FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Backend karetní hry pro OS Android
Student:	Jan Hamal
Vedoucí:	Ing. Miroslav Balík, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem této bakalářské práce je implementovat funkční prototyp backendu karetní hry pro OS Android, jejíž koncept bude navržen v rámci bakalářské práce Vojtěcha Mičky. Mělo by se jednat o sběratelskou karetní hru, ve které si bude hráč moci vytvořit balíček z karet ve své kolekci a poté s ním soupeřit v zápasech jeden na jednoho proti ostatním hráčům. Frontend pro tuto karetní hru je předmětem bakalářské práce Vojtěcha Mičky.

1. Popište navržený koncept karetní hry.
2. Zvolte vhodné technologie pro implementaci funkčního prototypu backendu pro tuto hru.
3. Implementujte funkční prototyp backendu pro tuto hru.
4. Vytvořte možnost hrát tuto hru v režimu hráč proti hráči a v režimu hráč proti robotovi.
5. Vytvořte systém pro tvorbu zápasů mezi hráči této hry (matchmaking).
6. Funkční prototyp backendu vhodně otestujte.
7. Úzce spolupracujte s Vojtěchem Mičkou, který pracuje na konceptu karetní hry a na frontendu pro tuto hru.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 24. ledna 2024 v Praze.

Bakalářská práce

BACKEND KARETNÍ HRY PRO OS ANDROID

Jan Hamal

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Miroslav Balík, Ph.D.
16. května 2024

Odkaz na tuto práci: Hamal Jan. *Backend karetní hry pro OS Android*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chci poděkovat panu Ing. Miroslavu Balíkovi, Ph.D. za pomoc a cenné rady při vedení práce. Kolegovi Vojtěchu Mičkovi děkuji za skvělou spolupráci. Děkuji také své rodině, přátelům a přítelkyni, kteří mi byli po celou dobu práce podporou.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 16. května 2024

Abstrakt

Tato bakalářská práce se zaměřuje na návrh a implementaci funkčního prototypu serverové části (backendu) karetní sběratelské hry. Serverová část hry je jedna ze dvou částí, ze kterých se celý projekt karetní hry skládá. Tato část aplikace zajišťuje zejména vykonávání aplikačních procesů včetně herní logiky a ukládání dat do databáze. Druhou částí projektu je poté klientská část (frontend), která je spouštěna na jednotlivých mobilních zařízeních s operačním systémem Android a slouží jako uživatelské rozhraní pro hráče hry. Tato práce se však nezaměřuje na klientskou část projektu, ale pouze specificky na jeho serverovou část.

V rámci procesu vývoje serverové části projektu je využito služeb platformy Google Firebase. Práce se zabývá specifikací funkčních a nefunkčních požadavků kladených na backendovou část a návrhem databázového modelu určeného pro ukládání dat v databázi Cloud Firestore. Dále se zabývá návrhem a implementací rozhraní pro spouštění funkcionalit podporovaných serverovou částí hry. Toto rozhraní je vystaveno klientské části za použití technologie cloudových funkcí. Nakonec se práce zaměřuje na testování vytvořeného funkčního prototypu.

Klíčová slova karetní hra, Firebase, Cloud Firestore, cloudové funkce, Python, tvorba zápasů, backend, bezserverová architektura

Abstract

This bachelor thesis focuses on the design and implementation of a functional prototype of the server part (backend) of a collectible card game. The server part of the game is one of the two parts that make up the entire card game project. This part of the application mainly provides the execution of application processes including game logic and data storage in the database. The second part of the project is the client part (frontend), which is run on individual mobile devices running the Android operating system and serves as the user interface for the game players. However, this thesis does not focus on the client part of the project, but only specifically on the server part of the project.

In the process of developing the server part of the project, the services of the Google Firebase platform are used. This thesis deals with the specification of functional and non-functional requirements placed on the backend part and designing a database model

intended for storing data in the Cloud Firestore database. It also deals with the design and implementation of the interface for running the functionalities supported by the server side. This interface is exposed to the client part using cloud functions technology. Finally, the thesis focuses on testing the developed functional prototype.

Keywords card game, Firebase, Cloud Firestore, cloud functions, Python, matchmaking, backend, serverless architecture

Seznam zkratek

API	Application Programming Interface
BPMN	Business Process Model and Notation
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
SDK	Software Development Kit
UML	Unified Modeling Language

Úvod

Za poslední dekády se trh s počítačovými hrami rapidně změnil a rozvinul. Dnes je již na trhu nespočet herních titulů různých herních žánrů, ať už pro počítače, mobilní telefony, tablety či nejrůznější herní konzole. Některé z herních titulů jsou placené, jiné jsou ke hraní zcela zdarma nebo jsou v rámci nich placené jen některé části hry.

Fenoménem posledních desítek let v herním odvětví se stal žánr sběratelských karetních her. Hráči v takových hrách se zaměřují na sbírání karet, ze kterých si mohou následně skládat své karetní balíčky. S těmito balíčky karet se mohou hráči poté účastnit různých typů zápasů specifických pro danou hru a porovnávat si své síly s jinými hráči. Původní fyzická forma sběratelských karetních her se začala časem dále rozvíjet v jejich digitální verze [1].

Ve své bakalářské práci se zaměřím na vývoj funkčního prototypu backendu karetní sběratelské hry pro mobilní telefony s operačním systémem Android. V rámci tohoto vývoje budu spolupracovat se studentem FIT ČVUT Vojtěchem Mičkou, který rovněž pracuje na své bakalářské práci, v rámci které se zaměřuje naopak na frontend pro tuto karetní hru. Koncept již zmíněné karetní hry vymyslel Vojtěch Mička a bude v mé bakalářské práci později vysvětlen a popsán. V průběhu své bakalářské práce navrhnu a implementuji funkční prototyp backendu zmíněné karetní hry, vhodně jej otestuji a nasadím na cílovou platformu podle využité technologie.

Kapitola 1
Cíle práce

Cílem celé této práce je zejména vytvořit funkční prototyp serverové části sběratelské karetní hry, na který bude možno napojit klientskou aplikaci, jejíž vývoj je náplní bakalářské práce mého kolegy Vojtěcha Mičky.

Cílem teoretické části práce je popsat koncept navržené karetní hry, který vymyslel student Vojtěch Mička, a také popsat technologie vybrané pro implementaci backendu této karetní hry.

Cílem praktické části práce je zejména navrhnout a implementovat funkční prototyp backendu sběratelské karetní hry. Dílčím cílem v této části bude také specifikovat požadavky, které jsou na vytvářenou aplikaci kladeny. Backend hry bude umožňovat hrát hru pro dva reálné hráče a také hrát hru pro jednoho reálného hráče s robotem. Navíc bude podporovat tvorbu herních zápasů v reálném čase (matchmaking). Bude potřeba, aby backend vhodně komunikoval s frontendem karetní hry, jehož vývoj je náplní bakalářské práce studenta Vojtěcha Mičky. Dalším cílem praktické části je vytvořený prototyp backendu vhodně otestovat a nasadit na cílovou platformu.

Kapitola 2
Popis konceptu karetní hry

Tato kapitola se zabývá stručným popisem konceptu karetní hry, který navrhl kolega Vojtěch Mička. Celý podrobný popis je k nalezení v jeho bakalářské práci s názvem *Frontend karetní hry pro OS Android*.

V této hře mohou hráči skládat ze svých karet své karetní balíčky po devíti kartách, se kterými se mohou poté účastnit zápasů s jinými hráči. Každá karta obsahuje jeden ze tří herních živlů, sílu útoku, pozitivní schopnost a negativní schopnost. Zápasů se ve hře mohou účastnit pouze dva hráči. Začíná se okamžikem, kdy mají oba hráči stejný počet životů a tři náhodné karty ze svého balíčku drží v ruce. V průběhu zápasu může nastat nanejvýš 9 kol. V každém jeho kole musí odehrát oba hráči jednu svou kartu z ruky a ze zbývajících karet svého balíčku si dobírají na začátku dalších kol vždy jednu náhodnou kartu. Jakmile oba hráči v daném kole odehrají karty, pak je kolo vyhodnoceno. Dané kolo vyhraje ten, jehož živel na odehrané kartě vyhrává proti živlu karty druhého hráče. Podle pravidel jsou poté uplatněny síly útoků a efekty jednotlivých karet. Toto po vyhodnocení každého kola může jednoho z hráčů zranit natolik, že už nebude mít žádné své životy a zápas skončí ve prospěch druhého hráče s více životy. Pokud nikdo během 9 kol v zápase nezemře, pak vyhraje ten hráč, který má více životů, případně hra skončí remízou. [2]

Kapitola 3
Analýza

V této kapitole jsou specifikovány funkční a nefunkční požadavky kladené na aplikaci a také popsány technologie vybrané pro implementaci.

3.1 Funkční a nefunkční požadavky

Následující podsekce specifikují výčty a popisy funkčních a nefunkčních požadavků, které jsou na návrh a implementaci backendové části aplikace kladeny. Na tvorbě požadavků jsem spolupracoval s kolegou Vojtěchem Mičkou.

3.1.1 Funkční požadavky

Výčet funkčních požadavků na aplikaci je následující:

1. **Založení nového uživatelského účtu** – backend aplikace bude podporovat možnost pro tvorbu nového uživatele v systému.
2. **Hra pro dva hráče** – karetní hru bude možné hrát v režimu jeden hráč proti jednomu hráči po síti.
3. **Hra hráče proti robotovi** – karetní hru bude možné hrát v režimu jeden hráč proti jednomu robotovi po síti, pokud nebude dostupný dostatek reálných hráčů v daném čase.
4. **Tvorba zápasů (matchmaking)** – aplikace bude zajišťovat automatickou tvorbu zápasu pro dva hráče, nebo pro jednoho hráče a robota.
5. **Zařazení hráče do fronty pro tvorbu zápasů** – pokud bude chtít uživatel hrát zápas, bude zařazen do čekací fronty, ve které může být spárován do zápasu s jiným hráčem. Pokud bude ve frontě čekat déle než daný limit a nebude s nikým spárován do zápasu, bude vytvořen zápas pro něj a pro robota.
6. **Odebrání hráče z fronty pro tvorbu zápasů** – pokud nebyl uživatel dosud s nikým spárován do zápasu, bude moci odejít z fronty pro tvorbu zápasů.

7. **Potvrzení zahájení zápasu** – možnost klienta odeslat potvrzení pro zahájení zápasu, pokud již byli hráči spárováni pro vytvoření zápasu.

8. **Řešení stavu neaktivního zahájení zápasu** – aplikace bude řešit stav, kdy dojde ke spárování hráčů z fronty pro tvorbu zápasu, ale nedojde k potvrzení zahájení zápasu od obou z nich.

9. **Odehrání tahu v zápasu** – hráč bude schopen odehrát svůj tah v daném kole karetní hry.

10. **Řešení stavu neaktivního zápasu** – backend aplikace bude podporovat možnost pro řešení situace, kdy se odehrání obou tahů hráčů v daném kole neuskuteční do daného limitu. Takový zápas bude předčasně ukončen kvůli neaktivitě a příslušně vyhodnocen.

11. **Vyhodnocení kola v zápasu** – jakmile jednotlivé kolo zápasu dospěje do stavu, že oba hráči budou mít odehráno, pak bude kolo vyhodnoceno.

12. **Vyhodnocení zápasu** – jakmile stav zápasu dojde do konečného stavu, bude automaticky backendem aplikace vyhodnocen. Hráči mohou za výhru získat denní odměny.

13. **Správa herních karet** – backend aplikace bude umožňovat spravovat herní karty a herní efekty, se kterými herní karty pracují.

14. **Správa balíčků herních karet** – hráči budou moci upravovat herní karty obsažené ve svých balíčcích karet, se kterými mohou hrát v zápasech proti hráčům.

15. **Změna přezdívky** – aplikace bude umožňovat uživateli změnu své přezdívky.

16. **Koupení herních karet** – jednotliví uživatelé aplikace si budou moci zakoupit herní karty, se kterými budou moci hrát zápasy.

17. **Pronájem herních karet** – jednotliví uživatelé aplikace si budou moci pronajímat herní karty, se kterými budou moci odehrát jen daný počet zápasů.

18. **Nastavení automatického prodloužení pronájmu herních karet** – aplikace bude umožňovat hráčům nastavit si pro svou pronajatou kartu automatické prodlužování pronájmu. Takový pronájem bude prodlužovat pronájmy dané karty, dokud nebude později zrušen.

19. **Herní měny** – aplikace bude podporovat dvě herní měny, které budou používány pro platby za koupi a pronájem karet.

20. **Koupě herních měn** – aplikace bude umožňovat uživatelům navýšit si svá konta herních měn.

