Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Webová aplikace na sběr a analýzu dat předmětů z inventáře uživatele herní platformy Steam

Matěj Varga

Bakalářská práce
2023

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Matěj Varga**
Osobní číslo:	**I19154**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Webová aplikace na sběr a analýzu dat předmětů z inventáře uživatele herní platformy Steam**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem této práce je vytvoření webové aplikace na správu a analýzu virtuálních předmětů herní platformy Steam. Hlavní funkcí aplikace bude sběr dat z dostupných API za účelem analýzy vývoje ceny jednotlivých předmětů například ze hry Rust. Dalšími funkcionalitami aplikace budou například tvorba a ukládání kolekcí předmětů nebo porovnávání změny dat v časovém období. Aplikace bude schopna poskytnout uživatelské rozhraní potřebné k přístupu rozšířených funkcionalit. Pro vytvoření webové aplikace budou využity následující technologie: HTML 5, CSS3, Javascript ES6, JSON, React, Node.js, Express.js, MongoDB, Git.

Rozsah pracovní zprávy: **min. 30 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

React: Quickstart Step-By-Step Guide To Learning React Javascript Library. Createspace Independent Publishing Platform, 2017. ISBN 1976210232.
LAURENČÍK, Marek. *Tvorba www stránek v HTML a CSS*. Praha: Grada Publishing, 2019. Průvodce (Grada). ISBN 978-80-271-2241-7.

Vedoucí bakalářské práce: **Ing. Jan Panuš, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **16. prosince 2022**
Termín odevzdání bakalářské práce: **12. května 2023**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2023

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 15. 12. 2023

Matěj Varga

PODĚKOVÁNÍ

Chtěl bych poděkovat svému vedoucímu Ing. Janu Panušovi, Ph.D. za vedení mé práce. Dále bych chtěl také poděkovat mým kamarádů, rodině, a hlavně své přítelkyni, že se mnou měli strpení, stáli po mém boku, a že mě vždy podporovali v nejtěžších chvílích.

ANOTACE

Bakalářská práce se zabývá vývojem moderní webové aplikace, která sbírá volně dostupná data týkající se Steam marketingu z konkrétní hry a poskytující vizuální výstup, který splňuje očekávání pro analýzu sbíraných dat. Data jsou uložena v NoSQL dokumentové databázi MongoDB, ze které jsou vybírána za pomoci serverové části aplikace. Serverovou složku zde zajišťuje Node.js společně s Express.js a konečný vizuální výstup je zpracován jednou z nejpoužívanějších JavaSriptových knihoven React.js.

KLÍČOVÁ SLOVA

Webová aplikace, MongoDB, Node.js, Express.js, React.js, JavaScript, TypeScript, REST API, Steam

ANNOTATION

The bachelor's thesis deals with the creation of a web application for the collection and analysis of data from items in the inventory of the user of the Steam gaming platform. The data comes from available APIs and is stored in a MySQL database, from which it is selected based on the queries received. The application monitors the development of the price of individual items and provides the user with a visual overview of the required data.

KEYWORDS

Web application, MongoDB, Node.js, Express.js, React.js, JavaScript, TypeScript, REST API, Steam

OBSAH

Seznam obrázků a tabulek **9**
Seznam zkratek **10**
Úvod **12**
1 Analýza požadavků **13**
1.1 Funkční požadavky 13
1.2 Nefunkční požadavky 14
2 Teoretický základ pro vývoj webové aplikace **15**
2.1 Architektura webové aplikace 15
2.2 Application Programming Interface (API) 16
2.2.1 SOAP 17
2.2.2 REST 19
2.2.3 GraphQL 22
2.3 Výběr databáze 23
2.3.1 Relační databáze (SQL) 23
2.3.2 Nerelační databáze (NoSQL) 24
2.3.3 MongoDB 25
2.4 JavaScript 26
2.5 TypeScript 26
2.6 Node.js 26
2.7 React.js 27
2.8 Redux 27
2.9 Git 28
3 Návrh aplikace **29**
3.1 Zpracovávaná data 29
3.2 Návrh pro backend 29
3.2.1 Struktura serverové části 29
3.2.2 API design 30
3.2.3 Uživatelské ověření 31
3.3 Návrh databáze 31
3.4 Návrh pro frontend 32
3.4.1 UI/UX Design 32
3.4.2 Dashboard 34
3.4.3 Showcase 35
3.4.4 Profil 36
3.4.5 Detail předmětu 37
3.4.6 Interakce s backendem 38
4 Implementace aplikace **39**
4.1 Inicializace projektů 39
4.2 Struktura projektů 40
4.3 Implementace serverové části 42

4.3.1 Nastavení Express 42
4.3.2 Routy 42
4.3.3 Modely 43
4.3.4 Kontroléry 44
4.4 Implementace klientské části 46
4.4.1 Nastavení stavového uložiště Redux 46
4.4.2 Routing a rozložení 47
4.4.3 Klientská databáze 48
4.4.4 Příklady implementovaných komponent 48
5 Testování a budoucnost nasazení 54
Závěr 55
Použitá literatura 56

SEZNAM OBRÁZKŮ A TABULEK

Obrázek 1- Klient-server architektura [4]. (request = požadavek, response= odpověď) 15
Obrázek 2 - Jak funguje API [8]. 16
Obrázek 3 - Struktura SOAP zprávy [12]. (envelope = obálka, header = hlavička, body = tělo) 18
Obrázek 4 - Struktura REST API URL. Zdroj vlastní. 21
Obrázek 5 - Použití GraphQL [24]. 22
Obrázek 6 - Rozvržení uživatelského rozhraní aplikace. Zdroj vlastní. 32
Obrázek 7 - Návrh zobrazení sub-aplikace Dashboar. Zdroj vlastní. 34
Obrázek 8 - Návrh zobrazení sub-aplikace Showcase. Zdroj vlastní. 35
Obrázek 9 - Návrh zobrazení sub-aplikace Profil. Zdroj vlastní. 36
Obrázek 10 - Návrh zobrazení sub-aplikace Detail předmětu. Zdroj vlastní. 37
Obrázek 11 - Struktura backendu. Zdroj vlastní. 40
Obrázek 12 - Struktura frontendu. Zdroj vlastní. 41
Obrázek 13 - Struktura sub-aplikace Dashboard. Zdroj vlastní. 42
Obrázek 14 - Implementace rout pro předměty. Zdroj vlastní. 43
Obrázek 15 - Příklad schéma a modelu inventáře uživatele. Zdroj vlastní. 43
Obrázek 16 - Reprezentace modelu inventáře uživatele v MongoDB Compass. Zdroj vlastní. 44
Obrázek 17 - Příklad primitivního kontroléru vracejícího všechny předměty zvolené hry. Zdroj vlastní. 44
Obrázek 18 - Příklad kontroléru synchronizující všechny předměty ze hry Rust. Zdroj vlastní. 45
Obrázek 19 - Příklad konfugurace Redux store. Zdroj vlastní. 47
Obrázek 20 - Zdrojový kód komponenty <InfoCard />. Zdroj vlastní. 49
Obrázek 21 – Implementace komponenty <InfoCard /> pro vybranou hru. Zdroj vlastní. 49
Obrázek 22 - Zobrazení komponenty <InfoCard /> v různých stylech použitích. Zdroj vlastní. 50
Obrázek 23 – Zdrojový kód komponenty <Items />. Zdroj vlastní. 50
Obrázek 24 - Použití komponenty <Items /> jako přehlídka všech předmětů. Zdroj vlastní. ..51
Obrázek 25 - Použití komponenty <Items /> jako katalog pro výběr předmětů kolekce. Zdroj vlastní. 52
Obrázek 26 - Zobrazení sub-aplikace Dashboard. Zdroj vlastní. 52
Obrázek 27 - Zobrazení sub-aplikace Profil. Zdroj vlastní. 53
Obrázek 28 - Zobrazení sub-aplikace Detail předmětu. Zdroj vlastní. 53

Tabulka 1 - API routy aplikace. Zdroj vlastní. 30

SEZNAM ZKRATEK

API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheet
DBMS	Database Management Systems
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
MUI	Material UI
MVC	Model View Controller
NPM	Node Package Manager
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SPA	Single Page Application
SQL	Structured Query Language
TCP	Transmission Control Protocol
UI	User Interface
URI	Unifrom Resource Identifier
URL	Uniform Resource Locator

UX	User Experience
XML	Extensible Markup Language

ÚVOD

Herní platforma Steam si za období své existence získala dominantní postavení ve světě distribuce digitálních kopií her, čímž si také vybudovala velkou komunitu hráčů. Platforma nabízí svým uživatelům multiplayerové a komunikační zázemí. Hráči se mohou navzájem propojovat na základě „seznamu kamarádů“ a společně sdílet herní zážitky. Ať už se jedná o záznamy, statistiky, úspěchy nebo nejrůznější doplňky do her. Steam svým uživatelům také nabízí navzájem si herní doplňky, tzv. předměty (ang. items) vyměňovat a prodávat. To vede k rozvíjení obchodování mezi komunitou hráčů. Cena zmíněných předmětů se odvíjí od různých kritérií.

Téma obchodování s virtuálními předměty mě doslova pohltilo, ale rychle jsem pochopil, že ke správnému a úspěšnému obchodování je zapotřebí mít dobrá data v podobě, která je pochopitelná i pro běžného uživatele.

Zaměřil jsem se na způsob, jak nejefektivněji předat uživatelům aktuální informace o dění v komunitním trhu herní platformy Steam. Jako optimální řešení jsem zvolil vytvoření webové aplikace, která umožní uživatelům snadný a přehledný přístup právě k těmto informacím.

Praktickým výsledkem bakalářské práce tedy bude webová aplikace, která umožní uživatelům platformy Steam sledovat co možná nejpřesnější vývoj ceny jednotlivých předmětů, sledovat globální pohyb celého trhu v rámci zvolené hry, analyzovat data z uživatelských inventářů podle různých požadavků nebo vytvářet uživatelské kolekce předmětů za účelem sledování vývoje hodnoty skupin předmětů.

1 ANALÝZA POŽADAVKŮ

Herní platforma Steam poskytuje ve svých aplikacích agendu komunitního trhu, v níž mohou její uživatelé najít všechny dostupné předměty veškerých her, které jsou určené k obchodování. Komunitní trh slouží v určitých směrech jako předloha základních funkcionalit vytvářené aplikace, jejichž výsledky nemění oficiální data Steamu. Tím jsou na mysli například samotné metody vytváření poptávek a nabídek. Vytvářená aplikace by měla přebírat z komunitního trhu především myšlenku katalogu předmětů, kde si uživatel může jednotlivé předměty rozkliknout pro zjištění informací o vlastnostech předmětu a vývoji jeho ceny v časovém intervalu. Zároveň by aplikace měla implementovat užitečné vlastnosti, které komunitní trh neposkytuje a které by mohli být pro uživatele vhodné při analyzování dat pro úspěšné obchodování. Ty jsou definovány v následujících kapitolách.

1.1 Funkční požadavky

- **Sběr dat** – aplikace, nezávisle na jejím používaní, sbírá data týkající se hodnot všech předmětů s hodinovým přírůstkem, je-li rozdílný
- **Jednoduchý přehled relevantních informací** – zpřístupnění důležitých informací přehledným způsobem za účelem snadné analýzy dat
- **Katalog předmětů** – dostupnost všech předmětů vybrané hry na jednom místě s možností filtrování
- **Detail předmětu** – zobrazení relevantních informací o vybraném předmětu
- **Přihlášení uživatele** – možnost přihlášení uživatele do systému prostřednictvím uživatelského účtu z platformy Steam
- **Rozšířené funkcionality** – zpřístupnění rozšířených funkcionalit pro přihlášené uživatele
    - **Uživatelský profil** – systém zajišťuje přehled dostupných informací o uživatelském profilu a poskytuje celkovou hodnotu aktuálního inventáře
    - **Správa vlastních kolekcí** – uživatelé mohou vytvářet, upravovat a mazat vlastní kolekce předmětů, které jim poskytnou skupinový přehled jejich celkové ceny a rychlý přístup k detailu jednotlivých předmětů
    - **Inventář vlastněných předmětů** – přehled všech vlastněných předmětů vybrané hry s možností filtrování

1.2 Nefunkční požadavky

- **Stabilita** – aplikace během svého životního cyklu zajišťuje stabilní přístup ke všem implementovaným funkcionalitám
- **Výkon** – aplikace je zpracována architekturou klient-server, která poskytne uspokojení v otázce zátěže, jak ze strany serveru, tak uživatelského rozhraní
- **Bezpečnost** – aplikace žádným způsobem nezpracovává ani nepřistupuje k přihlašovacím údajům uživatele, rozšířené funkce skrývá za uživatelským ověřením
- **Přehlednost** – aplikace zajišťuje přehledné zobrazení relevantních dat, které slouží jako podpůrný nástroj pro analýzu
- **Rozšiřitelnost** – systém je vyvíjen s ohledem na rozšíření přidáním více her a nárůstem celkového počtu zpracovávaných informací

2 TEORETICKÝ ZÁKLAD PRO VÝVOJ WEBOVÉ APLIKACE

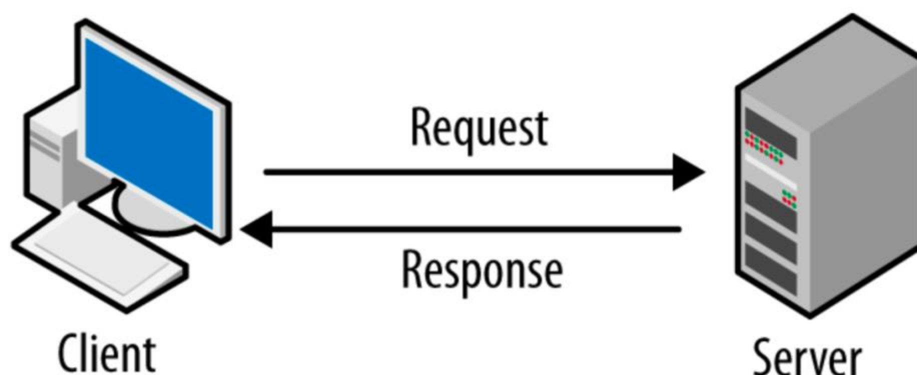
Tato kapitola se zabývá teoretickým popisem použitých technologií, které jsou nedílnou součástí pro tvorbu moderní webové aplikace. Ze začátku popisuje rozdělení aplikace na jednotlivé části (klientská a serverová) z hlediska architektury a komunikace mezi nimi skrze API (Application Programming Interface). Dále se zaměřuje na otázku ohledně výběru databázového systému a popisem dokumentové open source databáze MongoDB. Nakonec definuje použité programovací jazyky a jejich frameworky nebo knihovny pro vytvoření klientského i serverového prostředí.

