering-education/socket-programming-in-the-dotnet-framework-using-csharp/.

[20n] *Symfony*. 2020-05-05. Dostupné také z: https://symfony.com.

[Tho23] THOMPSON, Stephen. *Onion Architecture - Cutting onions, without the tears!* 2023-05-13. Dostupné také z: https://tech.ovoenergy.com/onion-architecture/.

[23av] *Tracking vs. No-Tracking Queries*. 2023-05-15. Dostupné také z: https://learn.microsoft.com/en-us/ef/core/querying/tracking.

[23aw] *Unity official website*. 2023-05-09. Dostupné také z: https://unity.com/ .

[23ax] *UNITY UI TOOLKIT*. 2023-05-20. Dostupné také z: https://unity.com/features/ui-toolkit.

[23ay] *UNIVERSAL RENDER PIPELINE (URP)*. 2023-05-20. Dostupné také z: https://unity.com/srp/universal-render-pipeline.

[Wei22] WEI, Lim How. *How to Fix "Nearby Friends" Not Showing on Facebook*. followchain, 2022-06-02. Dostupné také z: https://www.followchain.org/nearby-friends-not-showing-facebook.

[23az] *What is a REST API?* 2023-05-08. Dostupné také z: https://www.ibm.com/topics/rest-apis.

[20o] *What is the Difference Between GNSS and GPS?* 2020-01-20. Dostupné také z: https://www.everythingrf.com/community/what-is-the-difference-between-gnss-and-gps_58.

[Wil22] WILL FULTON, Jan Bitner. *The 10 best location-based games*. 2022-05-16. Dostupné také z: https://www.digitaltrends.com/gaming/best-location-based-gps-games/.

[ZHA20] ZHANG, JUNHUA. *Beidou: Assessing China's alternative to GPS*. 2020-01-27. Dostupné také z: `https://www.gisreportsonline.com/r/chinese-navigation-system/`.

Seznam obrázků

2.1 Ukázka implementace výseče na mapě v aplikaci mLite [23ad]. 10
2.2 Funkce vykreslování statistik v aplikaci Map My Run[23ab]. 12
2.3 Ukázka definování úkolů v rámci aplikace Find My Kids[23ab]. 13
2.4 Ilustrace rozdílu mezi GPS a GNSS [20o]. 16
2.5 Galileo satelity se srovnáním k GPS. 17
2.6 Ilustrace postupu vývoje BeiDou GNSS [ZHA20]. 18

3.1 Náhled zprostředkování API pro Google Maps[23u]. 26
3.2 Náhled do Mapbox editoru mapového zdroje dat. 27
3.3 Vykreslení mapy pomocí ArcGIS. 28
3.4 Ukázka vyhledávání a sestavování řetězce, resp. geohash. 30
3.5 Ukázka vykreslování 3D map pomocí Mapbox[23x]. 32

4.1 Graficky znázorněné typy architektur jako ukázka k přiloženému textu[Las22]. 34
4.2 Rozdíl mezi vrstvenou a doménově orientovanou architekturou[Las22]. 35
4.3 Rozdíl souborové struktury projektu s architekturou vrstvenou a doménově orientovanou[Las22]. 35
4.4 Příklad Hexagonal Architecture[Las22]. 36
4.5 Příklad Onion Architecture[Tho23]. 37
4.6 Příklad Clean Architecture[Las22]. 38
4.7 Event Sourcing pattern vizualizace [Ozk21]. 40
4.8 Rozdíl CQRS a Specification vzoru [Kho18]. 41
4.9 Grafické znázornění nutného bodu validace v Always-Valid Domain Model vzoru [Kho21]. 42

5.1 Grafický přehled všech řešení v rámci systému včetně projektů, ze kterých se skládají a jejich závislostí. 46
5.2 Struktura řešení `Server.AccountService`. 49
5.3 Projektový návrh řešení `Server.AccountService` včetně závislostí mezi projekty a závislostmi k řešení `Server.Shared`. 50
5.4 Návrh řešení kompozice repository s příkladem na `User` agregátu. . . 60

5.5 Struktura projektu klientské aplikace. 66
5.6 Diagram struktury závislostí celého systému. 67
5.7 Data Flow diagram přenosu dat ve směru klient->server. 69
5.8 Data Flow diagram přenosu dat ve směru server->klient. 70
5.9 Náhled Mapbox map editoru s vlastních výběrem dat. 72
5.10 Náhled Mapbox map editoru s vlastním stylem. 73
5.11 Náhled do editoru Unity s pohledem na mapu vykreslenou pomocí Mapbox SDK. 75
5.12 Používaná mapová postavička pro vývoj v Unity. 76
5.13 Pohled na Shadergraph rozložení zajišťující efekt výseče. 77
5.14 Rozložení prvků grafu pro tvorbu mapové výseče v okolí hráče. 78
5.15 Detailní pohled na graf výseče, zaměřený na část práce s pozicí. 79
5.16 Aplikace výseče na výkres mapy podle pozorovacího úhlu uživatele. . 79
5.17 Pohled uživatele s aplikovanou výsečí v prostředí emulátoru. 80
5.18 Ukázka z aplikace pro synchronizaci dat z databáze mezi klienty. . . . 82
5.19 Ukázka z aplikace pro synchronizaci dat z databáze mezi klienty. . . . 83

7.1 Náhled do trasovacího rozhraní úkolů této práce. 89
7.2 Přihlášení do systémuv klientské aplikaci. 92
7.3 Ukázka možného vstupu pro testovací scénář. 93
7.4 Ukázka možného vstupu pro testovací scénář. 93
7.5 Ukázka výpisu ze zátěžových testů. 94

A.1 Náhled do rozhraní Swagger. 99
A.2 Náhled do hierarchie projektu klientské aplikace. 99
A.3 Hlavní obrazovka s přihlášením uživatele. 100
A.4 Rozhraní pro chat s uživateli v okolí. 100
A.5 Ovládací panel uživatele. 100
A.6 Obrazovka registrace uživatele. 102
A.7 Uživatelské ovládací prvky. 102

Seznam tabulek

5.1 Obsah a význam projektů rozdělených dle DDD. 48

A.1 Seznam předem definovaných uživatelů. 101

Seznam výpisů

5.1 Výpis kódu metody pro integraci služeb do aplikace. 52
5.2 Kód metody konkrétní služby pro integraci do aplikace. 52
5.3 Metoda obsluhjící požadavek z klientské strany pro aktualizaci uživatelských dat. 54
5.4 Ukázka struktury rozhraní v Mediator patternu 55
5.5 Obslužná metoda Handle nacházející se ve třídě z předchozího příkladu - GetUserQueryHandler. 56
5.6 Náhled implementace základu pro doménové repository. 61
5.7 Náhled implementace datového repository. 62
5.8 Rozhraní repository pro User agregát. 62
5.9 Ukázka implementace User agregát repository. 62
5.10 Ukázka implementace konfigurace databázového modelu pro ukládání bodů mapy. 65
5.11 Ukázka implementace metody pro čtení datz z databáze v okruhu stanoveného bodu a nastaveného rádiusu. 65
7.1 Ukázka health-check pro databázový kontejner. 90
A.1 Příkazy pro instalaci SSL certifikátu. 97