3.1.2 Nefunkční požadavky

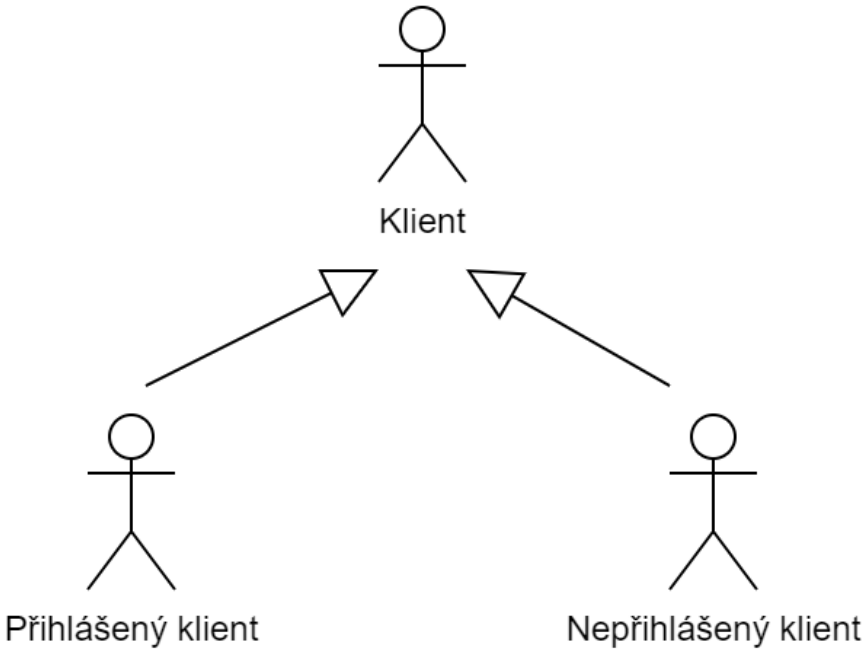
Výčet nefunkčních požadavků na aplikaci:

1. **Hra pro OS Android** – backendovou část aplikace bude možno napojit na klienta, který bude spouštěn na zařízeních s operačním systémem Android.
2. **Škálovatelnost aplikace** – backend aplikace bude škálovatelný pro možnost zvětšující se uživatelské základny.
3. **Rozšiřitelnost herních karet a efektů** – hru bude možno jednoduše rozšiřovat o nové herní karty a jejich efekty.

3.2 Diagram případů užití

3.2.1 Klientští aktéři

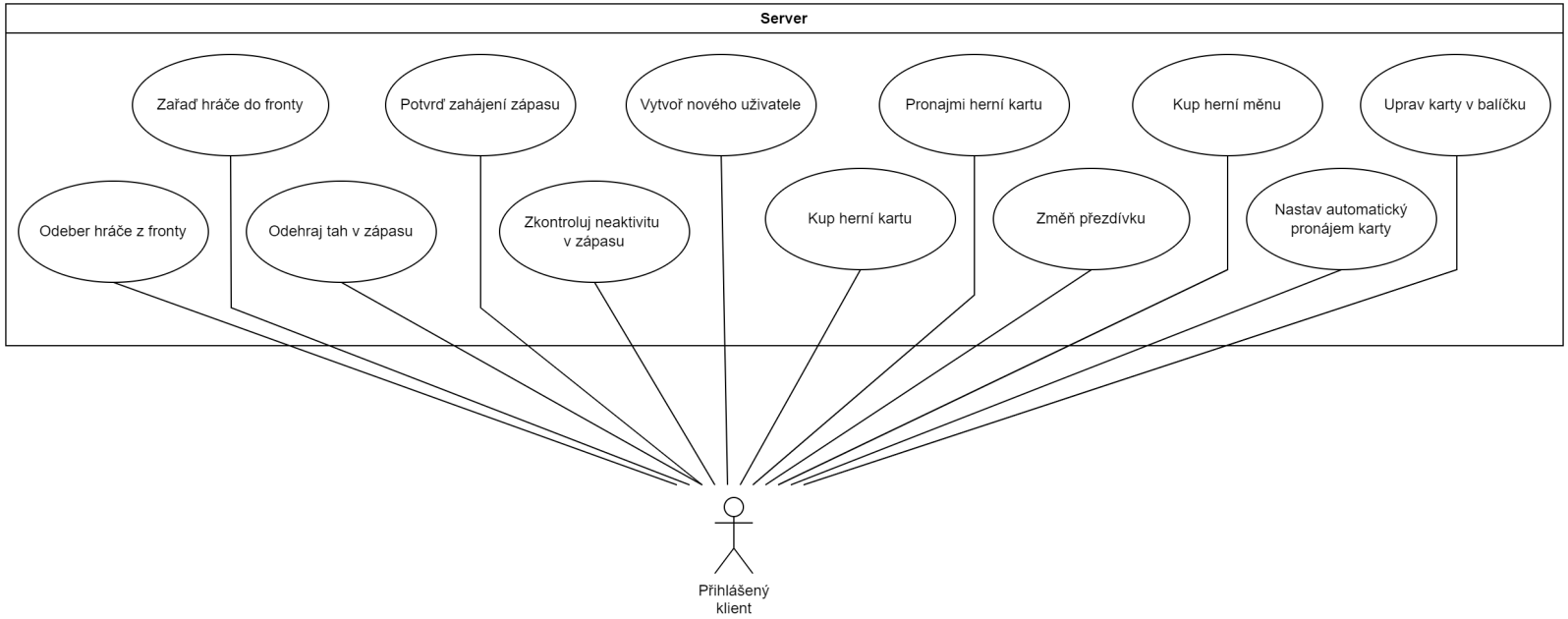
Klientská část může být buď přihlášená, nebo nepřihlášená vzhledem k herní aplikaci. Server umožní splnění klientských požadavků jen přihlášeným klientům. Na obrázku 3.1 lze vidět strukturu jednotlivých klientských aktérů.



Obrázek 3.1 Struktura klientských aktérů

3.2.2 Diagram klientských případů užití

Na obrázku 3.2 lze vidět znázorněné případy užití, které může provádět na serveru klient přihlášený do herní aplikace. Tyto případy užití uvedené v diagramu vychází přímo z funkčních požadavků kladených na herní backend. Jejich bližší popis zde tedy neuvádím, jelikož byl v textu již zmíněný.



Obrázek 3.2 Diagram případů užití pro klientskou část hry

3.3 Popis technologie

Tato podkapitola se zabývá bližším popisem technologií vybraných pro vývoj backendu karetní hry.

Po diskuzi a domluvě s Vojtěchem Mičkou jsme se rozhodli, že obě části našeho projektu karetní hry (backend i frontend) budeme implementovat s pomocí platformy Google Firebase.

V následujících podkapitolách se budu zabývat bližším popisem technologií vybraných pro vývoj mé backendové části aplikace.

3.3.1 Google Firebase

„Firebase je backendová platforma pro vytváření webových a mobilních aplikací.“ [3] (vlastní překlad)

Platforma je vlastněna společností Google. Nabízí různé služby spojené s vývojem aplikací. Tím vývojářům může značně usnadnit práci, jelikož se tak nemusejí zaměřovat na vlastní implementaci některých služeb, které jim poskytne právě platforma Firebase [3].

Firebase se zaměřuje na tři druhy problematiky, které se při životním cyklu softwaru snaží lidem zjednodušovat. Jedná se o:

- samotný aplikační vývoj,
- vydání aplikace a následné monitorování jejího chodu,
- věci týkající se propagace aplikace [3].

Platforma Firebase nabízí dva cenové plány Spark a Blaze pro využívání svých služeb. Plán Spark je plně zdarma a v rámci jeho používání je uživateli ve Firebase dostupná řada služeb (avšak s určitými omezeními). Plán Blaze je poté na rozdíl od plánu Spark plně placený, kdy člověk platí podle toho, jak moc různé Firebase služby ve skutečnosti využívá (základní neplacené limity plánu Spark však zůstávají uživateli zachovány). Tento plán již nemá žádná omezení, takže uživatel si musí své výdaje na platformě hlídat [4].

3.3.2 Cloudové funkce

Cloudové funkce (Cloud Functions) jsou jednou ze služeb, které platforma Firebase v rámci své nabídky nabízí.

Následující odstavec uvádí, jak společnost Firebase popisuje svou službu cloudových funkcí na svých stránkách.

„Cloudové funkce pro Firebase je bezserverový framework, který umožňuje automatickým způsobem spouštět backendový kód v reakci na události vyvolané událostmi na pozadí, HTTPS požadavky, administrátorským SDK nebo úlohami plánovače cloudu. Váš kód v jazycích JavaScript, TypeScript nebo Python je uložen v infrastruktuře Google Cloud a běží ve spravovaném prostředí. Není potřeba spravovat a škálovat vlastní servery.“ [5] (vlastní překlad)

Jak je již zmíněno v předchozím odstavci, cloudové funkce od Firebase jsou službou, která nese znaky takzvané bezserverové architektury. Bezserverová architektura je obecně

„způsob, jak vytvářet a provozovat aplikace a služby“ (vlastní překlad), aniž bychom museli nějak dodatečně konfigurovat servery a infrastrukturu prostředí, ve kterém poté daná aplikace běží. O tuto správu serverů se postará nějaká třetí strana (společnost). S tím se pojí také případná jednoduchost škálování serverů pro uživatele bezserverové architektury. Všechno zmíněné usnadňuje vývojářům aplikace práci, a proto se mohou pečlivěji zaměřovat na správnost jejího fungování a funkcionalit [6].

Dále je potřeba zmínit, že při jednotlivých zavoláních různých cloudových funkcí, je jejich vykonávání vzájemně v cloudovém řešení izolováno [5].

V těchto cloudových funkcích bude kód backendové části aplikace posléze implementován, takže jeho nasazení na produkční server by neměl být problém, jelikož jej nebude potřeba nějak zvláště složitě konfigurovat a podobně.

Cloudové funkce pro Firebase se dají implementovat ve třech programovacích jazycích. Já si pro svou implementaci zvolil jazyk Python verze 3.12, protože je mi ze všech tří dostupných jazyků nejbližší a taky jsem se s ním trochu již v praxi setkal. I přesto pro mě bude psaní tak velkého projektu v jazyce Python nové, což pro mě představuje jak výzvu, tak i novou zkušenost do života.

Jak jsem již zmínil, pro cloudové funkce existuje vícero různých způsobů, jak mohou být na serveru spouštěny. Pro potřeby této práce však pravděpodobně nejvíce využiji druh cloudových funkcí, které mohou být volány přímo z SDK klientské aplikace (tedy frontendové části našeho projektu). Tyto cloudové funkce se nazývají volatelné HTTP funkce (HTTP Callable functions). Typ těchto cloudových funkcí řeší oproti typu funkcí, které lze spouštět jen s pomocí HTTP požadavků, navíc také ověřování autentizačních tokenů, které jsou automaticky přikládány při volání dané cloudové funkce z klientského SDK do požadavku klienta, jsou-li na straně klienta dostupné [7].

3.3.3 Databáze

Databáze je z hlediska potřeby ukládání aplikačních dat velmi zásadní komponentou pro mou práci.

Pro účely této hry bude potřeba evidovat zejména data jednotlivých hráčů, zápasů, karet a efektů. Předpokládám, že objemy uchovávaných dat pro potřeby této práce mohou dosahovat nejvýše řádu jednotek gigabajtů. Vzhledem k tomu, že vyvíjená aplikace je tahovou hrou, čas databázového zpracování není zcela kritickou metrikou. Bude však vhodné, aby použitá databáze měla možnost dobré škálovatelnosti v případě, že by se uživatelská základna a s tím i objem uchovávaných dat časem zvětšoval. Bude potřeba, aby použitá databáze umožňovala transakční zpracování pro udržení konzistence dat v databázi.

Google Firebase nabízí v rámci svých služeb dvě různé NoSQL databáze, které provozuje ve svém cloudovém úložišti. Jedná se o databáze Cloud Firestore a Realtime Database. Cloud Firestore je databází, kterou na webu Firebase doporučují použít v případě, že je uživatel jejich novým zákazníkem [8].

V následujících sekcích popíši některé vlastnosti obou nabízených databází a na konci zvolím tu, která je vhodnější pro potřeby mé práce.

3.3.3.1 Cloud Firestore

„Cloud Firestore je flexibilní, škálovatelná databáze pro mobilní, webový a serverový vývoj“ (vlastní překlad) [9]. Jedná se o dokumentovou NoSQL databázi, čímž se značně odlišuje od běžných relačních SQL databází [9][10]. Data totiž uchovává v dokumentech sdružovaných v kolekcích. Jednotlivé uložené dokumenty v databázi mohou obsahovat data reprezentovaná jednoduchými datovými typy, ale také různými *„komplexními vnořenými objekty“* (vlastní překlad) či podkolekcemi [9].

Stručné porovnání některých základních pojmů a konceptů mezi relačními SQL a dokumentovými NoSQL databázemi uvádím v tabulce 3.1. Tabulka byla vypracována na základě srovnávání SQL databází a databáze MongoDB, což je jeden ze zástupců dokumentových NoSQL databází [11].