2.1 Architektura webové aplikace

Architektura webové aplikace představuje plánování současných interakcí mezi různými komponentami, instancemi databáze, středními vrstvami, uživatelskými rozhraními a servery v rámci samotné aplikace. Je možné ji též charakterizovat jako strukturu, která logicky definuje propojení mezi serverovou a klientskou částí s cílem zajistit lepší uživatelskou přístupnost [1].

Velká část internetu je založena na architektuře klient-server, kde se klienti v zastoupení uživatelských zařízení dotazují prostřednictvím sítě centrálně umístěných serverů na zdroje nebo služby místo toho, aby komunikovali mezi sebou. V obecném principu, koncová uživatelská zařízení, jako jsou notebooky, chytré telefony a stolní počítače, jsou považována za zákazníky určitého serveru, který jim na základě jejich požadavku poskytuje své služby [2].

Většina webových aplikací se skládá ze dvou odlišných částí (subprogramů). Klientská část (client-side) obstarává uživatelské rozhraní a poskytuje uživateli interakci s aplikací. Serverová část (server-side) reaguje na požadavky z klientské části, komunikuje s databází a servery třetích stran [3].



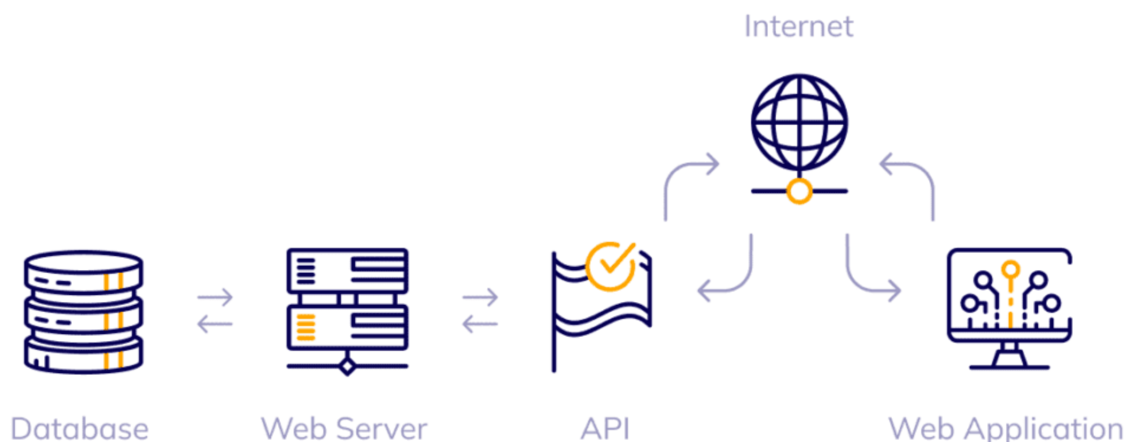
Obrázek 1- Klient-server architektura [4].
(request = požadavek, response= odpověď)

Výše zmíněné rozdělení vysvětluje, kde daná část aplikace operuje. Nicméně v běžném programování se často setkáváme s klasifikací podle funkcí, které tyto části vykonávají:

- **Frontend** – vizuální a interaktivní část aplikace, která se stará o to, jak aplikace vypadá a jak s ní uživatelé interagují. Mimo jiné zajišťuje také první validaci uživatelských vstupů, navigaci napříč uživatelským rozhraní nebo optimalizaci zobrazení pro různá zařízení [5]
- **Backend** – část aplikace, kterou uživatel nevidí. Zajištuje funkčnost webové aplikace, stará se o database management nebo vytváří aplikační rozhraní (API) [6]

2.2 Application Programming Interface (API)

Rozhraní pro programování aplikací (Application Programming Interface), známé zkratkou API, představuje sadu pravidel umožňujících vzájemnou komunikaci a přenos dat mezi různými částmi softwaru. Vývojáři využívají API k propojení jednotlivých úseků kódu s cílem vytvářet aplikace, které jsou silné, odolné, bezpečné a schopné splnit požadavky uživatelů. Ačkoliv nejsou viditelná, API jsou všudypřítomná neustále pracují v pozadí a poskytují energii pro digitální zážitky, které jsou klíčové pro náš moderní život [7].



Obrázek 2 - Jak funguje API [8].

API jsou mimořádně všestranná a podporují širokou škálu případů použití. Jedním z nejčastějších použití je integrace s interními a externími systémy, kde API může sloužit k propojení aplikací a databází v rámci jedné organizace nebo systémy třetích stran, které nejsou součástí vnitřní infrastruktury. Dalším případem je připojení zařízení IoT, které se skrze API připojují ke cloudovým uložištím, nebo mezi sebou navzájem a učiní je tak efektivní v rámci zamýšleného využití. API jsou také klíčovým prvkem při implementaci architektur založených na mikroslužbách. V těchto architekturách jsou aplikace vytvářeny jako soubor

malých služeb, které efektivně komunikují pomocí privátních API. Mikroslužby mohou být spravovány, nasazovány a zajišťovány nezávisle, což umožňuje vývojářům škálovat (rozšiřovat) své systémy spolehlivě a zároveň nákladově efektivně [7].

API lze rozdělit do tří základních typů podle přístupnosti, jehož správné určení je klíčové pro efektivní propojení softwarových systémů. *Veřejné API* poskytují široké možnosti přístupu k funkcím a datům bez významných omezení, čímž podporují inovativní integrace a scénáře použití. Naopak *privátní* či *interní API* jsou zaměřena na uzavřené uživatele v konkrétní organizaci a přinášejí vyšší stupeň bezpečnosti pro citlivá data a specifické firemní funkce. Mezi tímto rozdělením se nachází *partner API*, sloužící k důvěrnému sdílení dat mezi společnostmi za účelem obchodní spolupráce, přičemž klade důraz na ochranu soukromí. Každý z těchto typů API nese své vlastní charakteristiky a je navržen s ohledem na různorodé scénáře použití a uživatelské potřeby, což ovlivňuje jejich dostupnost, zabezpečení a povahu poskytovaných dat a funkcí [9].

Při programování rozhraní mezi dvěma stranami pomocí API jsou striktní směrnice, známé jako API protokoly, které jsou uplatňovány k regulaci interakcí.

Tyto pravidla zvyšují efektivitu při výměně dat mezi různými aplikacemi prostřednictvím standardizovaných komunikačních forem. Mezi požadavky stanovené API protokoly patří specifikace formátování pro výměnu požadavků/odpovědí nebo povolené typy dat, která lze sdílet, autentizační postupy a bezpečnostní opatření pro bezpečný přenos informací. Následování těchto standardů zaručuje konzistentní interakce a spolehlivý výkon aplikace [10].

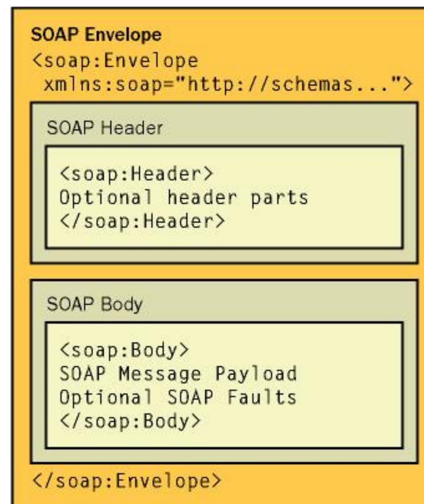
2.2.1 SOAP

SOAP (Simple Object Access Protocol) je standard umožňující komunikaci distribuovaných prvků aplikací. SOAP je velmi snadný protokol pro vytváření webových rozhraní, který podporuje širokou škálu komunikačních protokolů, včetně HTTP, SMTP a TCP. Určuje způsob zpracování zpráv, je velice flexibilní, nezávislý a umožňuje vývojářům definovat funkce a moduly [11].

SOAP zprávy jsou XML dokumenty, které se skládají ze tří základních stavebních bloků.

- **Obálka** (Envelope) – SOAP obaluje veškerá data ve zprávě a identifikuje XML dokument jako SOAP zprávu
- **Hlavička** (Header) – obsahuje další informace o SOAP zprávě. Tyto informace mohou být například ověřovací údaje, které používá volající aplikace

- **Tělo** (Body) – zahrnuje detaily skutečné zprávy, které mají být odeslány z webové služby volající aplikaci. Tato data zahrnují informace o volání a odpovědi



Obrázek 3 - Struktura SOAP zprávy [12].
(envelope = obálka, header = hlavička, body = tělo)

Čtvrtým volitelným stavebním blokem je Zpráva o chybě (Fault message). Pokud je vygenerována SOAP chyba, je vrácena jako HTTP 500 (Internal Server Error). Tyto zprávy o chybě obsahují chybový kód, řetězec, aktéra a detail samotné chyby [11].

S ohledem na pevnou strukturu XML jsou data přenášená pomocí SOAP API velmi podrobná a mohou vypadat více komplexněji, než je tomu například u modernější architektury REST. Během jejich přenosu jsou jednotlivé těsně zapouzdřené položky obaleny identifikačními značkami a jsou uspořádány pevně podle přístupového vzoru a struktury ve vyhrazeném souboru, což činí SOAP výrazně řízeným protokolem [13].

SOAP je protokolem, který definuje přísná pravidla komunikace. Má několik přidružených standardů, které kontrolují každý aspekt výměny dat.

- **Web Services Security** (WS-Security) – specifikuje bezpečnostní opatření, jako je používaní jedinečných identifikátorů (tokenů)
- **Web Services Addressing** (WS-Addressing) – vyžaduje zahrnutí informací o směrování jako metadata
- **WS-ReliableMessaging** – standardizuje zacházení s chybami v komunikaci SOAP
- **Web Services Description Language** (WSDL) – popisuje rozsah a funkci SOAP webových služeb

Vzhledem k jeho striktním pravidlům komunikace je SOAP výhodnou volbou pro vývoj privátních API. Kromě toho má SOAP zabudovanou podporu pro atomičnost, konzistenci, izolaci a odolnost (ACID), což může být pro systémy s vysokými nároky na integritu dat zásadní. Tento typ požadavků se typicky vyskytuje u finančních transakcí, kde je například nutné, aby selhal celý soubor aktualizací v případě chyby v jedné z nich [14].

2.2.2 REST

Architektura softwaru, která je známá jako Representational State Transfer (REST), stanovuje podmínky, jak by mělo API fungovat. REST byl vytvořen jako směrnice pro správu komunikace v rozsáhlých sítích, například na internetu, dále umožňuje podporu vysokého výkonu a spolehlivé komunikace na velké škále. REST-based architektura usnadňuje implementaci a úpravy, což přispívá k viditelnosti a přenositelnosti mezi různými platformami pro jakýkoli systém API.

API mohou být navrhována vývojáři s využitím různých architektur. API řídící se architekturou REST se nazývají REST API. Webové služby, které využívají REST architekturu, jsou označovány jako RESTful webové služby. Termín RESTful API obvykle zahrnuje RESTful webové API, ale oba termíny REST API a RESTful API lze používat vzájemně [15].

REST vychází z několika omezení a zásad, které napomáhají vytvořit jednoduché, škálovatelné a bez stavové řešení v návrhu.

První zásadou pro návrh REST služby je jednotné rozhraní. To pomáhá zjednodušit celkovou architekturu systému, zlepšuje přehlednost interakcí a umožňuje každé části se nezávisle vyvíjet. Jednotného rozhraní lze dosáhnout dodržováním následujících doporučení.

- **Identifikace zdrojů** – rozhraní by mělo jednoznačně identifikovat každý zdroj zapojený v interakci mezi klientem a serverem
- **Manipulace se zdroji pomocí reprezentací** – zdroje by měly mít jednotné reprezentace v odpovědi serveru a spotřebitelé API by měli tyto reprezentace využívat k úpravě stavu zdroje na serveru
- **Samo výstižné zprávy** – každá reprezentace zdroje by měla nést dostatečné informace k popisu zpracování zprávy a poskytnout také informace o dalších akcích, které může klient provádět se zdrojem

- **Hypermédia jako motor stavu aplikace** – klient by měl mít pouze počáteční URI (Uniform Resource Identifiers) aplikace a ta by měla dynamicky řídit všechny ostatní zdroje a interakce s využitím hypertextových odkazů

Jednoduše řečeno, REST stanovuje spolehlivé a sjednocené rozhraní pro komunikaci mezi klienty a servery. Konkrétně REST API založená na HTTP využívají standardní metody (GET, POST, PUT, DELETE atd.) a adresy URI pro identifikaci zdrojů [16].

Další zásadou je oddělení různých aspektů systému na klientskou a serverovou část. Oddělením uživatelského rozhraní od serverových funkcionalit umožňuje její přenositelnost napříč různými platformami a zároveň zjednodušuje práci serverových částí systému, což může zlepšit jeho škálovatelnost a výkon [17].