Tabulka 3.1 Porovnání základních pojmů a konceptů mezi relačními SQL a dokumentovými NoSQL databázemi. Získáno z webu MongoDB [10]

Relační SQL databáze	Dokumentové NoSQL databáze
databáze	databáze
tabulka	kolekce
řádek v tabulce	dokument
sloupec v tabulce	položka v dokumentu

Tato databáze také podporuje značné možnosti pro dotazování se nad daty v databázi. Jednotlivé dotazy mohou provádět jak řazení, tak i filtrování zároveň, uživatel může navíc specifikovat vícero podmínek pro filtrování dat najednou. Díky využité indexaci může být rychlost provádění dotazů nad touto databází značně zvýšena [9].

Databáze Cloud Firestore je navrhnuta tak, aby se dala dobře automaticky škálovat [8][9]. Je nabízena ve více konfiguracích pro použití v jedné nebo i ve více oblastech napříč různými datovými centry umístěnými různě po světě. Toto umožňuje zajistit, aby databáze byla škálovatelná a vysoce dostupná v globálním měřítku. Škálování databáze poté probíhá automaticky a jeho limity *„jsou přibližně 1 milion paralelních připojení a 10 000 zápisů za sekundu“* (vlastní překlad). Pro tuto databázi se doba odezvy většinou pohybuje do 30 ms [8].

Databáze Cloud Firestore také podporuje pokročilé transakční operace, v rámci kterých dokáže kdekoli v databázovém schématu zajistit atomicitu pro operace čtení a zápisu [8].

Tato databáze se podle všeho těší i nemalé oblibě u vývojářů. Podle společnosti Firebase jí důvěřuje více než 250 000 z nich [8].

3.3.3.2 Realtime Database

Realtime Database je stejně jako Cloud Firestore NoSQL databází, která je provozovaná v cloudu společnosti Google. Jedná se o JSON databázi, jelikož jsou v ní data ukládaná hierarchicky ve formátu JSON [12][8].

Toto databázové řešení již nepodporuje tak značné možnosti pro dotazování se nad daty uloženými v databázi jako předešlá databáze Cloud Firestore. Realtime Database již při dotazech týkajících se hodnoty nějakého pole v JSON stromu nepodporuje souběžné

filtrování a řazení, uživatel tedy musí jedno z nich upřednostnit. Navíc se při dotazování nad daty nepoužívá indexace, takže čím bude objem uložených dat vyšší, tím více může klesat rychlost provádění dotazů. To může být značnou nevýhodou [8].

Škálování této databáze není také tak ideální jako u předchozího zmíněného řešení a je limitované. Realtime Database je totiž databází nabízenou pouze v konfiguracích určených pro specifické oblasti, čímž není pro globální použití moc vhodná. Škálování je dosahováno s pomocí rozdělování dat (shardingu) z jedné databáze do více databázových instancí. Bez tohoto rozdělování dat může jednotlivá instance Realtime Database dosahovat limitů *„přibližně 200 000 paralelních připojení a 1 000 zápisů za sekundu“* (vlastní překlad). Pro tuto databázi se doba odezvy většinou pohybuje do 10 ms, což je velmi nízká hodnota a je to její velká výhoda [8].

Realtime Database podporuje také transakční operace, avšak atomicitu procesu úprav dokáže zaručit pouze v nějaké určité části datové hierarchie tvořené daty ve formátu JSON [8].

3.3.3.3 Rozhodnutí o výběru databáze

Pro svou práci jsem se rozhodl po srovnávání použít databázi Cloud Firestore, protože mi ze srovnání všeobecně vzešla jako vhodnější pro využití v mé backendové části projektu. Má sice o něco vyšší dobu odezvy oproti Realtime Database, ale to nebude pro využití v tahové karetní hře problém. Navíc je databáze Realtime Database vzhledem k objemu uložených dat daleko dražší službou než Cloud Firestore [4].

Kapitola 4
Návrh

Tato kapitola se zabývá záležitostmi týkajících se návrhu backendu aplikace. Lze v ní nalézt například popis použité architektury, navrženého databázového modelu či některých využitých návrhových vzorů.

4.1 Architektura aplikace

Pro vývoj backendu karetní hry jsem si vybral třívrstvou architekturu aplikace, která vychází z takzvané vícevrstvé architektury [13].

Obecná vícevrstvá architektura rozděluje kód aplikace do několika horizontálních vrstev, které jsou vzájemně logicky oddělené (nejedná se o fyzické oddělení vrstev mezi více výpočetních jednotek, ale pouze o oddělení komponent). Každá z vrstev si pak v rámci dané aplikace ponechává určitou funkci nebo zodpovědnost za něco, takže v ní pak jsou hromaděny komponenty s podobnými funkčnostmi. Vrstvy, na které je aplikace horizontálně rozdělena, musí záviset pouze jedna na druhé směrem dolů (viz obrázek 4.1) [13].

Svou aplikaci jsem tedy rozdělil do tří logicky oddělených vrstev, z nichž každá má v rámci aplikace svou zodpovědnost za určité funkcionality. Mnou zmíněné tři vrstvy jsou odshora následující:

- **Kontroléry** – jednotlivé kontroléry byly navrženy tak, že jsou tvořeny jednotlivými cloudovými funkcemi z platformy Firebase a zodpovídají za zpracování požadavků při jejich zavolání ze strany klienta (frontendu aplikace). Případné další zpracování, které klient v požadavku vznesl, dále směřují na vrstvu aplikační logiky.
- **Aplikační logika** – tato vrstva je tvořena servisními třídami, které se starají o veškeré aplikační algoritmy, herní procesy, ověřování správnosti obdržených parametrů, které této vrstvě předá vrstva kontrolérů, a podobné procesy.
- **Repozitáře** – vrstva repozitářů zodpovídá v rámci aplikace za perzistenci dat ve zvoleném databázovém řešení (v mé aplikaci databáze Cloud Firestore). Repozitáře nabízejí pro každý doménový model, který ukládám v databázi, sadu funkcí, které jsou určeny pro komunikaci s perzistentním médiem.

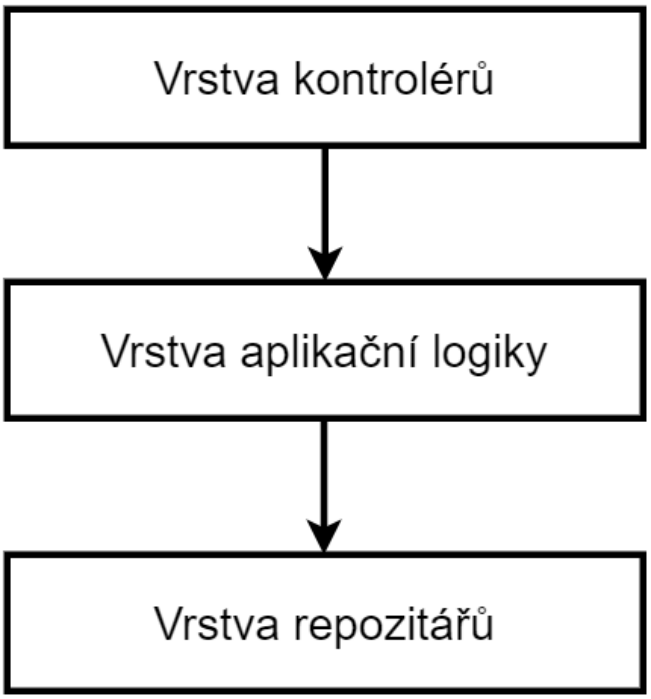
Na obrázku 4.1 je znázorněno, jak na sobě jednotlivé vrstvy v rámci mé architektury aplikace závisí.

Jako většina architektur, má i mnou zvolená vícevrstvá architektura jisté výhody a nevýhody. Mezi jednoznačné výhody lze zařadit:

- jednoduchost architektury pro použití v aplikacích,
- usnadnění testování kvůli rozdělení zodpovědností do vrstev,
- značně nezvyšuje výpočetní náročnost aplikace [13].

Mezi jednoznačné nevýhody lze zařadit:

- složitost rozšiřování aplikace používající tuto architekturu;
- fungování celé aplikace může být ohroženo i malou úpravou v nějaké horizontální vrstvě architektury, jelikož na sobě jednotlivé vrstvy závisí v řadě za sebou [13].



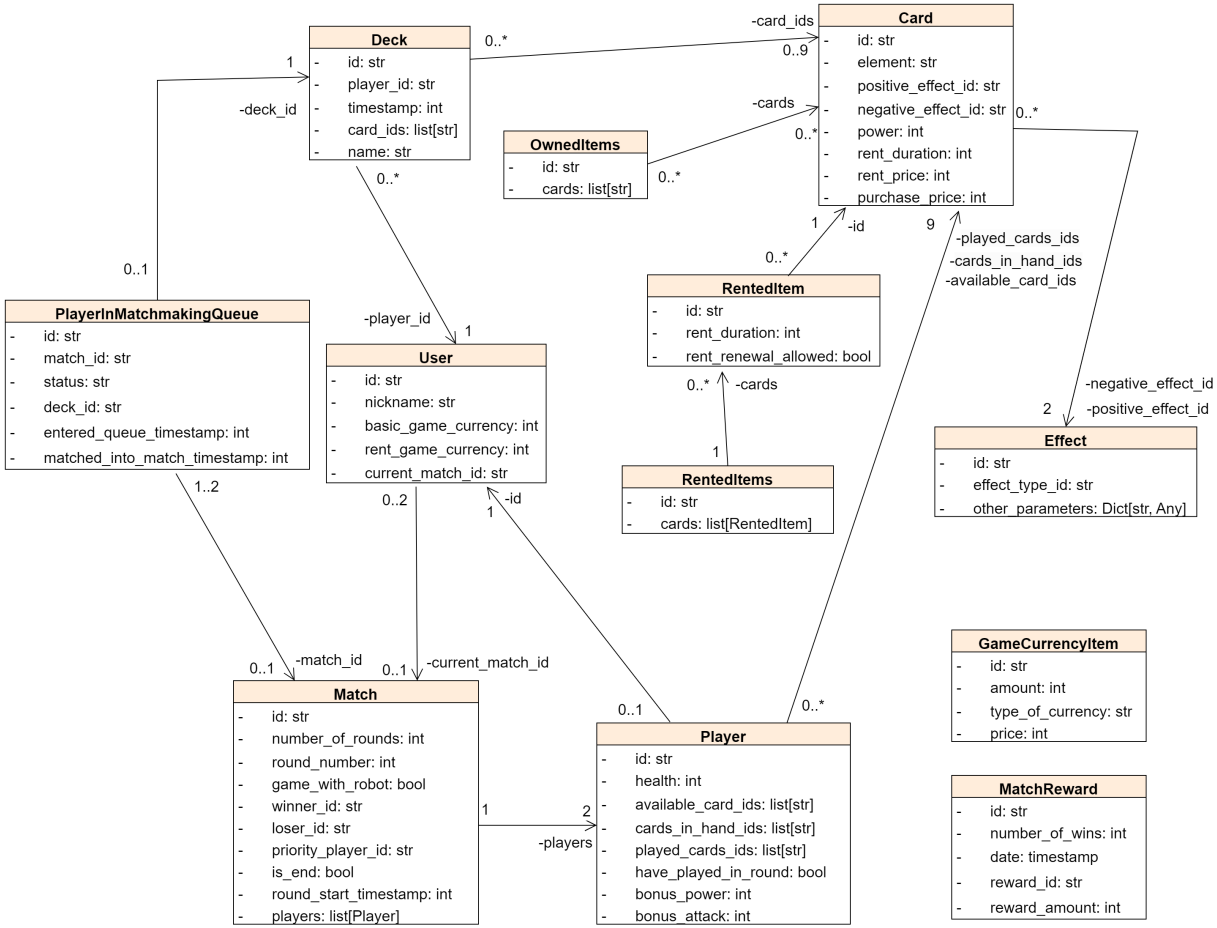
Obrázek 4.1 Třívrstvá architektura aplikace

4.2 Návrhový model tříd

Pro pochopení tříd, které v rámci své problémové domény využívám jako doménové modely, jsem sestavil návrhový model tříd, který je k vidění na obrázku 4.2.

Tyto doménové modely používám pro práci s daty v rámci celé své aplikace napříč třídami a také pro jejich perzistenci v databázi.

Na tomto místě je potřeba zmínit, že v rámci své aplikace používám veškeré modely na obrázku 4.2 jako samostatné doménové modely kromě těch, které jsou představovány třídami `RentedItem` a `Player`. Tyto dvě třídy jsou pouze vyňaty z jiných modelů pro větší přehlednost, ale při práci s doménovými modely jako celky jsou součástmi jiných tříd, v nichž pouze agregují data. Třída `RentedItem` je při používání součástí doménového modelu představovaného třídou `RentedItems`. Třída `Player` je zase při používání součástí doménového modelu představovaného třídou `Match`.



Obrázek 4.2 Návrhový model tříd pro doménu

4.3 Databázový model

Databázový model, který jsem použil, zde přímo neuvádím. Jelikož jsem si pro perzistenci aplikačních dat vybral NoSQL dokumentovou databázi, ukládaná data se nedají zachytit tak, jako to jde například pro relační SQL databáze různými diagramy. Můj databázový model je však do jisté míry velmi podobný návrhovému modelu tříd na obrázku 4.2, jelikož jsem pro ukládání dat zvolil NoSQL dokumentovou databázi. Do databáze tak mohu ukládat přímo objekty, které jsem si navrhl v již zmíněném návrhovém modelu tříd. Toho je docíleno tak, že mezi aplikačním kódem v jazyce Python a Cloud Firestore databází je prováděna konverze instancí Python tříd do databázových dokumentů a naopak pro různé čtecí a zápisové operace.

V následujícím seznamu popisuji jednotlivé kolekce ve své databázi. Názvy kolekcí v databázi jsou nyní pro jednoduchost použity stejné jako názvy tříd v návrhovém modelu tříd na obrázku 4.2.

User – kolekce, která uchovává uživatelské profily. Pro každý profil je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje uživatelský profil (generovaný skrze autentizační službu platformy Firebase);
- **nickname** – přezdívka hráče;
- **basic_game_currency** – výše zůstatku základní herní měny;
- **rent_game_currency** – výše zůstatku herní měny pro pronájem karet;
- **current_match_id** – identifikátor aktuálního zápasu, jehož je daný hráč účastníkem.

Card – kolekce, která uchovává herní karty. Pro každou kartu je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje kartu;
- **element** – herní živel, který karta obsahuje;
- **power** – síla útoku karty;
- **positive_effect_id** – identifikátor pozitivního efektu;
- **negative_effect_id** – identifikátor negativního efektu;
- **rent_duration** – délka pro pronájem dané karty;
- **rent_price** – cena pronájmu dané karty;
- **purchase_price** – cena koupě dané karty.

Deck – kolekce, která uchovává balíčky karet hráčů. Pro každý balíček je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje balíček karet;
- **player_id** – identifikátor hráče, který daný balíček vlastní;
- **timestamp** – časové razítko vytvoření balíčku karet;
- **card_ids** – seznam identifikátorů herních karet, které daný balíček obsahuje;
- **name** – název balíčku karet.

Effect – kolekce, která uchovává efekty herních karet. Pro každý efekt je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje karetní efekt;
- **effect_type_id** – identifikátor typu efektu;
- **other_parameters** – případné další parametry podle typu efektu.

GameCurrencyItem – kolekce položek herních měn, které se dají koupit. Pro každou položku je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje položku herní měny;
- **amount** – množství herní měny;
- **type_of_currency** – typ herní měny;
- **price** – cena za dané množství dané herní měny.

MatchReward – kolekce položek, které představují počty odměn získaných za výhry v zápase v daný den. Pro každou položku je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje položku pro odměny (identifikátor hráče);
- **date** – datum získání poslední odměny;
- **number_of_wins** – počet výher daný den, který je představován ukládaným datem;
- **reward_id** – identifikátor typu poslední udělené odměny;
- **reward_amount** – množství poslední udělené odměny.

Match – kolekce, která uchovává zápasy. Pro každý zápas je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje zápas;
- **number_of_rounds** – celkový počet kol zápasu;
- **round_number** – aktuální číslo kola zápasu;
- **game_with_robot** – zdali se jedná o zápas s robotem;
- **winner_id** – identifikátor vítězného hráče, existuje-li;
- **loser_id** – identifikátor poraženého hráče, existuje-li;
- **priority_player_id** – identifikátor prioritního hráče;
- **is_end** – zdali zápas již skončil;
- **round_start_timestamp** – časové razítko začátku kola;
- **players** – seznam hráčů v daném kole (třída `Player`). Pro každého hráče v seznamu je ukládáno:
  - **id** – identifikátor hráče;
  - **health** – aktuální počet životů;
  - **played_cards_ids** – identifikátory karet, které již hráč odehrál;
  - **cards_in_hand_ids** – identifikátory karet, které ještě hráč neodehrál a drží je v ruce;

- **available_card_ids** – identifikátory karet, které ještě hráč neodehrál a nedrží je v ruce;
- **have_played_in_round** – zdali v aktuálním kole již hráč odehrál svůj tah;
- bonusové síly útoku.

OwnedItems – kolekce, která uchovává položky vlastněných předmětů jednotlivých hráčů. Pro každou položku v kolekci je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje položku vlastněných předmětů (identifikátor hráče);
- **cards** – seznam identifikátorů vlastněných karet.

RentedItems – kolekce, která uchovává položky pronajatých předmětů jednotlivých hráčů. Pro každou položku v kolekci je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje položku pronajatých předmětů (identifikátor hráče);
- **cards** – seznam pronajatých karet (třída `RentedItem`). Pro každou pronajatou kartu je ukládáno:
  - **id** – identifikátor herní karty,
  - **rent_duration** – zbývající trvání pronájmu karty,
  - **rent_renewal_allowed** – zdali je povoleno automatické prodloužení pronájmu dané karty.

PlayerInMatchmakingQueue – kolekce, která uchovává hráče čekající ve frontě na zápas. Pro každou položku v kolekci je ukládáno:

- **id** – identifikátor daného dokumentu, který představuje čekajícího hráče ve frontě (identifikátor hráče);
- **match_id** – identifikátor zápasu, pokud byl daný hráč spojen s jiným hráčem pro hraní zápasu;
- **status** – stav čekajícího hráče;
- **deck_id** – identifikátor balíčku karet, se kterým chce hráč hrát v zápase;
- **entered_queue_timestamp** – časové razítko doby, kdy se hráč připojil do fronty;
- **matched_into_match_timestamp** – časové razítko okamžiku, kdy byl hráč spárován s jiným hráčem do zápasu.

4.4 Klientská rozhraní

Backendová část aplikace nabízí pro vhodné využití klientem dvě různá rozhraní. Jedno z nich je tvořeno cloudovými funkcemi, které mohou být volány ze strany SDK klientské aplikace (frontendu). Za druhé z nich považuji klientovi vystavenou databázi, ze které může klientská aplikace za běhu číst.

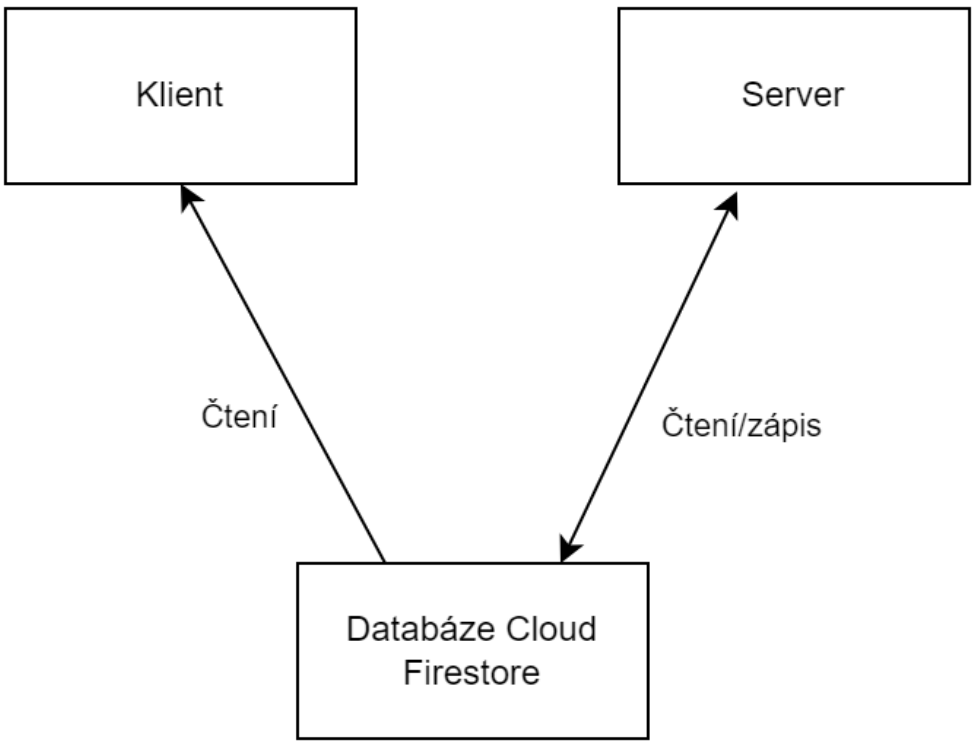
4.4.1 Rozhraní cloudových funkcí

Rozhraní cloudových funkcí, které jsou volány ze strany klienta, aby prováděly akce herní logiky a změny stavů, je tvořeno volatelnými (Callable) cloudovými funkcemi. Celý seznam cloudových funkcí včetně očekávaných parametrů a kódů odpovědí je možno nalézt v příloze A.

4.4.2 Databázové rozhraní

S kolegou Vojtěchem Mičkou, který pracuje na klientské části karetní hry, jsme se domluvili, že jím vytvořená klientská aplikace bude moci nahlížet a číst z databáze Cloud Firestore, kterou v backendové části aplikace využívám pro ukládání dat. Dohodli jsme se tak, protože klientská aplikace může v reálném čase „pozorovat“, zda se v databázi něco nezměnilo, a tím případně na změny rychleji reagovat.

Obrázek 4.3 znázorňuje, jak databáze komunikuje s klientskou a serverovou částí aplikace.



Obrázek 4.3 Popis komunikace databáze s klientskou a serverovou částí aplikace

4.5 Dependency Injection

V mé aplikaci byla již od návrhu využita technika návrhového vzoru Dependency Injection (DI). Tento návrhový vzor se zabývá problematikou, jak dodávat třídám jejich třídní závislosti na jiných třídách a objektech, aniž by byly tyto závislé třídy pevně svázané s jinými specifickými třídami. Toho se docílí tak, že závislé třídy budou využívat abstrakce tříd (rozhraní), na kterých jsou závislé. V samotném programu však budou poté reálně využívány specifické implementace těchto abstrakcí [14].

Existuje více možností, jak při implementaci využívat Dependency Injection. Já si během návrhu však vybral způsob, který využívá třídní konstruktory. To funguje tak, že během konstrukce instance dané třídy jsou jí do konstruktoru poskytnuty všechny její závislosti z jejího zevnějšku (v závislé třídě jsou tyto závislosti skryty za abstrakcemi) [14].

Používání návrhového vzoru DI při vývoji softwaru má následující výhody:

- usnadnění testování a využívání techniky mockování v testech,
- snížení závislostí mezi specifickými třídami,
- specifické implementace závislostí mohou být jednoduše v závislých třídách nahrazovány,
- zlepšování udržitelnosti kódu [14].

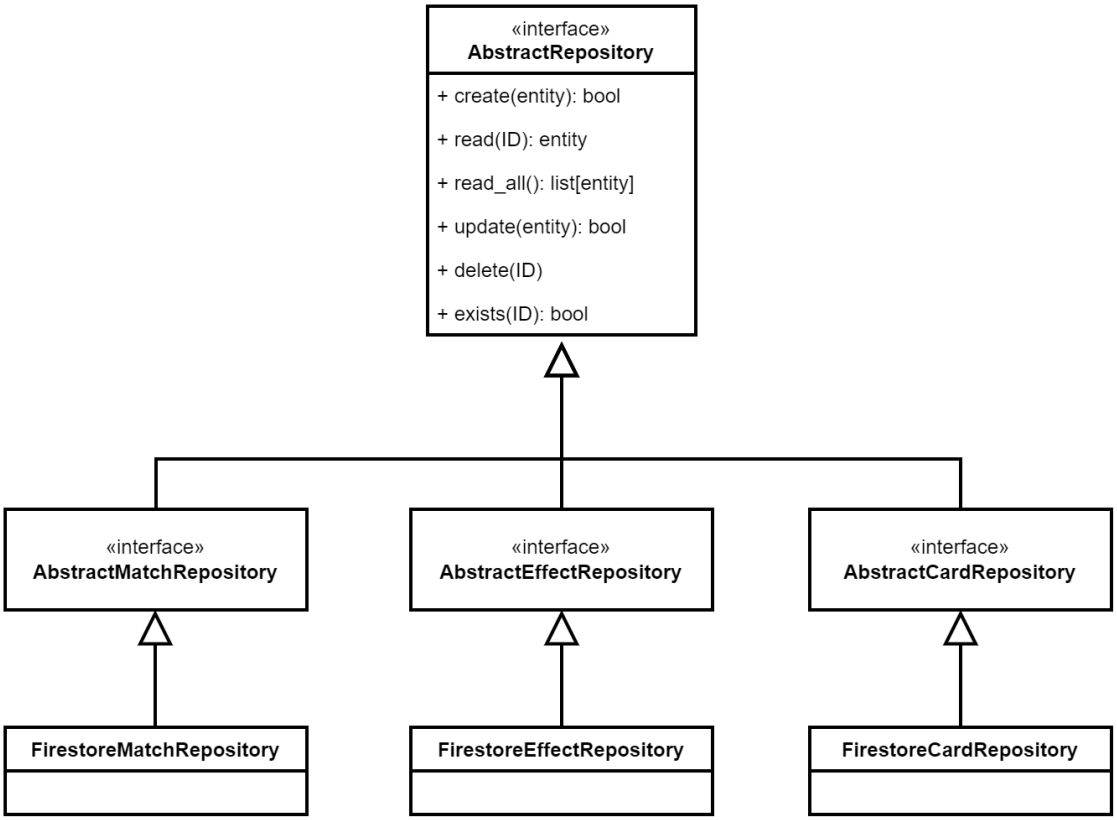
4.6 Návrh repozitářů

Repozitáře jsou v mé aplikaci umístěny ve vlastní nejspodnější vrstvě mé třívrstvé architektury. Pro jejich návrh, jak už název vrstvy napovídá, byl použit takzvaný návrhový vzor Repository.