Všechna RESTful API jsou bez stavová, což znamená, že API na serveru neuchovává žádný kontext klienta, který by přesahoval okamžitý požadavek a informace potřebné k autentizaci. Stav může být uložen v databázi, technicky ale není uchován přímo v samotném API. To znamená, že komunikace musí být bez stavová a stav relace je zcela uchováván na straně klienta. Tím se také zlepšuje transparentnost, protože monitorovací systém nemusí zkoumat více požadavků k pochopení celého kontextu žádosti. Absence potřeby uchovávat stav požadavku přispívá ke zlepšení škálovatelnosti a zajišťuje průběh různých požadavků nezávisle na sobě. Existují i určité nevýhody této jednoduchosti, například riziko snížení výkonu s narůstajícím množstvím opakujících se požadavků na data [18].

Ačkoliv je neobvyklé, že požadavky na stavovém API mají velké režijní náklady, existují i takové případy, kdy je to nevyhnutelné. Zde by mohl nastat problém s opakujícími se požadavky uživatele na stejná data a tím výrazně zatížit serverové služby. REST API by se správně s touto komplikací mělo vypořádat za pomocí mezipaměti. Klient by měl mít možnost lokálně uchovávat nezbytné části dat po stanovenou dobu a v případě jejich potřeby je znovu získat z tohoto uložiště, namísto využití služby serveru [19].

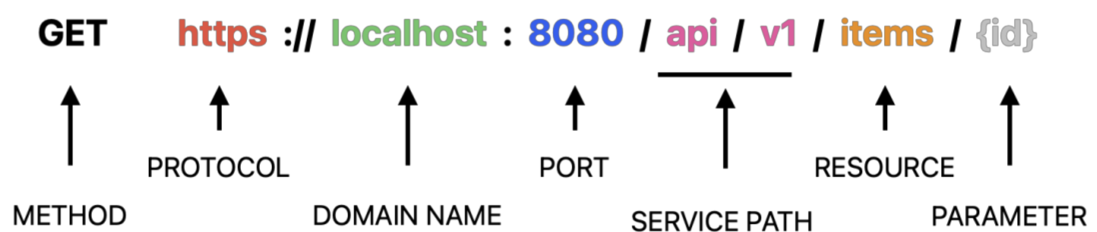
V poslední řadě REST umožňuje vytvářet vrstvenou architekturu systému, ve které může potencionálně několik serverů reagovat na jednu žádost. Klient by neměl snadno identifikovat, který systém reaguje na jeho žádost, zejména pokud je za branou API [20].

Samotné REST API definuje sadu zdrojů a operací, které mohou být prováděny na těchto zdrojích. Tyto operace jsou přístupné z libovolného HTTP klienta, včetně klientského JavaScriptu v prohlížeči. API má základní cestu, obdobnou kořenovému adresáři, která slouží k definici všech zdrojů v rámci tohoto API. Tato cesta může být využita k oddělení různých

verzí stejného API nebo k izolaci mezi různými API. Každý zdroj má určenou sadu operací, které mohou být volány HTTP klientem, a tyto operace jsou identifikovány kombinací cesty ke zdroji a použitou HTTP metodou [21].

- **GET** – Nejpoužívanější metoda, jejímž úkolem je získávání dat ze serveru. Požadované zdroje vrací například ve formátech JSON nebo XML. Operace s metodou GET jsou označovány za bezpečné, což znamená, že by neměly měnit stav jakéhokoliv zdroje na serveru.
- **POST** – Metoda, která se používá k vytvoření nového zdroje na serveru. Skrze operaci s touto metodou lze posílat data ve formátu XML nebo JSON, která mění stav zdroje na serveru, ten následně požadavek zpracuje, vytvoří nový zdroj a odpoví úspěšnou nebo chybovou zprávou a případnými relevantními metadaty.
- **PUT/PATCH** – Metoda sloužící k aktualizaci zdroje na serveru. V principu funguje jako metoda POST s tím rozdílem, že více identických PUT požadavků by mělo vést ke stejnému stavu zdroje, bez ohledu na jejich počet.
- **DELETE** – Metoda, která se používá k odstranění určitého zdroje na serveru.

Jméno každé operace musí být unikátní v rámci celého API a každý zdroj může mít jen jednu operaci definovanou pro konkrétní HTTP metodu. Hierarchická struktura cest může usnadnit porozumění dostupných zdrojů v daném REST API [21].



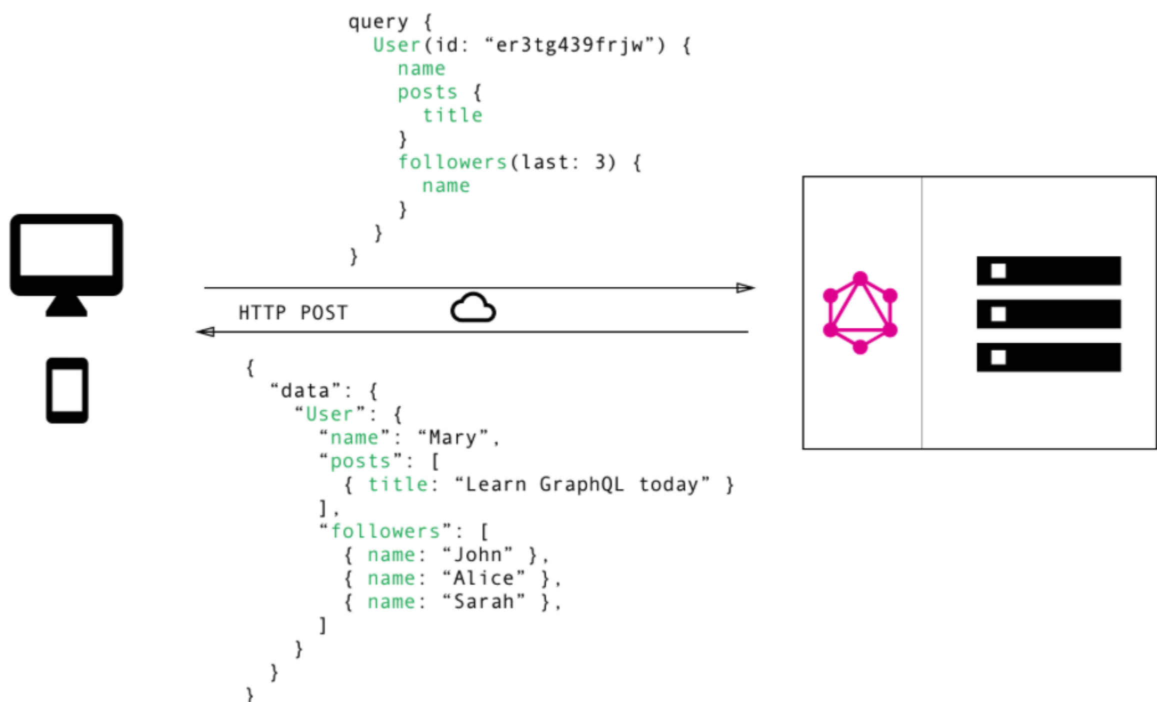
Obrázek 4 - Struktura REST API URL. Zdroj vlastní.

Protokol REST, představený v roce 2000 Royem Fieldingem, výrazně ovlivnil způsob práce s online obsahem. Tento rámec je používán mnoha předními webovými a cloudovými společnostmi, jako jsou Facebook, YouTube, Twitter a Google. Stal se preferovaným systémem pro webové aplikace díky tomu, že umí zpracovat různé typy požadavků a poskytovat data v různých formátech. Jeho schopnost škálovat se, bez ohledu na velikost nebo potřeby softwaru,

dává rostoucím aplikacím možnost rychle reagovat na nárůst požadavků. A integrace s existujícími webovými technologiemi dělá REST API relativně snadno použitelnými k získání dat pouze za pomoci URL zdroje [22].

2.2.3 GraphQL

Jedná se o dotazovací jazyk pro vývoj API a runtime prostředí pro vyhodnocování dotazů se stávajícími daty. Poskytuje komplexní popis dat v API a umožňuje klientům vyžádat si přesně to, co potřebují. Tímto způsobem zajišťuje rychlost a stabilitu aplikací, protože ovládá data, která získává. GraphQL dotazy přistupují nejen k vlastnostem jednoho zdroje, ale také plynule sledují odkazy mezi nimi. Dotazy získávají všechna potřebná data aplikace jedním požadavkem a tím umožňují rychlé a efektivní získávání informací i na pomalých mobilních sítích. Organizace API podle typů a polí zajišťuje přehlednost a minimalizuje chyby při požadavcích dat, což zvyšuje efektivitu aplikací. GraphQL API nabízí neustálý přístup k novým funkcím a podporuje snadnější údržbu serverového kódu, díky jednotnému vývojovému prostředí a schopnosti přidávat nové prvky bez vlivu na existující strukturu. GraphQL vytváří jednotné API napříč celou aplikací, aniž by byl omezen konkrétním úložištěm [23].



Obrázek 5 - Použití GraphQL [24].

GraphQL nalézá využití zejména v aplikacích pro mobilní zařízení, chytré hodinky a IoT zařízení, kde je klíčové šetřit datový tok. To znamená minimalizovat objem přenášených dat mezi těmito zařízeními nebo přes síťové připojení. Dále je GraphQL vhodný pro aplikace, které

potřebují načíst složitá vnořená data jedním dotazem. Příkladem mohou být blogy či sociální sítě, které potřebují načíst příspěvky spolu s jejich komentáři a dalšími podrobnostmi. GraphQL také nachází uplatnění v aplikacích, které kombinují data z různých zdrojů či API do jednoho souhrnného výstupu. To může být například u dashbordů, backendových služeb pro sledování spotřeby nebo nástrojů třetích stran pro analýzu interakcí uživatelů [25].

2.3 Výběr databáze

Pojem databáze označuje v informatice uspořádaný soubor pevně strukturovaných informací nebo dat, většinou uložených elektronicky v počítačovém systému. Takový soubor je dále obvykle řízen systémem pro správu databází (DBMS), který slouží jako rozhraní mezi databází a uživateli nebo programy. Tyto systémy umožňují získávat, aktualizovat a spravovat způsob uspořádání informací a usnadňují přístup k administrátorským operacím databáze jako jsou monitorování výkonu, ladění, zálohování a obnova dat [26].

Výběr správné databáze pro webovou aplikaci je klíčové pro dosažení optimálního výkonu, škálovatelnosti a účinné správy dat. V širokém spektru dostupných možností databází, z nichž každá míří na odlišné potřeby a požadavky, je nezbytné pečlivě zhodnotit několik faktorů [27]. Mezi ně patří například škálovatelnost, která se vypořádává s rostoucím počtem dat, uživatelů nebo požadavků bez ohrožení výkonu a dostupnosti. K řešení škálovatelnosti databáze se může přistupovat dvěma způsoby (vertikálně a horizontálně). Vertikální škálovatelnost znamená přidání výpočetní síly a paměti jednomu serveru, zatímco horizontální škálovatelnost znamená přidání více serverů (tzv. databázových uzlů) pro distribuci pracovní zátěže, přičemž využívá rozdělení dat (sharding) a replikací [28]. Další aspekt výběru databáze se týká záruky, jak konzistence, tak integrity dat. Při současném přístupu více uživatelů nebo procesů k datům v databázi je důležité zajistit konzistenci dat, aby zůstala jednotná napříč všemi systémy. Integrita dat pak zaručuje, že data zůstanou přesná, spolehlivá a bezchybná po celou dobu, kdy jsou v systému uložena [29]. Důležitým faktorem je také podoba zpracovávaných dat, konkrétně v jaké struktuře jsou data ukládána. Zde je na výběr pro rozhodování mezi dvěma hlavními typy databází relační a nerelační, které budou podrobněji rozebrány v následujících kapitolách [30].

2.3.1 Relační databáze (SQL)

Relační databáze představuje soubor informací, který strukturuje data do předem definovaných vztahů. Tato data jsou uchovávána v jedné nebo více tabulkách, kde každý řádek reprezentuje záznam a každý sloupec reprezentuje pole. To umožňuje snadnější vizualizaci a porozumění

tomu, jak různé datové prvky spolu souvisejí. Každý řádek v tabulce relační databáze obsahuje jedinečný identifikátor (primární klíč), který mohou ostatní řádky téže nebo jiné tabulky využít jako odkaz na konkrétní záznam (cizí klíč), čímž vzniká vazba mezi nimi (tzv. relace). Tato struktura umožňuje snadné dotazování a manipulaci s daty pomocí jazyka SQL (Structured Query Language), proto jsou často relační databáze označovány jako SQL databázové systémy. Relační databáze jsou strukturované tak, že logická struktura (tabulky, pohledy nebo indexy) je oddělena od té fyzické. Toto rozdělení zaručuje, že jakákoliv úprava ve fyzické struktuře, například přejmenování databázového souboru nemá vliv na uložená data v samotné databázi. Zásadním prvek relačních databází jsou transakce, které zajišťují celkovou konzistentnost a integritu díky aplikování ACID (atomicita, konzistence, izolace, trvanlivost) [31][32][33].

2.3.2 Nerelační databáze (NoSQL)

Nerelační databáze, často nazývané jako NoSQL (Not Only SQL) databáze, je označení pro jakýkoliv typ databáze, která nepoužívá pro uložení svých dat koncept strukturovaných tabulek, polí a sloupců. Namísto toho používá struktury typu dokument, klíč-hodnota nebo graf, které více vyhovují potřebám moderních aplikací obsahující širokou škálu typů a forem dat.

Dokumentové databáze ukládají data nejčastěji ve struktuře podobné JSON objektu, která jsou pro uživatele snadno čitelná a srozumitelná. V praxi mohou jednotlivé dokumenty představovat objekty v objektově orientovaných programovacích jazycích, čímž značně usnadní práci s daty. Jednotlivé dokumenty jsou také považovány za jednotky, které mohou být distribuovány přes více serverů, což přispívá k schopnosti horizontální škálovatelnosti.