Obecně se návrhový vzor Repository zabývá poskytováním jednotného rozhraní pro správu dat mezi aplikační logikou a specifickým typem datového úložiště. Toto slouží také jako obecná abstrakce specifického datového úložiště, které daná aplikace používá pro uchovávání dat. Tím je umožněno snadněji měnit abstrakcí skryté datové úložiště reálně využívané aplikací. Používání tohoto vzoru zlepšuje navíc testovatelnost a udržitelnost kódu aplikace [15].

V mé aplikaci tedy tento návrhový vzor abstrahuje použití specifické databáze Cloud Firestore a navenek pro vyšší vrstvu aplikační logiky poskytuje jednotné rozhraní pro práci s daty v perzistentním médiu. Pro každý doménový model, který chci ukládat do perzistentního úložiště, jsem si definoval vlastní repozitář, který bude pracovat pouze s daty tohoto doménového modelu.

Na obrázku 4.4 lze vidět diagram návrhu mnou používané repozitářové struktury. Pro tuto ukázku jsou na něm vyobrazeny pouze tři specifické repozitáře, které se váží k některým doménovým modelům mé aplikace. Ostatní repozitáře pro další doménové modely by byly vyobrazeny obdobně a z toho důvodu byly vynechány.



Obrázek 4.4 Ukázka struktury repozitářů

4.7 Karty a efekty

V nefunkčních požadavcích bylo specifikováno, aby mohly být karty a jejich efekty snadno rozšiřitelné. Tato snadná rozšiřitelnost karet a efektů byla vymyšlena tak, že bude spočívat v ručních zásazích do kolekcí dokumentů ukládaných v databázi přes webové rozhraní platformy Firebase. Karty tedy musí pracovat s předem definovanými efekty v databázi, aby nedocházelo k chybám během hry.

Karetní efekty byly navrženy s ohledem na jejich případnou snadnou rozšiřitelnost. Takže doménový model `Effect` dokáže ukládat proměnné množství parametrů podle lišících se druhů efektů. V herní logice tak v případě přidání nového efektu poté stačí přidat novou servisní třídu, která bude při vyhodnocování výsledku daného kola uplatňovat nově přidaný efekt na aktuální stav zápasu.

V herní logice při uplatňování karetních efektů byl využit návrhový vzor Factory method, aby mohlo být za běhu programu rozlišováno, která servisní třída má uplatňovat který druh efektu. Tento návrhový vzor umožnil skrýt za abstrakci specifickou implementaci metody, která za běhu podle dodaného identifikátoru druhu efektu vytváří specifickou instanci servisní třídy pro uplatňování daného efektu v zápase [16].

4.8 Procesy

Pro zjednodušení popisu zde uvádím popis pouze některých významných procesů, které se v herním backendu odehrávají.

4.8.1 Autentizace

Při volání jednotlivých cloudových funkcí ze strany klientské aplikace jsou tyto požadavky na backend automaticky platformou Firebase autentizovány.

4.8.2 Tvorba uživatele

Klientská aplikace řeší možnosti přihlášení a registrace uživatelů. Klient karetní hry při registraci uživatele zavolá danou cloudovou funkci, která pro příslušného hráče vytvoří nový uživatelský profil v backendu karetní hry. V rámci toho je hráči přidáno do vlastnictví několik herních karet pro začátek.

4.8.3 Tvorba zápasu

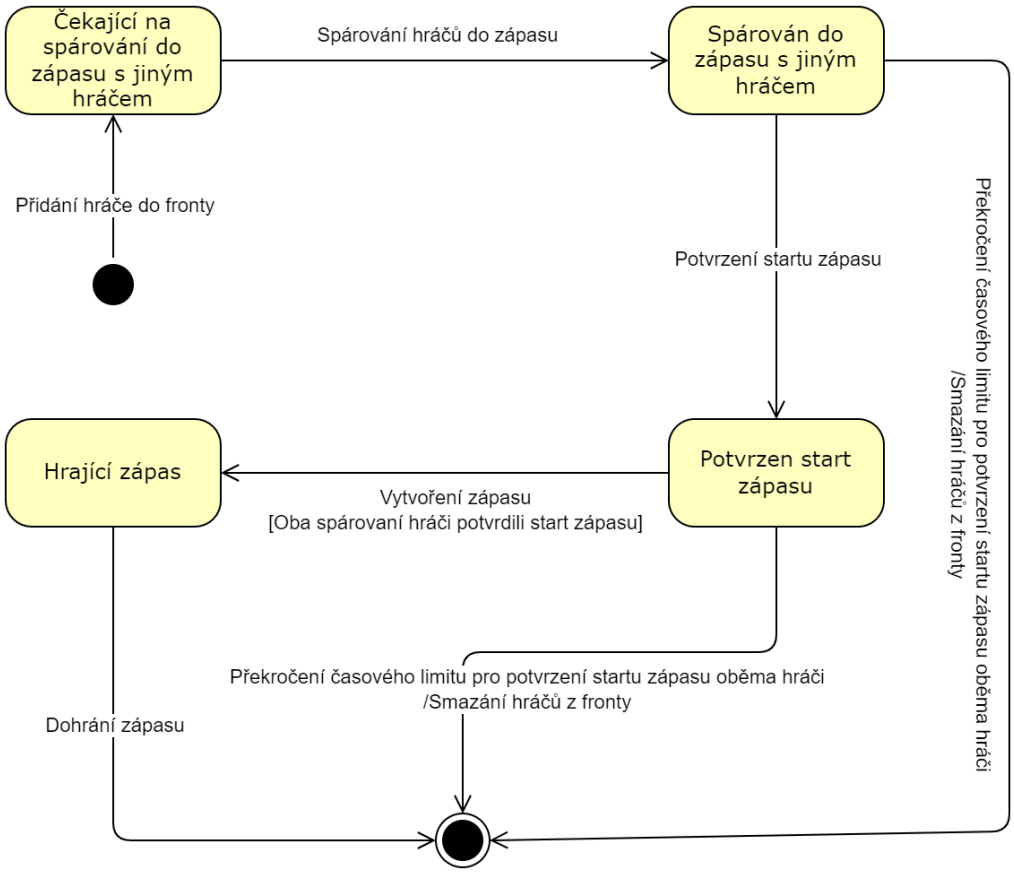
Herní backend umožňuje klientské aplikaci přidání hráče do fronty pro čekání na zápas. Při tom musí klient specifikovat také identifikátor balíčku karet, se kterým se chce hráč zápasu účastnit. Tvorba zápasu byla inspirována tímto internetovým článkem [17].

Specifikovaný balíček karet je nejprve zkontrolován, zda splňuje kritéria podle pravidel hry (například stejný podíl všech živlů v celém balíčku) a až poté je případně hráč zařazen do fronty. Jakmile je hráč jednou zařazen do fronty (kolekce v databázi), je toto časové razítko uloženo v databázi a hráč čeká na spárování do zápasu s jiným hráčem anebo s robotem.

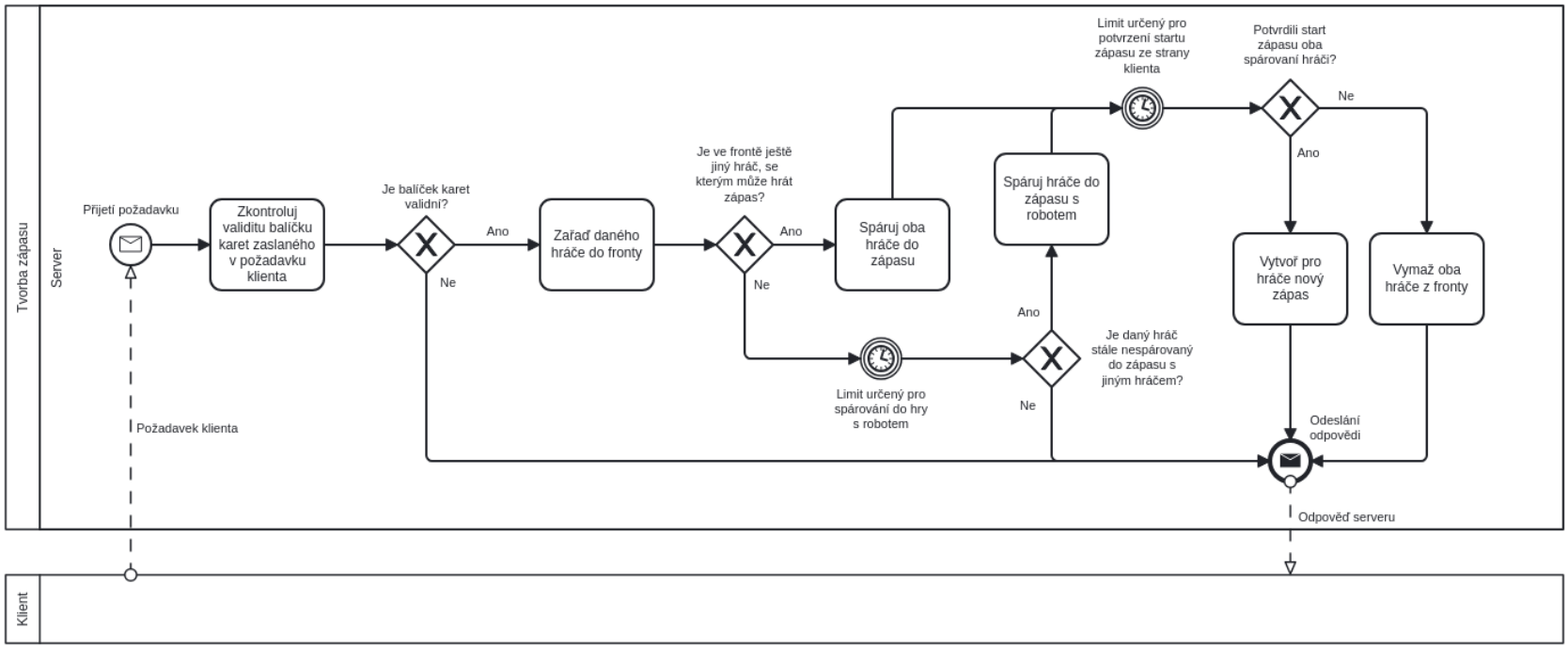
Na backendu poté běží periodicky spouštěná cloudová funkce, která sleduje časová razítka a stavy jednotlivých hráčů zařazených do fronty a při překročení daného časového limitu páruje dosud nespárované hráče z fronty do hry s robotem. Pokud se do fronty připojí ještě před spárováním s robotem jiný druhý hráč, je spárován do zápasu s prvním z těchto hráčů. Ať už byl první z hráčů nakonec spárován do zápasu s jiným hráčem nebo s robotem, je tento čas také zaznamenán do databáze.

Tímto začíná fáze, kdy se čeká na klienta, aby potvrdil samotné spuštění zápasu pro jednotlivé hráče (robot nepotřebuje toto potvrzení). Samotný zápas je vytvořen, až když je spuštění zápasu potvrzeno ze strany všech klientů (pokud nevyprší daný časový limit pro potvrzení). V tuto chvíli již mohou klientské aplikace odehrávat tahy ve vytvořeném zápase.

Na obrázku 4.5 lze vidět stavy, ve kterých se může nacházet hráč ve frontě pro tvorbu zápasu. Na obrázku 4.6 lze zase vidět BPMN diagram znázorňující proces tvorby zápasu více schematicky.



Obrázek 4.5 Stavový UML diagram pro hráče čekajícího ve frontě pro tvorbu zápasu



Obrázek 4.6 BPMN diagram znázorňující proces tvorby zápasu

4.8.4 Odehrání tahu v zápase

Pro odehrání tahu hráče v zápase potřebuje backend od frontendu získat identifikátor karty, se kterou chce hráč následně tah odehrát, a také identifikátor zápasu, kterého se hráč účastní. Tyto údaje jsou následně backendem ověřeny. Například kdyby se klient snažil odehrát kartu, kterou nemá hráč aktuálně v ruce, a podobně. Pokud je tah vyhodnocen jako validní, je zahraná karta přidána do seznamu zahraných karet *played_cards_ids* a odebrána ze seznamu karet *cards_in_hand_ids*, které hráč drží v ruce. Také je při odehrání tahu nastavena danému hráči Booleovská hodnota *have_played_in_round* jako pravdivá, aby se dalo rozlišit, kdo již odehrál v daném kole svůj tah. Tato proměnná bude nastavována na nepravdivou vždy při vyhodnocování kola, kdy již musí mít oba hráči odehrán svůj tah.

4.8.5 Vyhodnocení kola

Vyhodnocení daného kola v daném zápase funguje podle pravidel daných hrou. Kolo je vyhodnoceno, jakmile učiní v daném kole tah druhý hráč v pořadí. Při vyhodnocování kola je zároveň vybírána náhodná karta ze zbytku balíčku obou hráčů, aby jim mohla být přidána do seznamu karet, které mají aktuálně hráči v ruce. Probíhá tam samozřejmě také inkrementace hodnoty představující číslo aktuálního kola či reset hodnot indikujících odehrání jednotlivých hráčů v daném kole.

4.8.6 Koupě karty

Pro koupi karty musí frontend backendu aplikace poskytnout identifikátor karty, kterou si chce daný hráč zakoupit. Pokud je identifikátor validní, bude zkontrolováno, zda danou kartu již nevlastní, zrovna si ji nepronajímá a zda má dostatek herní měny na její zakoupení. Pokud to je vše v pořádku, bude mu karta zařazena do seznamu vlastněných karet a z uživatelského profilu odečtena částka v herní měně mincí, za kterou si danou kartu pořídil.

4.8.7 Pronájem karet

Pokud bude chtít klient pronajmout kartu danému uživateli, bude muset serverové části poskytnout identifikátor dané karty. Server poté zkontroluje validitu identifikátoru a také jestli není daná karta již pronajatá, vlastněná daným hráčem a jestli má hráč dostatek financí na její pronájem. Pokud předešlé platí, pak je karta přidána do seznamu pronajatých karet a automatické prodloužení pronájmu je u ní nastaveno na vypnuto. Z uživatelského profilu hráče je následně stržena daná částka za pronájem karty v herní měně klíčů.

4.8.8 Koupě herních měn

Herní měny se dají zakoupit skrze předdefinované balíčky herních měn, které jsou uloženy v databázi. Klient tak serveru poskytne identifikátor nějakého balíčku herní měny, který si chce daný hráč zakoupit. Množství dané měny v tomto balíčku je poté hráči přičteno do jeho uživatelského profilu.

4.8.9 Robot jako protihráč

Robot jako protihráč byl navržen tak, aby do zápasů vstupoval s předdefinovaným balíčkem karet, který je uložen v databázi. Tento robot poté odehrává své tahy v zápase, jakmile odehrává svůj tah jeho reálný protihráč. To znamená, že jakmile reálný hráč v daném kole odehraje kartu, ihned se kolo zápasu vyhodnotí.

Tahy robotického hráče byly navrženy tak, aby byly čistě náhodné a nezávisely na tom, jakou kartu zrovna odehrává jeho protihráč. Třída provádějící tahy robota je však připravena na případná vylepšení. Její aktuální implementace může být snadno nahrazena jinou, kde bude například místo naprosto náhodného algoritmu použit nějaký více komplexnější a důmyslnější algoritmus.

Kapitola 5
Implementace

Tato kapitola se zaměřuje na implementační záležitosti této bakalářské práce. Výpisy kódu použité v této kapitole jako ukázky jsou upravenými verzemi opravdového kódu implementace, pro zjednodušení z nich byly například vyjmuty závislosti a podobně.

5.1 Struktura vytvořeného projektu

Pro úvod do implementace zde stručně popíši strukturu mnou vytvořeného projektu herního backendu. Do popisu zahrnu jen ty nejpodstatnější části. V kořeni přiloženého projektu lze nalézt pár konfiguračních souborů souvisejících s platformou Firebase. Ve složce `functions` lze nalézt všechno týkající se cloudových funkcí. Tato složka obsahuje:

- `src` – složka sloužící jako kořenový balíček pro zdrojový kód v jazyce Python,
- `main.py` – inicializační soubor v jazyce Python sloužící k inicializaci jednotlivých cloudových funkcí.

Složka `src` je v jednotlivých zdrojových kódech používána jako kořenový balíček celého Python projektu. Tato složka dále obsahuje:

- `tests` – balíček obsahující testy pro implementaci,
- `main` – balíček obsahující samotnou implementaci.

Obsah složky `main` reflektuje do svého členění použitou třívrstvou architekturu. Složka `main` obsahuje:

- `controllers` – balíček, který představuje vrstvu kontrolérů a obsahuje implementaci jednotlivých cloudových funkcí;
- `services` – balíček, který představuje vrstvu aplikační logiky a obsahuje implementaci jednotlivých servisních tříd;
- `repository` – balíček, který představuje vrstvu repozitářů a obsahuje jejich implementaci;
- `models` – balíček obsahující implementaci doménových modelů;

- `loggers` – balíček obsahující implementaci komponent pro logování;
- `exceptions` – balíček obsahující implementaci výjimek;
- `configuration` – balíček obsahující konfigurační soubory.

5.2 Nástroje pro vývoj

Pro implementaci backendu své práce v jazyce Python jsem zvolil jako vývojové prostředí (IDE) Visual Studio Code [18]. V tomto IDE jsem pro vývoj použil některá rozšíření, která se do něj dají doinstalovat v jeho zabudovaném tržišti rozšíření. Jedná se o rozšíření Python a Pylance od společnosti Microsoft. Rozšíření Python je obecné rozšíření pro jazyk Python v rámci prostředí Visual Studio. Toto rozšíření poskytuje možnosti přidání dalších nadstaveb zaměřených na programovací jazyk Python. Při jeho instalaci do Visual Studia se s ním nainstalují také rozšíření Pylance a Python Debugger [19]. Pylance je užitečný statický typový kontrolor pro jazyk Python, díky kterému je schopen vývojář objevit některé chyby ihned při psaní kódu, a ne až při samotném spuštění programu [20]. Pro správu historie verzí mého projektu jsem využil systém pro správu verzí Git [21].

5.3 Konfigurační soubory

Během implementace se objevila potřeba sdružovat někam pospolu různé konfigurační hodnoty tak, aby nezůstávaly pevně svázané s kódem v různých třídách aplikace.

Přišel jsem tedy s tím, že jsem tyto literály vyjmul mimo používaná místa do konfiguračních tříd. Tyto konfigurační třídy se v projektu nacházejí v balíčku `src.main.configuration`.

V následujícím textu uvedu příklad implementace jedné konfigurační třídy určené pro herní karty. Pro konfiguraci herních karet jsem si zavedl abstraktní třídu `AbstractCardConfiguration`, kterou lze vidět ve výpisu kódu 5.1. Tato abstraktní třída obsahuje řetězce, které představují jednotlivé typy živlů herních karet, které používám v aplikační logice.

Zmíněnou abstraktní třídu jsem si zavedl z důvodu využívání návrhového vzoru Dependency Injection při vytváření tříd, které jsou závislé na mých konfiguračních třídách.

Výpis kódu 5.1 Abstraktní třída `AbstractCardConfiguration` pro konfiguraci herních karet

```
1 class AbstractCardConfiguration(ABC):
2     fire_element: str
3     earth_element: str
4     water_element: str
```

Specifickou implementací třídy `AbstractCardConfiguration` je třída `JsonCardConfiguration`, která, jak její název napovídá, čte hodnoty svých konfiguračních proměnných ze souboru ve formátu JSON. Implementaci třídy `JsonCardConfiguration` lze vidět na výpise kódu 5.2.

Výpis kódu 5.2 Implementace třídy `JsonCardConfiguration` pro konfiguraci herních karet využívající soubor ve formátu JSON

```
class JsonCardConfiguration(JsonCommonConfiguration,
                            AbstractCardConfiguration):

    def __init__(self, json_configuration_file_name: str =
                 "json_configuration_files/card_configuration.json"
             ):
        JsonCommonConfiguration.__init__(self, json_configuration_file_name)

    def _parse_configuration(self, configuration: Dict[str, Any]):
        self.fire_element = configuration["fire_element"]
        self.earth_element = configuration["earth_element"]
        self.water_element = configuration["water_element"]
```

Jak jde na výpise kódu 5.2 vidět, třída `JsonCardConfiguration` implementuje abstraktní třídu `AbstractCardConfiguration` a také abstraktní třídu `JsonCommonConfiguration`.

Implementaci třídy `JsonCommonConfiguration` lze vidět ve výpisu 5.3. Tato třída spouští ze svého konstruktoru metodu, která nejdříve načítá JSON soubor v předem dané cestě a převede ho na slovníkový datový typ. Poté volá metodu `_parse_configuration`, která je implementovaná v jejím potomkovi (třídě `JsonCardConfiguration`) a která zajistí načtení hodnot ze slovníkového typu do třídních proměnných, které jsou předepsány již v abstraktní třídě `AbstractCardConfiguration`, ze které `JsonCardConfiguration` dědí.

Výpis kódu 5.3 Abstraktní třída `JsonCommonConfiguration` implementující společné metody pro načítání dat ze souboru ve formátu JSON

```
class JsonCommonConfiguration(ABC):
    def __init__(self, json_configuration_file_name: str):
        self._file_path = Path(__file__).parent / json_configuration_file_name
        self._initialize_configuration()

    def _initialize_configuration(self):
        config = self._load_configuration()
        self._parse_configuration(config)

    def _load_configuration(self) -> Dict[str, Any]:
        with open(self._file_path, 'r') as file:
            return json.load(file)

    @abstractmethod
    def _parse_configuration(self, configuration: Dict[str, Any]):
        pass
```

5.4 Logování

Při implementaci jsem také realizoval možnost logovat aktivitu, kterou procesy v aktuálním čase vykonávají. Pro tento účel jsem vytvořil abstraktní třídu `AbstractLogger`, která poskytuje jednotné rozhraní pro logování (tato abstraktní třída bude opět poskytována jiným třídám namísto jejích specifických implementací v souladu s návrhovým vzorem Dependency Injection). Tímto rozhraním je myšlena metoda `log`, která bere jako parametry řetězec zprávy, kterou chceme zaznamenat, a typ zaznamenávané zprávy, který je implementován s pomocí výčtového datového typu `LogType`, jehož implementaci lze vidět na výpise kódu 5.5. Rozhraní nabízené třídou `AbstractLogger` lze vidět na výpise kódu 5.4.

Výpis kódu 5.4 Abstraktní třída `AbstractLogger` poskytující jednotné rozhraní pro logování aktivity aplikace

```
1 class AbstractLogger(ABC):
2     @abstractmethod
3     def log(self, log_type: LogType, message: str):
4         pass
```

Výpis kódu 5.5 Výčtový datový typ `LogType` používaný při logování aktivity aplikace

```
1 class LogType(Enum):
2     INFO = "info"
3     WARNING = "warning"
4     ERROR = "error"
5     DEBUG = "debug"
```

Specifickou implementací třídy `AbstractLogger` je pro účely mé aplikace třída `FirebaseLogger`, která využívá možnost zaznamenávat aktivitu aplikace s pomocí modulu `logger` od platformy Firebase. V implementaci této třídy, kterou lze vidět na výpise kódu 5.6, jsem rozdělil chování podle typu zaznamenávané zprávy získaného díky výčtovému datovému typu `LogType`.

Záznamy aktivity, které jsou vytvářené s pomocí třídy `FirebaseLogger`, se poté dají procházet v rámci rozhraní platformy Firebase.

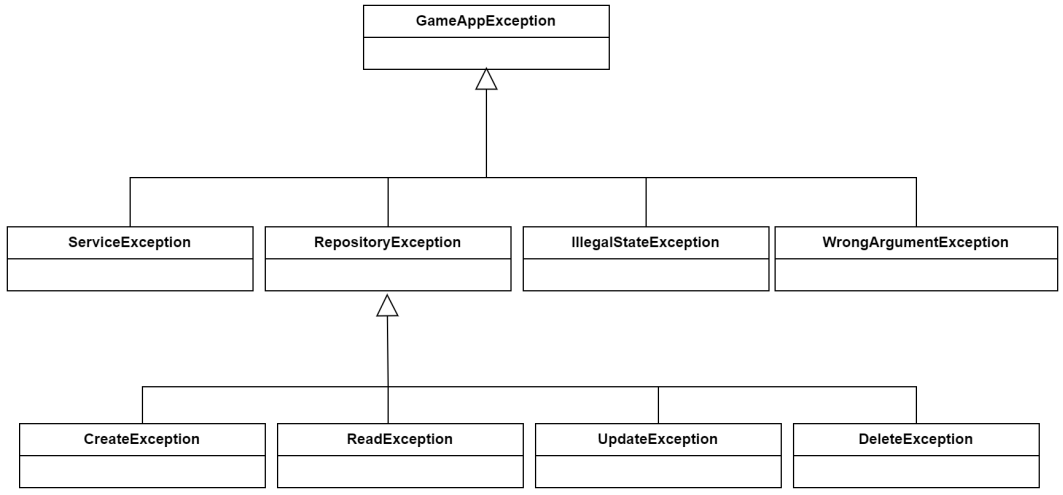
Výpis kódu 5.6 Třída `FirebaseLogger` implementující rozhraní abstraktní třídy `AbstractLogger` pro logování aktivity aplikace

```
class FirebaseLogger(AbstractLogger):
    def log(self, log_type: LogType, message: str):
        log_methods = {
            LogType.INFO: logger.info,
            LogType.WARNING: logger.warn,
            LogType.ERROR: logger.error,
            LogType.DEBUG: logger.debug
        }

        log_methods[log_type](message)
```

5.5 Ošetření chyb

Nastalé chyby se dají jednak v aplikaci logovat, ale také jsem v backendové části aplikace vytvořil strukturu výjimek, které lze vidět na obrázku 5.1. Pokud je za běhu aplikace nějaká výjimka vyvolána, je poté zachycena až na úrovni vrstvy kontrolérů.