Databáze typu klíč-hodnota je nejjednodušší typ databáze, kde jsou informace uloženy ve formě klíčů a příslušných hodnot. Klíče, které jsou považovány za jedinečné v celé databázi, se používají k získání odpovídajících dat. Tato jednoduchost zaručuje rychlé čtení a zápis dat, ale zároveň omezuje možnosti pro složitější datové požadavky, které nelze efektivně podporovat.

Grafové databáze jsou vysoce specializovanými typy NoSQL databází, které využívají strukturu uzlů pro uchování dat a hran pro zaznamenávání vztahů mezi nimi. Definice vztahů v hranách zajišťuje rychlé vyhledávání souvisejících informací a snadnou flexibilitu pro přidávání nových prvků. Nicméně nejsou ideální pro celkové dotazování databáze, kde vztahy nejsou dobře definovány, a chybí jim standartní dotazovací jazyk, což ztěžuje přechod mezi různými typy grafových databází [34].

NoSQL databáze jsou koncipovány tak, aby vyhovovaly principům distribuované databáze. To znamená, že jejich data jsou často ukládána na více místech, která můžou být geograficky rozdílná a nesdílejí fyzické komponenty. V distribuovaných databázích existují dvě odlišná rozdělení homogenní a heterogenní. V homogenní databázi pracují stroje, uzly nebo servery se stejnými daty, datovými modely a operačním systémem. Dále se dělí do dvou podkategorií. První z nich jsou autonomní databáze, ve kterých uzly pracují nezávisle a pro globální aktualizaci využívají jednu aplikaci. A neautonomní databáze, kde uzly spoléhají na centrální systém pro správu (DBMS) pro koordinaci distribuce dat, komunikaci a veškeré aktualizace. Všechny uzly homogenních distribuovaných databází nabízejí významnou ochranu dat prostřednictvím redundance a zjednodušeného řízení díky podobnosti všech uzlů. Heterogenní databáze naopak představují systémy, ve kterých se stroje, uzly nebo servery mohou lišit v datech, operačních systémech nebo schématech, a dokonce o sobě ani nemusí mít ponětí existence, což může vést k problémům při zpracovávání dotazů a transakcí za cenu větší flexibility [35][36].

Pro zajištění distribuované databáze, která umožňuje uživatelům přístup k relevantním datům bez ovlivňování práce ostatních, se využívá replikace. Tento proces zahrnuje ukládání dat na více než jednom místě nebo uzlu, což v podstatě spočívá v kopírování dat z jedné databáze do druhé. Cílem je poskytnout všem uživatelům stejná data bez jakýchkoli nesrovnalostí [37].

V síťových databázích nebo distribuovaných systémech není prakticky možné dosáhnout zároveň konzistence, dostupnosti a tolerance vůči výpadkům. Tento fakt vede k existenci CAP (Consistency, Availability, Partition tolerance) teorému, který upozorňuje na nutnost kompromisů při výběru NoSQL databázového systémů. To znamená, že v případě rozkladu sítě lze splnit maximálně dvě z těchto vlastností najednou [38].

2.3.3 MongoDB

MongoDB je open-source NoSQL multiplatformní databáze napsaná v jazyce c++, která k ukládání dat a reprezentaci vztahů mezi nimi využívá principů kolekcí a dokumentů. Informace jsou v ní uloženy jako BSON dokument, což je binární formát, kde jedna entita obsahuje nulu, nebo více klíč-hodnota párů. BSON je odvozen od formátu JSON, který je snadno čitelný. MongoDB nabízí podporu pro spoustu přidaných funkcí jako jsou například sekundární indexace (zrychluje vyhledávání napříč databází), využívání JavaScriptových funkcí, podpora MapReduce (nástroj pro agregaci dat), horizontální škálování nebo automatické rozdělování databáze na menší prvky zpracovávané na více strojích (sharding).

Dokumentový model MongoDB umožňuje ukládat různorodá data, jako jsou události, časové řady, geoprostorové souřadnice, text, binární a jiné typy dat. Flexibilita schématu dokumentu umožňuje snadné přizpůsobení struktury dat přidáním nových polí, což zjednodušuje ukládání nových informací do databáze. Horizontální škálování, které MongoDB podporuje, umožňuje automatické rozdělení dat na skupiny serverů. Díky široké podpoře různých typů indexů a dotazů je MongoDB schopná provádět sofistikované analýzy v reálném čase a poskytovat podrobné reporty přímo na místě [39].

2.4 JavaScript

JavaScript, často zkracován jako JS, je jazyk pro psaní skriptů, který se především využívá ve webovém prohlížeči za účelem vylepšit HTML stránky. Jedná se o interpretovaný jazyk, což znamená, že není potřeba ho překládat do strojového kódu. JavaScript dává webovým stránkám interaktivitu a dynamiku, což umožňuje reagovat na události, vytvářet různé vizuální efekty, manipulovat s textem, ověřovat data, pracovat s cookies nebo rozpoznávat prohlížeč, který uživatel používá [40]. Pokud jde o vývoj webových aplikací, JavaScript je především využíván na straně frontendu, kde prostřednictvím oblíbených frameworků (React, Angular a Vue) pomáhá vytvářet moderní uživatelská rozhraní. Nicméně díky svým schopnostem je JavaScript vhodný i pro tvorbu backendových systémů, a to prostřednictvím frameworku Node.js [41].

2.5 TypeScript

TypeScript je rozšířením JavaScriptu, což znamená, že chápe všechny jeho syntaxe, funkce a přidává nové prvky. Jeho hlavní předností oproti JavaScriptu je statické typování, kterým přináší typovou kontrolu probíhající při kompilaci kódu. Tato kontrola zaručuje, že všechny operace pracují se správnými typy dat nebo zda hodnoty určité datové struktury odpovídají deklarovaným datovým typům. V TypeScriptu jsou „type“ a „interface“ dva způsoby, jak definovat strukturu dat a typy v kódu. Interface je klíčovým prvkem pro určení struktury objektů nebo tříd. Slouží k definování konkrétních typů a uspořádání dat, která by měla být obsažena v objektu. Type poskytuje možnost vytvářet aliasy jak pro jednoduché, tak pro komplexní typy, čímž usnadňuje práci s kódem a zlepšuje jeho čitelnost [42][43].

2.6 Node.js

Node.js je univerzální open-source prostředí pro běh JavaScriptu, které se využívá v široké škále projektů. Tento framework využívá jádro Javascriptového enginu V8, které je také používáno v prohlížeči Google Chrome, díky němuž dosahuje vysoké výkonosti. Aplikace

v Node.js běží v jednom procesu, což znamená, že se nevytvářejí nová vlákna pro každý požadavek. Node.js při provádění I/O operací, jako je čtení ze sítě, přístup k databázi nebo k souborovému systému, neblokuje vlákno a neplýtvá výpočetními cykly čekáním, ale obnovuje operace, jakmile přijde odpověď. Tímto způsobem zvládá Node.js zpracovávat tisíce souběžných spojení s jedním serverem bez složité správy souběžnosti vláken [44]. Node.js má široké spektrum využití díky své flexibilitě a efektivitě. Nejběžnějšími případy použití je například pro tvorbu real-time webových aplikací, kolaborativních nástrojů, single-page aplikací (SPA), mikroslužeb nebo API [45].

2.7 React.js

React.js je open-source JavaScriptová knihovna vytvořená společností Meta s cílem zjednodušit tvorbu interaktivních uživatelských rozhraní. Lze si ji představit jako soubor stavebních bloků, kde každý z nich reprezentuje malý, opakovaně využitelný blok HTML kódu. V rámci Reactu jsou vytvářeny aplikace pomocí těchto opakovaně použitelných komponent, které fungují jako samostatné prvky celého uživatelského rozhraní aplikace. Tímto způsobem kombinuje React rychlost a efektivitu JavaScriptu s efektivnější metodou manipulace s DOM (Document Object Model) pro rychlejší vykreslování webových stránek a tvorbu vysoce dynamických a responzivních webových aplikací. React umožňuje vytvářet single-page aplikace (SPA), které načtou jediný HTML dokument při prvním požadavku. Poté, pomocí JavaScriptu, aktualizují pouze potřebné části obsahu stránky. Tento koncept je známý jako routování na straně klienta, protože pro každý nový požadavek uživatele není potřeba kompletně znovu načítat celou stránku. React zachytí požadavek a pouze upraví sekce, které potřebují změnu, bez nutnosti obnovy celé stránky. V Reactu se využívá virtuální DOM, což je kopie skutečného DOM. Virtuální DOM se okamžitě aktualizuje, když dojde ke změně v datovém stavu. React následně porovnává virtuální DOM se skutečným DOM, aby identifikoval, co přesně bylo změněno. Tímto způsobem React efektivně a rychle reaguje na změny v UI komponentách, což vede k dynamickému a rychlému uživatelskému zážitku, aniž by bylo nutné znovu načítat celou stránku [46].

2.8 Redux

Redux je nástroj pro JavaScriptové aplikace, který zajišťuje konzistentní chování aplikací napříč různými prostředími. Umožňuje centrální správu stavu aplikace, čímž zvyšuje předvídatelnost změn a usnadňuje sledování událostí v aplikacích. Své využití najde pro sdílení a aktualizaci dat mezi různými komponentami, přičemž na nich zůstává nezávislý. V rozsáhlých

aplikacích slouží také pro ukládání stavu na jednom místě, ze kterého je sdílen mezi komponentami. Redux se skládá ze tří hlavních komponent (akce, store, reducery). Akce jsou události, které poskytují jediný způsob, jak poslat data z aplikace do storu (uložiště Reduxu). Akce vypadají jako prosté JavaScriptové objekty, které musí obsahovat atribut „type“, který označuje typ akce a objekt „payload“, který obsahuje informace, které by měly být použity ke změně stavu. Akce jsou vytvářeny za pomocí funkce tvůrce akcí a spouštěny skrze metodu *dispatch*, která odesílá akci do uložiště. Uložiště Reduxu (store) je JavaScriptový objekt, který uchovává stav aplikace, ke kterému Redux umožňuje připojit jednotlivé komponenty a jediný způsob, jak ho lze změnit je prostřednictvím akcí. Reducery jsou funkce, které přjímají aktuální stav aplikace, provedou akci a vrací nový stav. Reducer v zásadě řídí, jak se stav (data aplikace) změní jako reakce na akci. Jejich princip vychází z defaultní funkce *reduce* v JavaScriptu, kde se jedna hodnota vypočítá z více hodnot po provedení zpětného volání funkce [47].

2.9 Git

Git je nejrozšířenější systém správy verzí, který sleduje a ukládá změny v souborech. Tím poskytuje historii provedených úprav a umožňuje návrat k určitým verzím, pokud je to potřeba. Usnadňuje vývojářům spolupráci na jednom projektu aplikace díky schopnosti uceleně sloučit provedené změny od více lidí do jednoho zdroje. Git je software, který běží lokálně, kde také ukládá veškeré soubory a jejich historii. Nicméně za pomocí online serverů jako je například GitHub nebo GitLab, je také možné udržovat správu složek a jejich historii online, dostupnou odkudkoliv. Schopnost Gitu automaticky sloučit změny z různých zdrojů zabraňuje ztrátě práce, dokonce i při práci dvou lidí na různých částech stejného souboru [48].

3 NÁVRH APLIKACE

Tato kapitola detailně pojednává o návrhu webové aplikace. Rozebírá otázku ohledně zpracovávaných dat, popisuje návrh jak pro serverovou, tak i klientskou část a databázi. V serverové sekci jsou rozepsány jednotlivé komponenty a jejich vzájemná interakce, včetně designu API a plánu zabezpečení. Návrh databáze se zaměřuje především na strukturu dat a poskytuje jejich praktický příklad. Nakonec, klientská část podrobně popisuje návrh uživatelského rozhraní aplikace.

Celkový návrh by se měl také opírat o zvolenou technologickou sadu pro vývoj webové aplikace, čímž je MERN stack [49]. Ten byl zvolen z důvodu použití jednoho programovacího jazyka JavaSript a jeho nadstavby TypeSript jak pro klientskou, tak i serverovou část aplikace. Název MERN stack stojí za technologiemi MongoDB, Express.js, React a Node.js.

3.1 Zpracovávaná data

Data, na kterých je aplikace závislá se dělí do dvou skupin, cizí a vlastní. Cizí data představují množinu informací týkající se předmětů určité hry a jejich závislostí, kterými jsou například cena, kategorie nebo kolekce. Tato data by měla být primárně poskytována přímo jejich zdrojovou platformou Steam z důvodu aktuálnosti a pravdivosti. Steam však tyto data poskytuje pouze svým ověřeným vývojářům, kteří vlastní partnerský API klíč. Proto se aplikace obrací na služby třetích stran, z nichž některé jsou zpoplatněné a některé volně k použití, za účelem získání potřebných dat v dostatečné kvalitě, aby odpovídali stanoveným kritériím aplikace. Vlastní data pak odpovídají transformací cizích dat a uživatelských kolekcí.

3.2 Návrh pro backend

Očekává se, že serverová část aplikace bude hlavním pilířem celého systému. Je proto klíčové, aby zajišťovala neustálý provoz a propojení uživatele s databází prostřednictvím API. Kromě toho by měla také poskytovat základní úroveň ověřování uživatele a umožňovat přístup k datům na základě ověření.

3.2.1 Struktura serverové části

Struktura serverové části se dělí do čtyř základních modulů, kterými jsou server, routa, model a ovladač. Každý z těchto modulů představuje určitou úlohu, která přispívá k celkovému řešení serverového návrhu.