Obrázek 5.1 Ukázka struktury tříd výjimek

5.6 Transakce

Transakční zpracování umožňuje zachování konzistence dat v databázi. Pokud by nebyly operace prováděné backendem nad databází plně transakční, hrozí řada problémů.

Vzhledem k implementované karetní hře to může způsobovat například následující problémy:

- mizející odehrané tahy jednotlivých hráčů v zápase;
- hráč čekající ve frontě na tvorbu zápasu je spojen do zápasu s více než jedním protihráčem;
- mizející měna hráčů, aniž by byla zakoupená karta přidána do vlastnictví.

Pro to, aby byly jednotlivé aktivity spouštěné ze strany klientské aplikace prováděny v backendové části aplikace atomicky (zejména však v databázi), jsem chtěl využít techniky transakčního zpracování, které nabízí platforma Firebase pro své databázové řešení Cloud Firestore. Až během implementace tohoto transakčního zpracování do své aplikace jsem však narazil na problémy, které se mi podařilo vyřešit pouze kompromisem a rozhodně ne zcela.

Platforma Firebase nabízí pro mou implementaci v jazyce Python možnost, jak implementovat transakční zpracování, s pomocí konstruktů jazyka Python nazvaných dekorátory. Na výpise kódu 5.7 lze vidět ukázku z dokumentace platformy, jak tento transakční dekorátor (`@firestore.transactional`) v jazyce Python používat. Jde o to, že transakčním dekorátorem se anotuje funkce, která bere jako svůj první parametr transakční objekt, který je ve funkci poté distribuován až k operacím prováděným nad databází (viz řádek 6 v ukázce 5.7) [22].

Výpis kódu 5.7 Ukázka použití transakčního dekorátoru pro databázi Cloud Firestore v jazyce Python. Získáno z webu platformy Firebase [22]

```
1  transaction = db.transaction()
2  city_ref = db.collection("cities").document("SF")
3
4  @firestore.transactional
5  def update_in_transaction(transaction, city_ref):
6      snapshot = city_ref.get(transaction=transaction)
7      transaction.update(city_ref,
8          {"population": snapshot.get("population") + 1})
9
10 update_in_transaction(transaction, city_ref)
```

Úskalím těchto transakcí, které jsou platformou Firebase pro databázi Cloud Firestore nabízeny, je však fakt, že v nich musejí být všechny čtecí operace provedeny před všemi zápisovými operacemi do databáze (čtecí a zápisové operace se tedy nesmí vzájemně míchat) [22]. To zkomplikovalo mé přemýšlení nad tím, jak by měly databázové transakce přirozeně fungovat vzhledem k vrstvě aplikační logiky.

Má implementace, která má vrstvu aplikační logiky oddělenou od specificky použitého databázového systému totiž přirozeně různě míchá čtecí a zápisové přístupy do databáze tak, jak je to vzhledem k zapouzdření jednotlivých tříd aplikační logiky vhodné. Takový způsob procesů aplikační logiky se mi tedy nakonec nepovedlo na transakce od

Firebase přizpůsobit, jelikož by to celé obecně transakčně nemohlo fungovat vzhledem k omezující podmínce na pořadí prováděných operací v rámci databázových transakcí. Přizpůsobovat aplikační logiku použitému databázovému systému Cloud Firestore jsem také příliš nechtěl, jelikož by to ovlivnilo použité zapouzdření tříd, což mi nepřišlo jako ideální řešení. Zapouzdření tříd by tak totiž muselo reflektovat, které operace budou prováděny v transakcích, a podle toho případně upravovat přístupy do databáze, aby nekolidovaly již s přístupy, které již provedla jiná třída v rámci té stejné transakce.

Pro svou implementaci jsem tedy zvolil kompromisní řešení, kdy jsem do transakčního bloku zahrnul pouze operace na nejvyšších vrstvách zanoření v rámci provádění nějakého procesu aplikační logiky. Tím jsem splnil podmínku na požadované pořadí čtecích a zápisových operací, které po mně požadují transakce databáze Cloud Firestore. Nezaručil jsem tím ale plnou atomicitu všech změn, které se v databázi při vykonávání některých procesů stávají. Provádění některých databázových změn, které jsou v hierarchii volání metod značně zanořeny, totiž způsobuje v rámci jedné transakce porušení podmínky na pořadí čtecích a zápisových operací vzhledem k vyšším vrstvám hierarchie volání.

Má implementace díky částečně zakomponovanému transakčnímu zpracování dokáže v některých aplikačních procesech zaručit většinovou atomicitu jejich provedení vůči databázi. Určitě ale svou implementaci nemohu v kontextu celé backendové části aplikace nazvat naprosto transakčně bezpečnou. V teoretické rovině za jistých okolností mohou paralelní úpravy v databázi vést k problémům podobným těm, které jsem zmínil na počátku této sekce o transakcích. Během následného testování funkčního prototypu se žádný takový problém nevyskytnul.

Má implementace pro databázové transakce je opět postavena na abstrakcích, aby nevznikaly přímé závislosti mezi vrstvou aplikační logiky a specificky použitým typem databáze. Implementované transakční objekty lze nalézt v balíčku `src.main.repository.transactions`.

Zavedl jsem abstraktní třídu `AbstractTransaction` (viz výpis kódu 5.8) a `AbstractTransactionManager` (viz výpis 5.9). Třída `AbstractTransaction` slouží k abstrakci specifického transakčního objektu představujícího transakční kontext. Třída `AbstractTransactionManager` zase umožňuje vykonat dodanou funkci v kontextu jedné transakce. Funkce v rámci své vlastní logiky musí pracovat s transakčním objektem, který je jí při spouštění dodán díky transakčnímu manažeru.

Výpis kódu 5.8 Ukázka abstraktní třídy `AbstractTransaction` představující transakci

```
1 class AbstractTransaction(ABC, Generic[T]):
2     @abstractmethod
3     def get_transaction(self) -> T:
4         pass
```

Výpis kódu 5.9 Ukázka abstraktní třídy `AbstractTransactionManager`, která představuje transakčního manažera

```
class AbstractTransactionManager(ABC):
    @abstractmethod
    def run_transaction(self, function: Callable, *args, **kwargs) -> Any:
        pass

    @abstractmethod
    def get_transaction(self) -> AbstractTransaction:
        pass
```

Specifické implementace výše zmíněných abstraktních tříd pro použití s databází Cloud Firestore lze vidět na výpisech kódu 5.10 a 5.11.

Výpis kódu 5.10 Ukázka třídy `FirestoreTransaction` představující implementaci transakčního objektu pro databázi Cloud Firestore

```
class FirestoreTransaction(AbstractTransaction[Transaction]):
    def __init__(self, transaction: Transaction):
        self._transaction = transaction

    def get_transaction(self) -> Transaction:
        return self._transaction
```

Výpis kódu 5.11 Ukázka třídy `FirestoreTransactionManager` představující implementaci transakčního manažera pro databázi Cloud Firestore

```
class FirestoreTransactionManager(AbstractTransactionManager):
    def __init__(self, firestore_client: firestore.Client):
        self._transaction = firestore_client.transaction()

    def run_transaction(self, function: Callable, *args, **kwargs) -> Any:

        @firestore.transactional
        def transactional_function(transaction: Transaction, *args, **kwargs):
            return function(*args, **kwargs)

        return transactional_function(self._transaction, *args, **kwargs)

    def get_transaction(self) -> AbstractTransaction:
        return FirestoreTransaction(self._transaction)
```

Pro práci s transakcemi jsem musel upravit také rozhraní repozitářů tak, aby jimi nabízené operace umožňovaly být vykonány i v rámci nějaké dodané transakce. Tedy jsem například metodě určené ke čtení z repozitáře přidal nový nepovinný parametr, kterým se metodě dá předat transakční objekt (viz výpis kódu 5.8) obsahující doposud proběhlý transakční kontext.

5.7 Popis nasazení

Výsledný herní backend byl nasazen na servery platformy Firebase. Toto bylo učiněno v rámci soukromého projektu, ke kterému jsme měli s Vojtěchem Mičkou během práce oba přístup.

Nasazení mého backendu na cílovou platformu Firebase probíhalo jednoduchým způsobem. Nebylo potřeba žádných složitých konfigurací běhového prostředí. Celý projekt se tak dá nasadit z operačního systému Linux pomocí jednoho příkazu.

Kapitola 6
Testování

Tato kapitola se zabývá testováním funkčního prototypu backendové části karetní hry. Prototyp jsem otestoval jednotkovými, integračními a uživatelskými testy.

6.1 Jednotkové testy

Jednotkové testy, které jsem pro svou implementaci vytvořil, se nacházejí v přiloženém projektu v balíčku `src.tests.unit_tests`. Otestoval jsem jimi servisní třídy obsažené ve vrstvě aplikační logiky a doménové modely. Pro jejich tvorbu jsem využil testovací framework pytest [23], který je pro jazyk Python podporován. V rámci jednotkových testů jsem využil metodu mockování závislostí jednotlivých tříd na jiné třídy. Pro mockování jsem využil plugin pytest-mock [24] ve frameworku pytest.

Jednotkové testování mi pomohlo v mé implementaci odhalit například následující druhy chyb či možnosti pro zlepšení:

- chybné podmínky v příkazu if,
- chybné upravování některých objektů v cyklech,
- zlepšit na některých místech návratové hodnoty pro lepší testovatelnost.

6.2 Integrační testy

Integrační testy, které jsem pro backendovou část aplikace vytvořil, se nacházejí v přiloženém projektu v balíčku `src.tests.integration_tests`. Těmito testy jsem otestoval metody servisních tříd, které jsou využívány jednotlivými kontroléry. Jedná se například o metodu zpracovávající hráčův tah v zápase nebo o metodu, která zajišťuje párování hráčů ve frontě pro tvorbu zápasů. Pro vytvoření integračních testů jsem využil opět framework pytest jako pro jednotkové testy. Jako databázi jsem pro účely těchto testů využil a implementoval repozitáře, které neukládají do databáze Cloud Firestore, ale pouze do dočasné paměti za běhu programu.

Integrační testy mi pomohly odhalit například problém s výchozími argumenty v jazyce Python.

6.3 Uživatelské testy

Celý herní backend byl otestován společně s frontendem mého kolegy jako ucelený systém. Uživatelského testování se zúčastnilo celkem 22 lidí. Během něj bylo vyzkoušeno, zda jednotlivé funkcionality implementované v backendu opravdu fungují podle představ frontendové části aplikace.

V rámci uživatelského testování byly v backendové části aplikace objeveny například tyto problémy:

- chybná podmínka v příkazu if při automatickém mazání hráčů spárovaných do zápasu, kteří ale nepotvrdili jeho start;
- klientem neočekávaná návratová hodnota při volání cloudové funkce.

Kapitola 7

Spolupráce v týmu

S mým kolegou Vojtěchem Mičkou jsme při vývoji hry velmi úzce spolupracovali. Naše spolupráce byla v průběhu práce založena na počátečním rozlišení rolí, kdy já jsem převzal odpovědnost za vývoj backendu a Vojtěch se soustředil na frontend. Tento přístup nám umožnil rozdělit si práci a zaměřit se na specifické části hry.

Klíčovým prvkem naší spolupráce byla schopnost pravidelné komunikace. Vzhledem k tomu, že jsme pracovali na různých částech aplikace, byla nezbytná občasná synchronizace a diskuze k řešení problémů. K tomu nám sloužily zejména online hovory, které jsme plánovali podle aktuálních potřeb. Během těchto hovorů jsme diskutovali o aktuálním pokroku, řešili technické výzvy a synchronizovali naše úkoly tak, aby obě části aplikace byly vzájemně kompatibilní.

Kapitola 8
Závěr

Cílem celé této práce bylo zejména vytvořit funkční prototyp serverové části sběratelské karetní hry, na který bude možno napojit klientskou aplikaci, jejíž vývoj je náplní bakalářské práce mého kolegy Vojtěcha Mičky. Cílem teoretické části práce bylo popsat zmíněnou sběratelskou karetní hru, jejíž koncept vymyslel právě kolega Vojtěch Mička. Navíc bylo cílem popsat technologie použité pro implementaci řešení serverové části karetní hry. Oba zmíněné cíle byly splněny. Koncept sběratelské karetní hry jsem stručně popsal a pro více detailních informací o ní jsem odkázal na bakalářskou práci kolegy Vojtěcha Mičky s názvem *Frontend karetní hry pro OS Android*. Pro vývoj prototypu backendu hry byla zvolena platforma Google Firebase, jejímuž popisu se v práci věnuji. Následně jsem popsal také služby, které jsem pro vývoj serverové části karetní hry na této platformě využil. Zmínil jsem zde také, že jsem si pro vývoj herního backendu vybral programovací jazyk Python a uvedl důvody proč.