Serverový modul se stará o inicializaci a konfiguraci serverové části aplikace. Vytváří Express aplikaci, nastavuje middleware pro zpracování požadavků a odpovědí, řídí CORS (Cross-

Origin Resource Sharing) pro řízení přístupu ke zdrojům a konfiguruje prostředí pomocí modulu dotenv. *Modul routy* definuje chování aplikace pro určité HTTP požadavky na konkrétní URL cesty. Pomocí rout může server strukturovat aplikaci a směrovat různé cesty k různým částem kódu (kontrolérům). *Modul modelu* definuje strukturu dat určité kolekce, se kterou pracuje knihovna pro práci s databází. *Kontrolérový modul* pak zajišťuje veškeré operace nad daty. Přijímá požadavky od uživatele skrze specifické cesty (routy), provádí výpočty, manipuluje s databází (vytváří, upravuje, maže, získává nebo ukládá konkrétní informace) a získává potřebná data z externích aplikací, aby nakonec vrátil uživateli požadované odpovědi.

3.2.2 API design

Pro propojení klientské a serverové části je v aplikaci použité API zpracováno architekturou REST. Vytvořené API má za úkol poskytnou uživateli přístup pro zpracování veškerých potřebných požadavků, od získání všech předmětů, jejich cen nebo kolekcí, až po autorizaci uživatele skrze Steam OpenID Authentication [50]. Níže v tabulce jsou popsány všechny routy, které by měla serverová část zajistit pro své uživatele.

Metoda	Routa	Popis
GET	/items	Vrací předměty
PUT	/items	Synchronizuje předměty
GET	/types	Vrací typy předmětů
GET	/prices	Vrací ceny předmětů
GET	/collections	Vrací kolekce přihlášeného uživatele
POST	/collections	Vytváří kolekci pro přihlášeného uživatele
PUT	/collections/:id	Edituje kolekci přihlášeného uživatele
DELETE	/collections/:id	Maže kolekci přihlášeného uživatele
GET	/user	Vrací data přihlášeného uživatele
GET	/user/logout	Odhlašuje uživatele
GET	/user/login	Přihlašuje uživatele
GET	/user/inventory	Vrací inventář uživatele

Tabulka 1 - API routy aplikace. Zdroj vlastní.

3.2.3 Uživatelské ověření

Aplikace poskytuje uživatelům přihlášení do systému a tím jím zpřístupňuje rozšířené funkcionality. Ověřování je zpracováno za pomocí autentizační knihovny Passport [51], která umožňuje jednoduchou autentizaci uživatele v aplikaci pomocí různých strategií. Passport strategie pro Steam využívá OpenID protokol, který uživatelům umožňuje se autentizovat pomocí jejich účtu na Steamu. Po konfiguraci Passportu pro použití Steam strategie, je uživatel přesměrován na stránku přihlášení na Steamu, kde se přihlásí. Poté je přesměrován zpět na aplikaci s unikátním identifikátorem (Steam ID) a dalšími informacemi o svém účtu. Passport poté ověří tyto údaje u Steamu a aplikaci předá informace o uživateli. To umožňuje aplikaci pracovat s těmito údaji, jakož i přihlašovat uživatele přes jejich účet na platformě Steam.

Ověření uživatelé mají poté v aplikaci přístup ke svému profilu, kde mohou sledovat vývoj celkové ceny vlastněných předmětů v průběhu času u vybrané hry nebo vytvářet vlastní kolekce předmětů, pro lepší přehled určitých skupin.

3.3 Návrh databáze

Webová aplikace používá jako hlavní uložiště svých dat NoSQL databázi MongoDB. Ta byla vybrána z důvodu flexibility datového modelu a rychlosti dotazů a operací při velkém množství dat. Pro uchování informací o předmětech a jejich závislostech, používá MongoDB kolekce s následující modely, které je zastupují.

- itemDescriptionModel – představuje hlavní strukturu jednoho předmětu
- itemPriceModel – obsahuje data vhodná pro grafy, které zpřehledňují vývoj ceny předmětu v čase
- itemTypeModel – typy předmětů rozdělené do skupin, které se využívají pro filtrování v přehlídce předmětů
- itemInfoModel – ukládá informaci o poslední synchronizaci předmětů
- userModel – zaznamenává přihlášené uživatele do aplikace

Aplikace na serverové části každou hodinu vynutí synchronizaci všech předmětů. Při té příležitosti je také zaznamenána aktuální cena předmětu, o kterou rozšíří pole v itemPriceModelu a aktualizuje průměrnou denní hodnotu. Data v tomto modelu jsou tvořena především sběrem samotné aplikace a jejich ztráta by mohla znehodnotit důležité funkcionality.

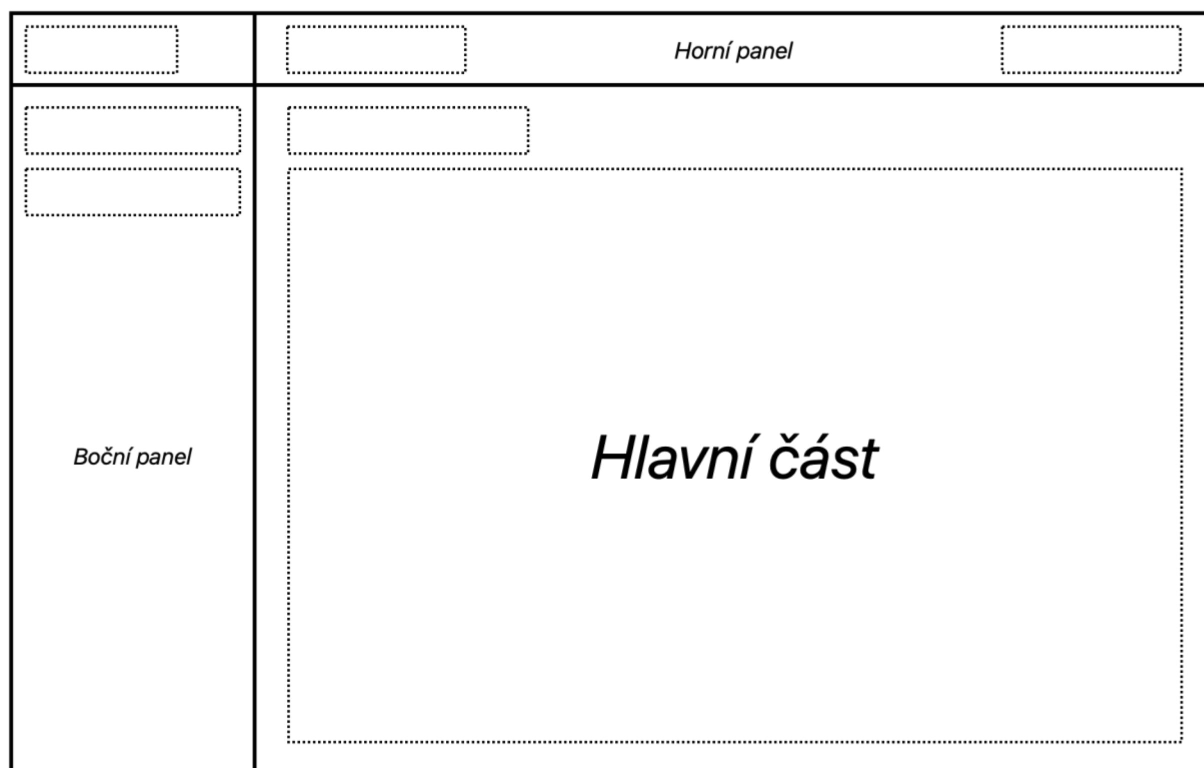
Proto se tyto data zálohují každou synchronizací předmětů do souborového adresáře serverové části kódu, aby bylo možné, v případě ztráty, tyto data zase obnovit.

3.4 Návrh pro frontend

Uživatelské rozhraní aplikace je postaveno na SinglePageApplication (SPA) architektuře v Reactu. To znamená, že její hlavní části jsou poskládány z jednotlivých komponent, které využívají stav aplikace za účelem dynamické aktualizace jednotlivých prvků. Stav aplikace je umístěn na globální úrovni za použití knihovny Redux, která zajišťuje skrze své funkce přístup z jakékoliv části kódu.

3.4.1 UI/UX Design

Grafický návrh se řídí principy a pravidly Material Designu [52], vyvinutého společností Google, který se zaměřuje na tvorbu vizuálně atraktivního rozhraní. Jednotlivé dílčí prvky (tlačítka, paragrafy seznamy nebo bloky) poskytované knihovou Material Design zaručují jednotný styl vzhledu a rozvržení, čímž usnadňují vývoj a zaručují konzistentnost napříč celou aplikací. Uživatelské rozhraní je rozvrženo do tří hlavních částí (boční panel, horní panel a hlavní část). Každá z těchto částí obsahuje své aktivní prvky, s jejichž interakcí může a nemusí ovlivňovat zbylé dvě části.



Obrázek 6 - Rozvržení uživatelského rozhraní aplikace. Zdroj vlastní.

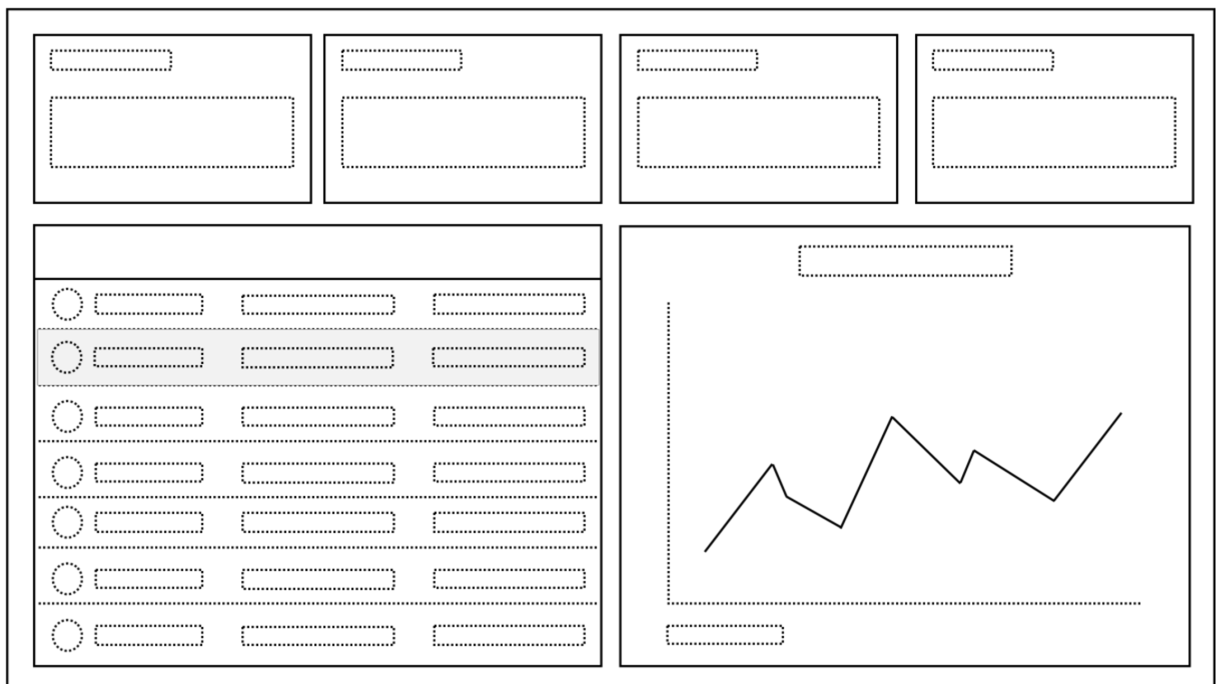
Boční panel je část uživatelského rozhraní umístěná na levém okraji stránky, která slouží uživatelům k navigaci skrze aplikaci. Obsahuje aktivní prvky tvořeny ikonkou a názvem sub-aplikace, které mění obsah hlavní části. Boční panel se zobrazuje ve dvou variantách (otevřená a zavřená) pro uvolnění zobrazovaného místa ostatním částem a zajištění očekávaného chování při změně velikosti okna aplikace.

Úlohou horního panelu, zobrazovaného ve vrchním segmentu okna aplikace, je především jednoznačná identifikace obsahu hlavní části, kterou zajišťuje pomocí paragrafu umístěného v levé oddíle panelu. Horní panel má celkem dva aktivní prvky. Tím druhým je tlačítko sloužící pro aktivaci funkcí spojených s ověřováním uživatele a pro zobrazování jeho jména a avatara.

Hlavní část je primární zobrazující prvek aplikace, který mění svůj obsah na základě interakce uživatele s bočním panelem. V jeho horní části se nachází drobečkové menu, jehož obsahem je lineární sekvence uzlů, která následuje směr průchodu uživatele ke koncové sub-aplikaci. Každý z uzlů je sám aktivním prvek umožňující uživateli vracet se ke konkrétnímu obsahu, který uživatel při svém průchodu navštívil. Obsahem hlavní části jsou poté sub-aplikace, které jsou navrženy tak, aby pokryli veškeré požadavky očekávané od vytvořené aplikace. Ve výsledku se jedná o Dashboard, Showcase, Profil a Detail předmětu.

3.4.2 Dashboard

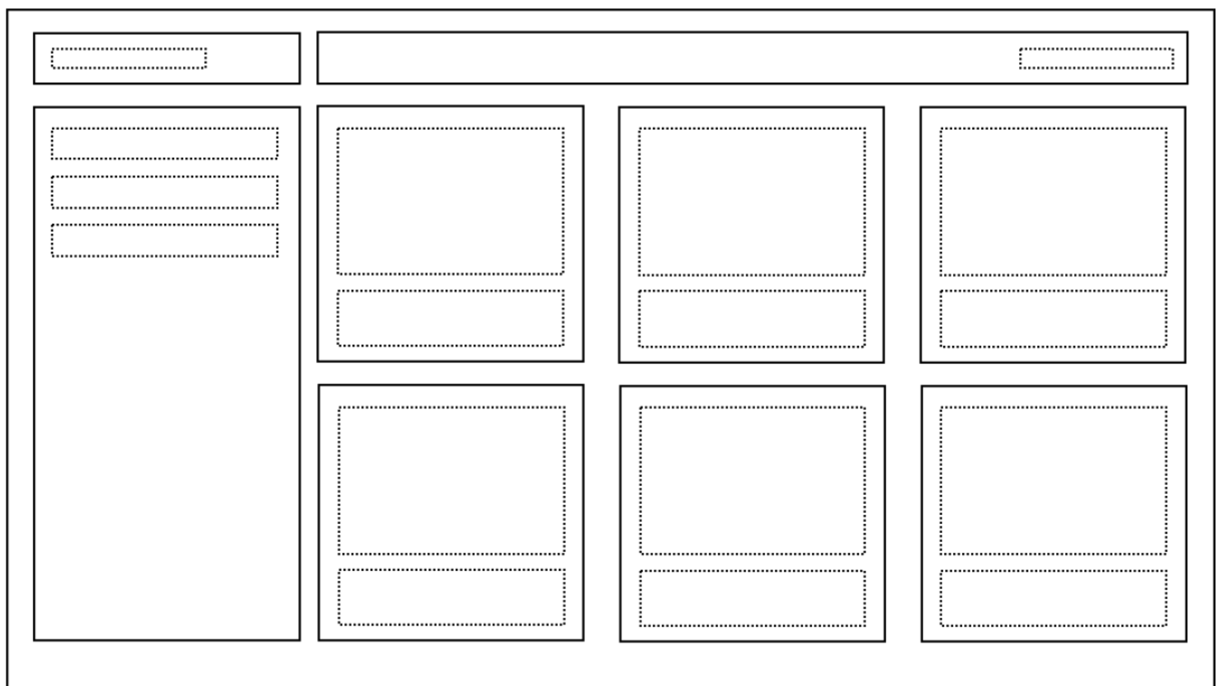
Dashboard, jakožto domovský obsah aplikace, nepotřebuje ke svému zobrazení žádné ověření. Poskytuje uživatelům prostý a jednoduše pochopitelný náhled do dat, z kterých lze jednoznačně vyčíst informace vhodné pro určení správné obchodní strategie. V první řadě zobrazuje čtyři karty obsahující zvolenou hru, počet kontrolovaných předmětů, globální pohyb průměrné celkové ceny trhu zvolené hry a datum poslední aktualizace předmětů. V dolní části poté poskytuje uživateli souhrnnou tabulku všech předmětů seřazených podle aktuálního trendu vývoje průměrné ceny a vedle ní v pravé části graf, který je s ní úzce spojený a zobrazující vývoj průměrné ceny zvoleného předmětu.



Obrázek 7 - Návrh zobrazení sub-aplikace Dashboar. Zdroj vlastní.

3.4.3 Showcase

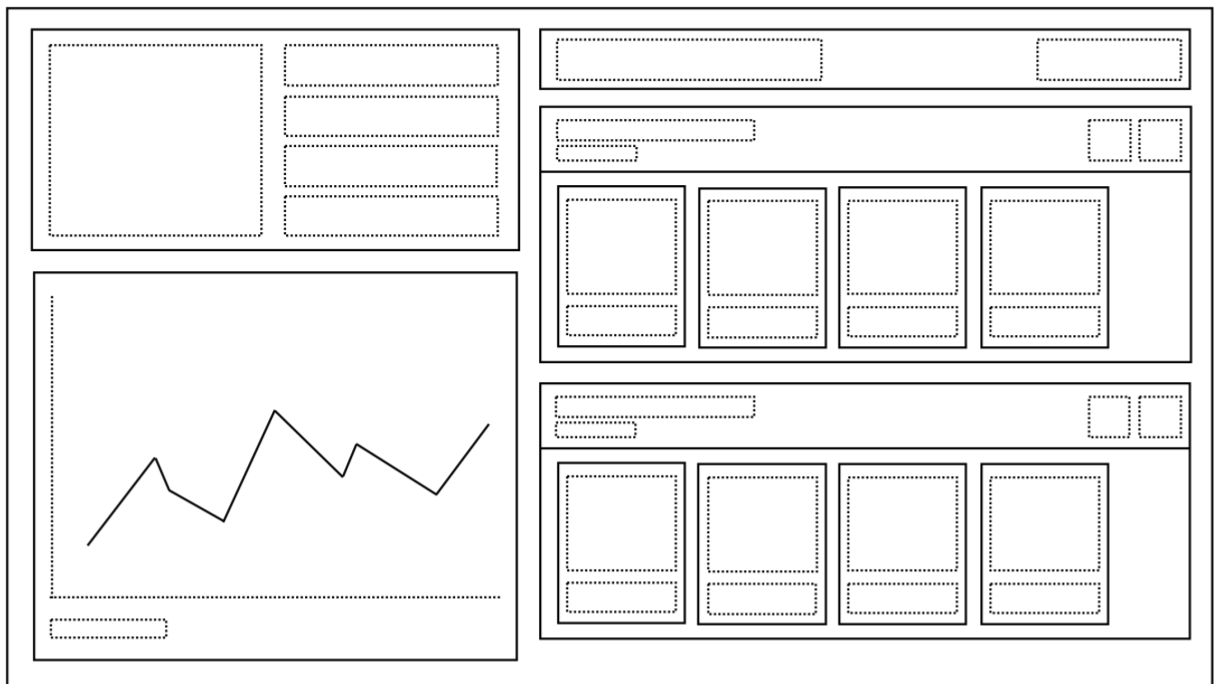
Sub-aplikace Showcase, která je dostupná bez potřeby ověření, má za úkol uživateli zobrazit skupinu předmětů na základě jejich zvolených kritérií (filtrů). Dělí se na dvě části. Tou první je blok poskytující filtrovací funkce a je umístěn po levé straně. Uživatel má možnost filtrovat předměty na základě jejich názvu a typu. Zvolené podmínky jsou poté spojeny pomocí logického operátoru AND a ohraničují sadu předmětů, která se uživateli zobrazí. Druhou částí sub-aplikace Showcase je gridová soustava předmětů, která z důvodu optimalizace zobrazuje vždy pouze 12 z nich najednou a je řízena stránkovací komponentou v jejím horním segmentu.



Obrázek 8 - Návrh zobrazení sub-aplikace Showcase. Zdroj vlastní.

3.4.4 Profil

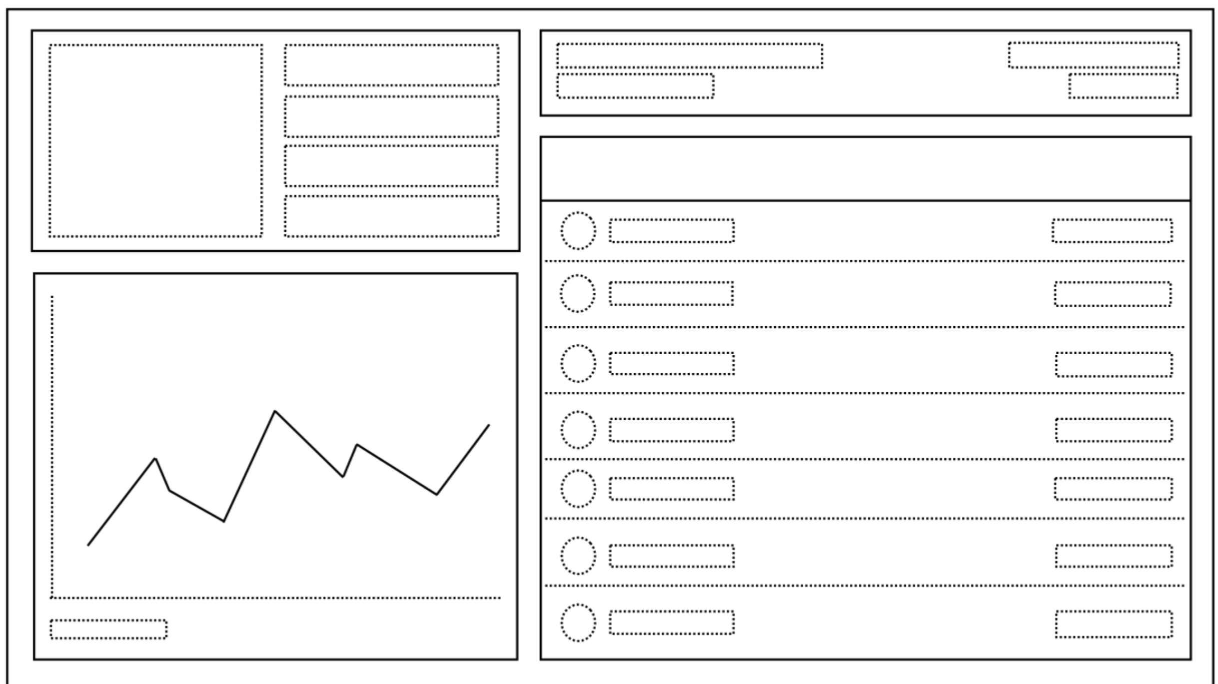
Profil je část aplikace, která nabízí uživateli svůj osobní prostor. Zobrazuje veřejně dostupné informace uživatele, které systém získal po jeho ověření a poskytuje mu možnost vytvářet, editovat a mazat kolekce předmětů. Levá část profilu začíná od shora kartou uživatele. Na této kartě může kromě osobních informací, získat i přehled o stavu svého profilu. Proto, aby navržený systém mohl sledovat údaje týkající se obsahu inventáře, musí mít uživatel nastavený stav profilu na platformě Steam na veřejný. Pokud tak udělá, poskytne mu aplikace sledovat celkovou hodnotu svého inventáře zvolené hry v průběhu času za pomocí grafu umístěného pod kartou uživatele. V pravé části profilu je pak prostor pro práci s osobními kolekcemi. Zde je uživateli umožněno vytvářet kolekce za pomoci sub-aplikace Showcase zobrazené v dialogovém okně, ve které předměty do kolekce vybírá. Následně je může editovat a mazat v samotném bloku vytvořené kolekce.



Obrázek 9 - Návrh zobrazení sub-aplikace Profil. Zdroj vlastní.

3.4.5 Detail předmětu

Detail předmětu je sub-aplikace, která je zaměřena na poskytnutí přidaných a užitečných informací o zvoleném předmětu. Levá část je podobná té, která je navržená pro profil hráče. Začíná kartou předmětu, na které je vidět zvětšený pohled předmětu a informace jako jsou datum kdy byl předmět přidán do hry, aktuální cena a originální cena předmětu. Dále je levá část zakončena grafem vývoje průměrné denní ceny předmětu stejně jako je tomu v sub-aplikaci Dashboard. Pravá část detailu předmětu nabízí uživateli přehled o předmětech, které patří do stejné kolekce. Tím je na mysli kolekce, kterou vytváří přímo autor předmětu a podle které jsou i předměty pojmenovány, například „Blackout Hoodie“.



Obrázek 10 - Návrh zobrazení sub-aplikace Detail předmětu. Zdroj vlastní.

3.4.6 Interakce s backendem

Interakce se serverovou částí probíhá prostřednictvím asynchronního volání příslušných funkcí, které jsou součástí vrstvy aplikační logiky. Tyto funkce využívají rozhraní „fetch" k odeslání požadavků na server s určitou cílovou adresou a požadovanými daty v těle požadavku. Při úspěšné odpovědi ze serveru jsou zpracována získaná data, která jsou následně odeslána pomocí akcí do stavového uložiště klientské části aplikace za účelem inicializace nebo aktualizace zobrazovaných dat. V případě neúspěšné odpovědi serveru je vyvolána chyba s detailním popisem neúspěšného požadavku. Komunikace mezi frontendem a backendem probíhá v rámci aplikační vrstvy, využívá standartních nástrojů pro zasílání a zpracování HTTP požadavků a jako formát dat používá JSON, což umožňuje snadnou serializaci a deserializaci dat mezi nimi.

4 IMPLEMENTACE APLIKACE

Kapitola obsahuje stručné shrnutí implementace dílčích prvků obou částí aplikace (frontend a backend), které tvoří jejich nedílnou součást. Dále kapitola obsahuje strukturu složek vytvořeného projektu a části zdrojových kódů pro lepší pochopení zpracovaných funkcionalit.

4.1 Inicializace projektů

Jak už z návrhu vyplívá, aplikace je rozdělena do dvou projektů, které se nazývají backend a frontend. Pro počáteční inicializace obou projektů je použit balíčkovací systém NPM (Node Package Manager) [53], který zprostředkovává oběma projektům správu závislostí (instalaci externích balíčků, knihoven a jejich verzování) a spouštění vlastních skriptů. Projekt backendu využívá prostředí Node.js na jehož začátku stojí příkaz *npm init*. Ten provede inicializaci projektu a vytvoří soubor „package.json", ve kterém jsou uloženy metadata o projektu, včetně názvu, verze, popisu a závislostí na externí balíčky. Ještě před spuštěním projektu je potřeba nainstalovat všechny externí balíčky pomocí příkazu *npm install,* který vytvoří složku node-modules v níž se budou všechny stažené a instalované externí balíčky nacházet. Spuštění backendového projektu (serveru) se provádí příkazem *node*, který informuje Node.js interpret, aby spustil JavaScriptový soubor uvedený jako argument. V souvislosti s tímto projektem a podle struktury popsané níže je kompletní příkaz pro spuštění *node server.js.* Tímto příkazem se připravený server zpřístupní na adrese *http://localhost:4000*, kde port lze volně konfigurovat. Projekt frontendu (klientské části) je vytvořen pomocí knihovny React a pro jeho inicializaci se používá nástroj „create-react-app". Ten není nutné explicitně instalovat. Místo toho lze spustit přímo z registru NPM pomocí příkazu *npx.* Výsledný příkaz pro inicializaci frontendu *npx create-react-app frontend --template typescript* vytvoří nový projekt s názvem „frontend" obsahující šablonu pro React aplikaci, která je nakonfigurována pro použití TypeScriptu, jenž je zde použit jako hlavní programovací jazyk. Pro spuštění projektu je opět potřeba instalace externích balíčků stejným způsobem jako je tomu u backendu. Samotné spuštění se poté provádí příkazem *npm start,* který je definován v souboru „package.json". Tento příkaz zavolá node server, který aplikaci frontendu zpřístupní na adrese *http://localhost:3000*. Posledním krokem přípravy projektů je inicializace lokální databáze MongoDB. Její instalace a spuštění závisí na použitém operačním systému a kompletní dokumentace k tomu potřebná je k nalezení na oficiálních stránkách MongoDB [54].