Cílem praktické části této práce bylo zejména navrhnout a vytvořit funkční prototyp serverové části sběratelské karetní hry. V rámci něj bylo cílem specifikovat požadavky kladené na aplikaci. Nakonec jsem si kladl za cíl vytvořený funkční prototyp aplikace vhodně otestovat a nasadit na cílovou platformu. Všechny tyto požadavky jsem v rámci možností splnil. V rámci návrhu byl například vytvořen databázový model pro použitou NoSQL databázi Cloud Firestore, popsána architektura aplikace, rozhraní vystavované klientovi, stěžejní aktivity probíhající v serverové části hry a další. Na těchto základech byl poté prototyp aplikace vyvinut. Při vývoji jsem se nejvíce zdržel při hledání řešení neideálních databázových transakcí, které nabízí použitá databázová technologie. Nakonec jsem byl nucen učinit v implementaci kompromis, který není sice ideální, ale nynějšímu prototypu backendu dostačuje. Více danou problematiku vysvětluji v příslušné podkapitole. Cíl ohledně otestování vytvořeného funkčního prototypu jsem v rámci časových možností splnil tím, že jsem vytvořil pro vrstvu aplikační logiky a doménové modely jednotkové testy. Část servisních tříd jsem otestoval také integračními testy. Testování prototypu proběhlo také na úrovni uživatelského testování společně s klientskou částí, kterou vytvořil kolega. Cíl nasadit aplikaci na cílovou platformu jsem také v rámci práce úspěšně splnil. Vzhledem ke zvolené technologii pro vývoj bylo nasazení na servery platformy Firebase nenáročné tak, jak si klade za cíl využitý koncept bezserverové architektury.

Příloha A

Rozhraní cloudových funkcí

V této příloze popisuji rozhraní jednotlivých cloudových funkcí, které jsou vystavovány pro volání ze strany klientské aplikace. U každé cloudové funkce uvádím parametry, které jsou pro její zavolání potřeba včetně jejich datových typů, a také kódy odpovědí v rámci platformy Google Firebase, které zavolání dané cloudové funkce může vracet. Pokud zavolání dané cloudové funkce proběhne bez problémů, je klientské části vrácen slovníkový datový typ ve tvaru: {„status“: „OK“}. V následujících výpisech bude nahrazen pouze kódem odpovědi „OK“.

A.1 add_player_to_queue

Tato cloudová funkce přidá hráče s daným identifikátorem do fronty pro tvorbu zápasů.

A.1.1 Parametry požadavku

user_id: string identifikátor hráče, který by měl být zařazen do fronty pro čekání na zápas (získání přes autentizaci)

deck_id: string identifikátor balíčku karet, se kterými bude hráč v nalezeném zápase hrát

A.1.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.2 delete_player_from_queue

Tato cloudová funkce odebere hráče s daným identifikátorem z fronty pro tvorbu zápasů, pokud ještě nebyl spárován do zápasu s jiným hráčem.

A.2.1 Parametry požadavku

user_id: string identifikátor hráče, který by měl být odebrán z fronty pro čekání na zápas (získání přes autentizaci)

A.2.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.3 acknowledge_to_start_match

Tato cloudová funkce potvrdí hernímu backendu ze strany frontendu, že daný hráč, pro kterého již byl nalezen zápas, souhlasí se startem zápasu.

A.3.1 Parametry požadavku

user_id: string identifikátor hráče, který čeká ve frontě na start zápasu a byl již spárován do zápasu s jiným hráčem; hráč s tímto identifikátorem potvrdí, že daný zápas může začít (získání přes autentizaci)

A.3.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.4 make_move_in_match

Tato cloudová funkce vykoná tah daného hráče v daném zápasu.

A.4.1 Parametry požadavku

user_id: string identifikátor hráče, který vykonává tah (získání přes autentizaci)

match_id: string identifikátor zápasu, který daný hráč hraje

played_card_id: string identifikátor karty, kterou chce daný hráč odehrát

A.4.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.5 auto_match_ender

Tato cloudová funkce zkontroluje neaktivitu při odehrávání tahů v zápase, pokud je zápas příliš dlouho neaktivní, je nakonec ukončen a vyhodnocen.

A.5.1 Parametry požadavku

user_id: string identifikátor hráče, který chce zkontrolovat neaktivitu v zápase (získání přes autentizaci)

match_id: string identifikátor zápasu, který chce daný hráč zkontrolovat

A.5.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.6 update_cards_in_deck

Tato cloudová funkce upraví seznam karet, které obsahuje balíček karet daného hráče.

A.6.1 Parametry požadavku

user_id: string identifikátor hráče, který upravuje svůj balíček karet (získání přes autentizaci)

card_ids: list[string] list identifikátorů jednotlivých karet, které má mít hráč nově ve svém balíčku

deck_id: string identifikátor existujícího balíčku karet daného hráče

A.6.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.7 update_name_of_deck

Tato cloudová funkce upraví název balíčku karet daného hráče.

A.7.1 Parametry požadavku

user_id: string identifikátor hráče, který upravuje jméno svého balíčku karet (získání přes autentizaci)

name: string nové jméno pro balíček karet daného hráče

deck_id: string identifikátor existujícího balíčku karet daného hráče

A.7.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.8 add_new_user

Tato cloudová funkce zaregistruje nového uživatele do herního backendu. Vytvoří se nový uživatelský profil pro daného hráče.

A.8.1 Parametry požadavku

user_id: string identifikátor hráče, který se registruje do hry (získání přes autentizaci)

nickname: string přezdívka nového hráče

A.8.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.9 update_user_nickname

Tato cloudová funkce upravuje přezdívku daného hráče v rámci jeho uživatelského profilu.

A.9.1 Parametry požadavku

user_id: string identifikátor hráče, který mění svou přezdívku (získání přes autentizaci)

nickname: string nová přezdívka hráče

A.9.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.10 buy_card

Tato cloudová funkce zakoupí kartu pro daného uživatele, pokud má dostatek herní měny a podobně.

A.10.1 Parametry požadavku

user_id: string identifikátor hráče, který si chce koupit danou kartu (získání přes autentizaci)

card_id: string identifikátor karty, kterou si chce daný hráč zakoupit

A.10.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.11 rent_card

Tato cloudová funkce pronajme kartu pro daného uživatele, pokud má dostatek herní měny a podobně.

A.11.1 Parametry požadavku

user_id: string identifikátor hráče, který si chce pronajmout danou kartu (získání přes autentizaci)

card_id: string identifikátor karty, kterou si chce daný hráč pronajmout

A.11.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.12 buy_game_currency_item

Tato cloudová funkce zakoupí pro daného hráče herní měnu z předdefinovaného balíčku herní měny.

A.12.1 Parametry požadavku

user_id: string identifikátor hráče, který si chce zakoupit herní měnu (získání přes autentizaci)

game_currency_item_id: string identifikátor balíčku herní měny, kterou si chce hráč zakoupit

A.12.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

A.13 change_automatic_card_rent_renewal

Tato cloudová funkce mění nastavení automatického pronájmu dané karty pro daného hráče.

A.13.1 Parametry požadavku

user_id: string identifikátor hráče, který si chce změnit nastavení automatického pronájmu karty (získání přes autentizaci)

card_id: string identifikátor karty, u které chce hráč automatický pronájem změnit

rent_renewal_allowed: bool bool hodnota určující, zda má být automatický pronájem dané karty zapnut, nebo ne

A.13.2 Kódy odpovědí

OK úspěch

UNAUTHENTICATED volání dané funkce nebylo autentizováno

INVALID_ARGUMENT očekávané parametry nebyly v dobrém formátu

FAILED_PRECONDITION danou operaci nebylo možné vzhledem k aktuálnímu stavu aplikace vykonat

UNKNOWN došlo k chybě při zpracování požadavku

Bibliografie

1. MATTHEWS, Sebastian. *The Rise of Collectible Card Games: A Look into the Gaming Phenomenon* [online]. 2023. [cit. 2024-03-12]. Dostupné z: `https://sebastian-matthews.medium.com/the-rise-of-collectible-card-games-a-look-into-the-gaming-phenomenon-7eea0fa55f93`.
2. MIČKA, Vojtěch. *Frontend karetní hry pro OS Android*. 2024. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií.
3. AGREDA, Victor. *What is Firebase?* [online]. 2023. [cit. 2024-03-19]. Dostupné z: `https://developer.oracle.com/learn/technical-articles/what-is-firebase`.
4. GOOGLE. *Get started at no cost, then pay as you go.* [online]. 2024. [cit. 2024-05-15]. Dostupné z: `https://firebase.google.com/pricing`.
5. GOOGLE. *Cloud Functions for Firebase* [online]. 2024. [cit. 2024-04-27]. Dostupné z: `https://firebase.google.com/docs/functions`.
6. AMAZON WEB SERVICES. *Building Applications with Serverless Architectures* [online]. c2024. [cit. 2024-03-26]. Dostupné z: `https://aws.amazon.com/lambda/serverless-architectures-learn-more`.
7. GOOGLE. *Call functions from your app* [online]. 2023. [cit. 2024-04-09]. Dostupné z: `https://firebase.google.com/docs/functions/callable?gen=2nd`.
8. GOOGLE. *Choose a Database: Cloud Firestore or Realtime Database* [online]. 2024. [cit. 2024-04-05]. Dostupné z: `https://firebase.google.com/docs/database/rtdb-vs-firestore`.
9. GOOGLE. *Cloud Firestore* [online]. 2024. [cit. 2024-04-05]. Dostupné z: `https://firebase.google.com/docs/firestore`.
10. MONGODB. *SQL to MongoDB Mapping Chart* [online]. c2023. [cit. 2024-04-09]. Dostupné z: `https://www.mongodb.com/docs/manual/reference/sql-comparison`.
11. MONGODB. *What Is MongoDB?* [online]. c2024. [cit. 2024-04-09]. Dostupné z: `https://www.mongodb.com/company/what-is-mongodb`.
12. GOOGLE. *Firebase Realtime Database* [online]. 2024. [cit. 2024-04-05]. Dostupné z: `https://firebase.google.com/docs/database`.

13. BAELDUNG. *Layered Architecture* [online]. 2021. [cit. 2024-04-09]. Dostupné z: https://www.baeldung.com/cs/layered-architecture.

14. SARDAR KHAN. *Understanding Dependency Injection: A Powerful Design Pattern for Flexible and Testable Code* [online]. 2023. [cit. 2024-04-14]. Dostupné z: https://medium.com/@sardar.khan299/understanding-dependency-injection-a-powerful-design-pattern-for-flexible-and-testable-code-5e1161dd37dd.

15. THESUNPANDEY. *Repository Design Pattern* [online]. 2024. [cit. 2024-04-14]. Dostupné z: https://www.geeksforgeeks.org/repository-design-pattern.

16. GEEKSFORGEEKS. *Factory method Design Pattern* [online]. 2024. [cit. 2024-05-08]. Dostupné z: https://www.geeksforgeeks.org/factory-method-for-designing-pattern.

17. ROTOLO, Domenico. *Firebase Multiplayer Game & Matchmaking in Unity!* [online]. 2020. [cit. 2024-05-08]. Dostupné z: https://medium.com/firebase-developers/firebase-multiplayer-game-matchmaking-in-unity-1d2d04989426.

18. MICROSOFT. *Code Editing. Redefined.* [online]. 2024. [cit. 2024-03-06]. Dostupné z: https://code.visualstudio.com/.

19. MICROSOFT. *Python* [online]. 2024. [cit. 2024-03-06]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=ms-python.python.

20. MICROSOFT. *Pylance* [online]. 2024. [cit. 2024-03-06]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance.

21. *Git* [online]. 2024. [cit. 2024-03-06]. Dostupné z: https://git-scm.com/.

22. GOOGLE. *Transactions and batched writes* [online]. 2024. [cit. 2024-04-15]. Dostupné z: https://firebase.google.com/docs/firestore/manage-data/transactions#python.

23. KREKEL, Holger; PYTEST-DEV TEAM. *pytest: helps you write better programs* [online]. c2015. [cit. 2024-05-14]. Dostupné z: https://docs.pytest.org/en/8.2.x/index.html.

24. OLIVEIRA, Bruno. *pytest-mock* [online]. c2022. [cit. 2024-05-14]. Dostupné z: https://pytest-mock.readthedocs.io/en/latest/#.

Obsah přiloženého repozitáře

- `README.txt` ..... stručný popis obsahu repozitáře
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `thesis.pdf` ..... text práce ve formátu PDF