4.2 Struktura projektů

Každý z projektů (backend a frontend) můžeme vnímat jako odlišnou část aplikace, která má svou vlastní strukturu složek a souborů. Projekt backendu obsahuje především zdrojové kódy pro tvorbu RESTového API a komunikaci s databází. Nachází se zde konfigurace serveru, funkce napojené na API endpointy pro dotazy klienta, modely pro databázovou strukturu a nastavení strategie pro autentizaci klienta.

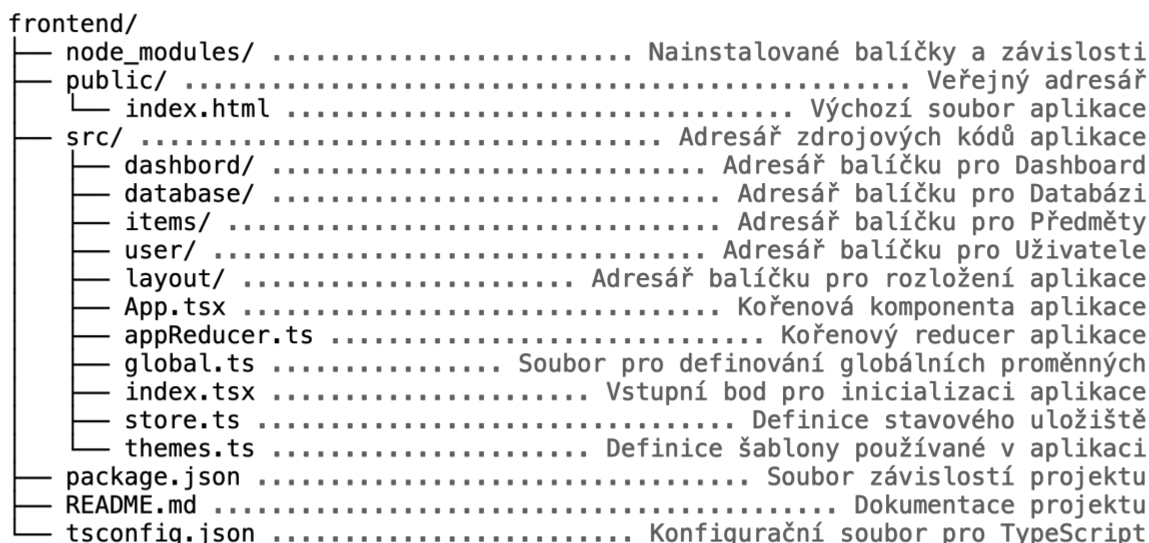
Obrázek 11 - Struktura backendu. Zdroj vlastní.

Struktura backendu působí ve snaze implementovat architekturu Model-View-Controller (MVC) [55]. Tato architektura rozděluje aplikaci do tří základních vrstev, což usnadňuje lepší organizaci a čitelnost kódu. Důraz je kladen na oddělení logiky aplikace (model) od její prezentační části (view), což umožňuje vytvořit strukturu, kde každá část má svou specifickou roli a není vše obsaženo v jediném souboru.

Nicméně, v tomto projektu neobsahuje struktura typický prvek „view", který bývá zodpovědný za prezentaci dat uživateli. Místo toho je zde využito routovacích mechanismů uložených v adresáři „routes“, které přebírají požadavky od uživatelů spolu s jejich daty a směrují je k příslušnému kontroléru (controller). Kontrolér následně manipuluje s modelem, přiřazuje mu

data, a nakonec vrací výsledky zpět routovacím mechanismům. Ty poté reagují na klienta s výslednou odpovědí. Za účelem postupného růstu projektu přidáním dalších her a zpracovávání jejich dat, jsou modely a kontroléry rozděleny do jednotlivých adresářů týkající se jedné z konkrétních her.

Projekt frontendové části obsahuje zdrojové kódy jednotlivých aplikací včetně stylů určující jejich vzhled, s nimiž uživatel interaguje. Standardní struktura projektu v Reactu se snaží vytvořit přehledné rozdělení mezi komponentami, styly, funkcemi, soubory konfigurace a dalšími zdroji. Základními adresáři, které nabízí po inicializaci jsou *node_modules*, *public* a *src*. V dalším průběhu implementace bude záviset hlavně na adresáři src, ve kterém jsou umístěny podadresáře vytvořených sub-aplikací a soubory starající se o stavové uložiště klientské části.



```
frontend/
├── node_modules/ ........................ Nainstalované balíčky a závislosti
├── public/ ............................................. Veřejný adresář
│   └── index.html .............................. Výchozí soubor aplikace
├── src/ ................................. Adresář zdrojových kódů aplikace
│   ├── dashbord/ .............................. Adresář balíčku pro Dashboard
│   ├── database/ ............................. Adresář balíčku pro Databázi
│   ├── items/ ................................ Adresář balíčku pro Předměty
│   ├── user/ ................................. Adresář balíčku pro Uživatele
│   ├── layout/ ..................... Adresář balíčku pro rozložení aplikace
│   ├── App.tsx ............................... Kořenová komponenta aplikace
│   ├── appReducer.ts ............................. Kořenový reducer aplikace
│   ├── global.ts ............... Soubor pro definování globálních proměnných
│   ├── index.tsx .................... Vstupní bod pro inicializaci aplikace
│   ├── store.ts .............................. Definice stavového uložiště
│   └── themes.ts .................... Definice šablony používané v aplikaci
├── package.json ............................... Soubor závislostí projektu
├── README.md .......................................... Dokumentace projektu
└── tsconfig.json ........................ Konfigurační soubor pro TypeScript
```

Obrázek 12 - Struktura frontendu. Zdroj vlastní.

Samotný React nemá konkrétní způsob tvorby struktury adresářů a souborů, nicméně existuje několik běžných přístupů, které jsou populární v jeho ekosystému. Jedním z nich, který také tento projekt využívá, je seskupování podobných souborů dohromady. Proto jsou soubory patřící pod jednu sub-aplikaci seskupeny pod jedním adresářem a v něm zas uspořádány do složek podle úkonů, které vykonávají (viz. příklad níže).

```
dashboard/
├── api/ .......................... Adresář funkcí pro komunikaci s backendem
├── functions/ ......................... Adresář funkcí pro interní manipulaci
├── components/ ........................ Adresář komponent balíčku Dashboard
│   └── Dashboard.tsx ............................. Hlavní komponenta balíčku
├── actions.ts ................................... Soubor funkcí pro Redux
├── constants.ts ......................................... Soubor konstant
├── reducer.ts ................................. Reducer balíčku Dashboard
├── selectors.ts ............................. Soubor selectorů pro Redux
└── types.ts .................................................. Soubor typů
```

Obrázek 13 - Struktura sub-aplikace Dashboard. Zdroj vlastní.

4.3 Implementace serverové části

4.3.1 Nastavení Express

Framework Express.js zde velmi usnadňuje vytvoření a správu serverové části, ve které pomáhá rychlým způsobem nastavit počáteční konfiguraci a různé druhy middleware, které si lze představit jako sadu filtrů, přes které požadavek od uživatele putuje. První z nich implementovaný umožňuje Cross-Origin Resource Sharing (CORS) a nastavuje hlavičky, které povolují přístup z různých zdrojů. Dále následuje middleware, který automaticky zpracovává data v JSON formátu a URL-encoded formátu v tělech příchozích požadavků. Nesmí chybět ani middleware pro správu relací, který nastavuje parametry relace, včetně tajného klíče, uchování nesestavených relací a dalších. Express také definuje routy aplikace pro předměty a uživatele. Tyto routy jsou směrovány na příslušné routovací mechanismy, které obsluhují určité druhy požadavků. Nakonec s pomocí funkce *app.listen(port)* spouští webový server, který naslouchá na zadaném portu.

4.3.2 Routy

Jak už bylo zmíněno, prvek „view" v architektuře Model-View-Controller, podle které se řídí backendová architektura, zde nahrazují dva routovací mechanismy, jež jsou zprostředkované za pomocí komponenty Routeru z frameworku Express.js. Tyto routery umožňují definovat různé cesty (routy), zpracované podle návrhu, a mapovat je na odpovídající funkce nebo části kódu, které mají být provedeny.

```
// router
const router = express.Router();

// items routes
router.get("/items", getRustItems);
router.put("/items", synchronizeRustItems);

// types and prices routes
router.get("/types", getRustItemTypes);
router.get("/prices", getRustItemPrices);

// collections routes
router.get("/collections", getUserCollections);
router.post("/collections", createUserCollection);
router.put("/collections/:id", updateUserCollection);
router.delete("/collections/:id", removeUserCollection);
```

Obrázek 14 - Implementace rout pro předměty. Zdroj vlastní.

4.3.3 Modely

Tvorba modelů hraje v rámci implementace důležitou roli, co se týče práce s daty. Veškerá data, procházející aplikací, mají návaznost na databázi MongoDB a z toho důvodu je zapotřebí použití knihovny, která komunikaci s ní zjednoduší. Pro tuto část aplikace, vzhledem k tomu, že běží na Node.js prostředí, byla vybrána knihovna Mongoose. S její pomocí je backendová část schopna v první řadě definovat schémata, které díky specifikaci polí, typů dat, validací nebo výchozích hodnot, poskytují jasný formát pro tvorbu následných modelů. Mongoose modely představují objektové reprezentace dat uložených v MongoDB na základě definovaných schémat. Ve výsledku představují konkrétní kolekce, skrze které lze vykonávat operace jako je čtení, zápis, aktualizace nebo mazání dat a tím zajistit datovou logiku aplikace.

```
const inventoryItemSchema = new mongoose.Schema({
    id: Number,
    name: String,
    count: Number,
  });

  const gameSchema = new mongoose.Schema({
    appId: String,
    inventory: [inventoryItemSchema],
  });

  const userInventorySchema = new mongoose.Schema({
    userId: String,
    games: [gameSchema],
  });

module.exports = mongoose.model("UserInventory", userInventorySchema);
```

Obrázek 15 - Příklad schéma a modelu inventáře uživatele. Zdroj vlastní.

```
  _id: ObjectId('65663c8d7645b03754189394')
  userId: "76561198074923638"
▾ games: Array (1)
  ▾ 0: Object
      appId: "rust"
    ▾ inventory: Array (16)
      ▾ 0: Object
          id: 4580329656
          name: "Comics SAR"
          count: 1
          _id: ObjectId('65663c8d7645b03754189396')
      ▾ 1: Object
          id: 4029168713
          name: "Blackout Jacket"
          count: 1
          _id: ObjectId('65663c8d7645b03754189397')
```

Obrázek 16 - Reprezentace modelu inventáře uživatele v MongoDB Compass. Zdroj vlastní.

4.3.4 Kontroléry

Posledním důležitým prvkem serverové části jsou kontroléry, které zajištují aplikační logiku. V zásadě se jedná o množinu asynchronních funkcí, které přijímají požadavky od klienta a vrací nazpět požadované odpovědi. Jejich činnost spočívá ve využívání připravených modelů, interních funkcí, komunikaci se službami třetích stran a vytváření odpovědi pro klienta. V projektu lze najít různě složité kontroléry. Mezi ty primitivní se řadí například kontrolér, který se stará o vrácení všech dostupných předmětů zvolené hry (viz. obrázek níže).

```
// get all rust items
const getRustItems = async (req, res) => {

  // find all items whose price is greater than 0
  const items = await RustItemDescription.find({
    buyNowPrice: { $gt: 0 },
  }).sort({ timeAccepted: -1 });

  res.status(200).json({ items: items });
};
```

Obrázek 17 - Příklad primitivního kontroléru vracejícího všechny předměty zvolené hry. Zdroj vlastní.

Kontrolér „getRustItems" nachází za pomoci určitého modelu všechny předměty ze hry Rust, jejichž aktuální cena je větší než 0. Tím tyto předměty filtruje od těch, které jsou sice nastavené jako obchodovatelné, ale jejich vlastnictví je bezcenné a tím pádem i pro celkovou aplikaci nezajímavé. Nalezené předměty následně třídí podle datumu vydání od nejnovějších a posílá je zpět uživateli se statusovým kódem 200 a ve formátu JSON.

Mezi složitější kontroléry, které jsou v projektu implementovány, patří například kontrolér synchronizující předměty. Ten využívá jak interních funkcích zprostředkované vlastní importovanou knihovnou „lib“, tak dotazů na veřejné API za účelem získání aktuálních dat.

```
// synchronize all items from rust
const synchronizeRustItems = async (req, res) => {
  try {
    // backup
    await lib.backupCollection(RustItemPrice);
    // Delete all items in the database
    await RustItemDescription.deleteMany();
    // Get rust items from the public api
    const items = await api.getRustItems("EUR");
    // Create new items in the database
    await RustItemDescription.create(items);
    // Get all item prices from the database
    const dbItemPrices = await RustItemPrice.find({});
    // Get all item prices from the public api
    const apiItemPrices = await api.getRustItemsPrices();
    // Delete all prices in the database
    await RustItemPrice.deleteMany();
    // Merge items prices
    const itemsPrices = lib.mergeRustItemsPrices(apiItemPrices, dbItemPrices);
    // Create new itemsPrices in the database
    await RustItemPrice.create(itemsPrices);

    // Update items synchronize date
    await RustItemInfo.deleteMany();
    await RustItemInfo.create({ lastItemsUpdate: Date.now() });

    res.status(200).json(items);
  } catch (error) {

    res.status(500).json({ error: 'Internal server error' });
  }
};
```

Obrázek 18 - Příklad kontroléru synchronizující všechny předměty ze hry Rust. Zdroj vlastní.

Kontrolér „synchronizeRustItems“ je navržený tak, že krom synchronizace všech předmětů zvolené hry aktualizuje také jejich cenový model. Ten zastupuje taková data, která aplikace využívá k zobrazení časových průběhů průměrné denní hodnoty předmětu a jsou sbírána právě navrženým kontrolérem. Proto prvním krokem kontroléru je záloha cenového modelu předmětů do adresáře „backup“ ve formátu JSON souborů, aby se v případě jejich ztráty mohly znovu nahrát do systému. Dalšími funkcemi jdoucí za sebou jsou smazání všech dostupných předmětů, stažení předmětů za pomoci API třetích stran a vytvoření nových předmětů do databáze. Tímto způsobem se provede jejich rychlá aktualizace. Dále se provede rozšíření cenového modelu každého z předmětů o aktuální hodnotu a vzápětí se přepočítá i průměrná hodnota předmětu vázaná na den spuštění synchronizace. Posledním krokem kontroléru je aktualizace časového razítka poslední synchronizace předmětů.

Herní společnost Facepunch, která stojí za hrou Rust, vydává novou kolekci svých předmětů vždy jednou týdně. Dalo by se tedy říct, že tato synchronizace by se mohla spustit pouze jeden konkrétní den v týdnu a tím ušetřit náklady se synchronizací spojené. Nicméně navržený systém není schopen rozpoznat změnu na službách třetích stran (myšleno přidání nových předmětů), proto je synchronizace defaultně spuštěna každou hodinu a je zpracována tak, aby během jednoho průběhu přidala nové předměty, pokud existují, a zároveň aktualizovala jejich cenu.

4.4 Implementace klientské části

V této kapitole bude rozebrána implementace na straně klienta s důrazem na konkrétní detaily stavového uložiště nebo příklady komponent jednotlivých sub-aplikací zpracované za pomoci frameworku React.js. Sub-aplikace vyvíjené pomocí tohoto frameworku implementují znovupoužitelné komponenty, které využívají funkcionálního přístupu. To znamená, že jednotlivé komponenty mají podobu klasických funkcí, které přijímají parametry, řeší vnitřní stav a vrací HTML, jenž udává výsledný vzhled komponenty.

4.4.1 Nastavení stavového uložiště Redux

Proces nastavení Reduxu začíná instalací potřebných knihoven a závislostí, která se provede příkazem *npm install redux react-redux redux-thunk reselect*. Samotná knihovna *redux* poskytuje jádro pro správu stavu aplikace. Její hlavní prvky zahrnují store, akce a reducery. Následující knihovny jsou jen užitečné doplňky, které rozšiřují nebo usnadňují práci s Reduxem. Knihovna *react-redux* poskytuje propojení mezi Reactem a Reduxem. Pro potřeby aplikace poskytuje důležitou komponentu *<Provider />*, která umožňuje propojit redux store s vytvořenými React komponentami a její zasazení je přímo u samotného kořene jejich stromové hierarchie. Knihovna *redux-thunk* poskytuje Reduxu middleware, který umožňuje psaní asynchronních akcí, jež jsou důležité například pro operace volající API dotazy. Poslední knihovnou je *reselect.* Ta poskytuje efektivní a paměťově úsporné selektory, které ukládají své výsledky do mezipaměti a znovu je používají, pokud se vstupní data nezmění. Tyto knihovny jsou často používané společně za účelem optimalizace výkonu samotného Reduxu a pro zajištění tvorby robustní a efektivní správy stavu aplikace na straně klienta.

Dalším krokem nastavení je konfigurace redux storu. Ten se vytváří pomocí funkce *createStore*, která přijímá jako své parametry počáteční stav, kořenový reducer a objekt rozšiřujících funkcí, například přidání middleware z knihovny *redux-thunk*.

```
const composeEnhancers = process.env.NODE_ENV !== 'production' ?
  ((window as any).__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose) : compose;

export const configureStore = (initialState: object = {}) => {

  return createStore(
    createAppReducer(),
    initialState,
    composeEnhancers(
      applyMiddleware(
        thunk
      ),
    ),
  );
};
```

Obrázek 19 - Příklad konfugurace Redux store. Zdroj vlastní.

Jak je vidět na obrázku výše, kořenový reducer je zastoupen funkcí *createAppReducer*, která seskupuje všechny vytvořené reducery od různých sub-aplikací a poskytuje k nim přístup na stejné úrovni (tzn. hierarchicky jsou všichni umístěni hned za kořenem, kterým je globální stav). Výsledný redux store je následně používán jako parametr komponenty *<Provider />*.

4.4.2 Routing a rozložení

Středem routování na frontendové části je komponenta *<Router />*, která rozhoduje o tom, jaká sub-aplikace bude zobrazována v hlavní části rozložení. Ke své činnosti využívá komponentu *<Routes />* importovanou z knihovny *react-router-dom* jako nadřazený prvek pro všechny programově definované routy, jež obaluje. Každá z rout je pak tvořena z cesty a komponenty zastupující cílovou sub-aplikaci.

Rozložení aplikace začíná u souboru *App.tsx*. Z pohledu frontendové části se jedná o kořenovou komponentu, kterou začíná celá hierarchie, a proto implicitně implementuje komponentu *<Layout />* starající se o zobrazení jako je popsané v návrhu. Ta je navržena jako rodičovská komponenta pro potomky zobrazující se v hlavní části, které mu poskytuje *<Router />*.

Webová aplikace ve frontendové části využívá dílčí komponenty poskytované knihovnou *@mui* za účelem usnadnění tvorby vlastních datově napojených komponent. Jednou z důležitých komponent týkající se rozložení aplikace je komponenta *<Grid />*, která se používá ve dvou verzích (item a container) a pomáhá udržovat vzhled aplikace v responsivním režimu díky rozdělení své velikosti do dvanáctkové soustavy. Všechny komponenty poskytované knihovnou *@mui* lze stylově ovládat na základě změny rozlišení koncového zařízení. MUI [56] nabízí defaultně zlomové body změny, podle kterých se v jednotlivých komponentách rozhoduje.

- Extra small (xs) – 0px
- Small (sm) – 600px
- Medium (md) – 900px
- Large (lg) – 1200px
- Extra large (xl) – 1536px

Následné použití zlomových bodů v komponentě *<Grid xs={12} md={6} lg={3} />* pak určuje, kolik dílku z celkových dvanácti má zabrat komponenta jejího potomka v určitém rozlišení. Gridové komponenty napsané za sebou se vždy ve výsledném zobrazení skládají vedle sebe tak, aby zaplnili místo všech dvanácti dílků, pokud je to možné.

4.4.3 Klientská databáze

Klientská databáze zde představuje sub-aplikaci bez viditelné prezentace, která poskytuje datový základ stavů pro všechny prvky aplikace. V zásadě se jedná o instanci reduxu, do které se nahrávají informace pro určitou část aplikace v okamžik, když jsou potřeba. V dalším průběhu se již nadále aplikace nemusí dotazovat serverové části na data, ale využije právě ty, která jsou uchována v její klientské části. Je nutné zmínit, že veškerá viditelná data v aplikaci tvářící se jako statická jsou načítána právě z reduxové databáze. Data jako jsou například texty ve formulářích nebo odkazy v podobě identifikátorů na jiné objekty (dynamická data se kterými uživatel zrovna interaguje), jsou uloženy v separátních uložištích každé sub-aplikace, které jim poskytují editovací prostředí. Tímto způsobem se lze v případě nechtěné změny uživatele velice snadno vrátit k původnímu stavu dat přepisem z klientské databáze, která udržuje konzistentní stav.

4.4.4 Příklady implementovaných komponent

Jak už bylo zmíněno, sub-aplikace jsou poskládány z mnoha komponent. Ty jednoduší z nich fungují jen jako prostor pro data k zobrazení, což znamená, že nijak nemění stav aplikace ani neimplementují další vytvořené komponenty. Dobrým příkladem takového jednoduší komponenty je třeba karta v sub-aplikaci Dashbord vytvořená za pomocí komponenty *<InfoCard />*.

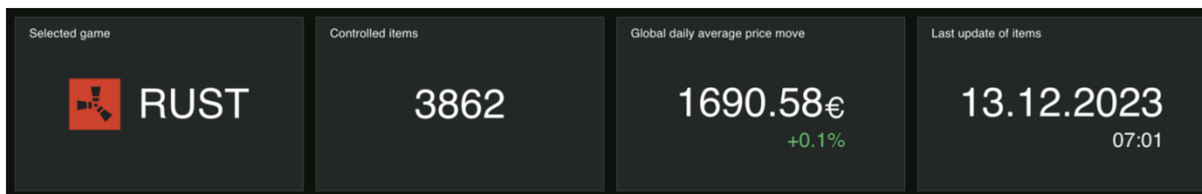
```
const InfoCard = ({title, content} : {title: string, content: ReactNode}) => {
  const theme = useTheme();
  return (
    <Box sx={{
      position: "relative",
      width: "100%",
      height: "23.78vh",
      backgroundColor: theme.palette.background.default,
      color: "white",
      display: "flex",
      alignItems: "center",
      justifyContent: "center",
      borderRadius: "1px",
      border: `1px solid ${theme.palette.divider}`
    }}>
      <Box sx={{ position: "absolute", zIndex: "2", top:"0.5em", left: "1em" }}>
        <Typography variant='body2'>
          {title}
        </Typography>
      </Box>
        {content}
    </Box >
  )
}

export default InfoCard
```

Obrázek 20 - Zdrojový kód komponenty <InfoCard />. Zdroj vlastní.

Tato komponenta přijímá za své parametry název karty a kontent, který se v ní má vykreslit. Jak je vidět ve zdrojovém kódu této komponenty, tak kontent je typu *ReactNode*, což ve své podstatě znamená, že komponenta přijímá jako kontent jakoukoliv další komponentu. Zbytek kódu se dále týká nastavení stylů různých prvků komponenty za pomoci CSS, aby se výsledné zobrazení drželo nastaveného designu. Výsledná implementace a zobrazení karet pak může vypadat například tak, jako je vyobrazeno na obrázcích níže.

```
<InfoCard title={"Selected game"}
  content={
    <Box
      sx={{
        display: "flex",
        alignItems: "center",
        justifyContent: "center",
        gap: 3,
        maxHeight: 80
      }}>
      <img src={GAMES_ICONS[GAME]} alt={GAME}
        style={{
          height: 60
        }}>
      </img>
      <Typography variant='h3' textTransform={"uppercase"}>
        {GAME}
      </Typography>
    </Box>
  } />
```

Obrázek 21 – Implementace komponenty <InfoCard /> pro vybranou hru. Zdroj vlastní.



Obrázek 22 - Zobrazení komponenty <InfoCard /> v různých stylech použitích. Zdroj vlastní.

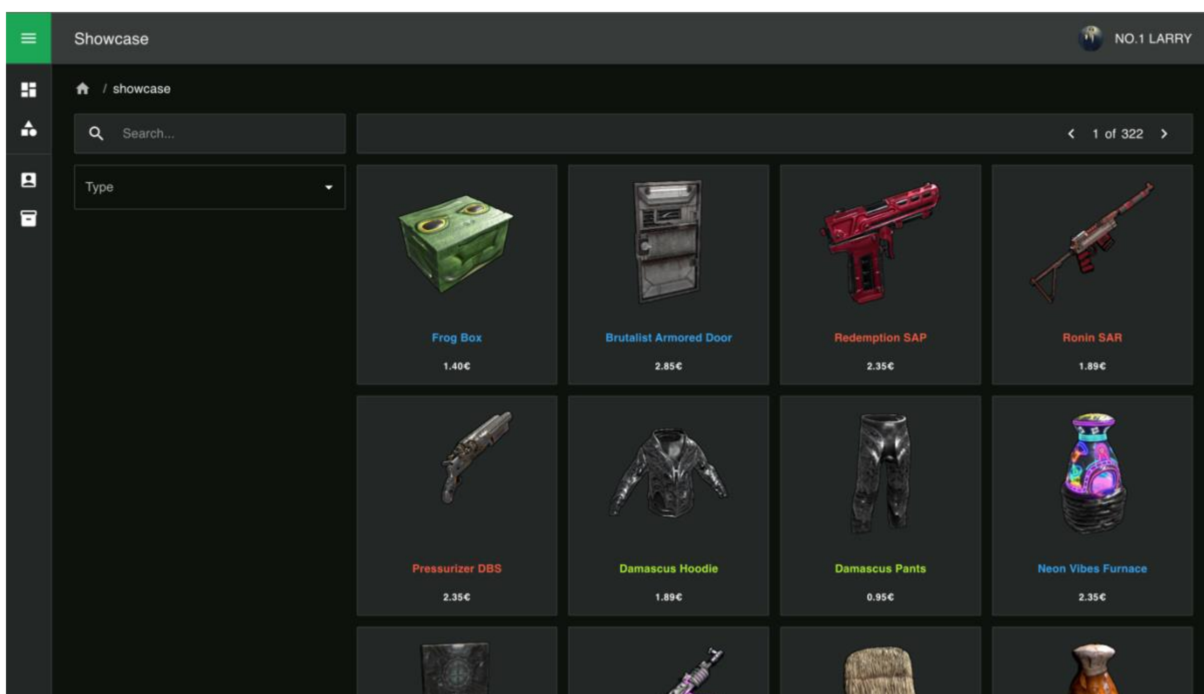
Vytvořené složitější komponenty dobře popisují zamýšlenou logiku při vývoji aplikace. Příkladem takové komponenty je komponenta *<Items />*, která je využívána pro gridové zobrazování předmětů s implementovaným filtrováním komponentou *<Filterbox />* a stránkováním komponentou *<Pager />*.

```
const Items = (
    { items, size = "medium", onItemClick, collection = false, addItem }:
        { items: IItem[], size?: string, onItemClick?: (id) => void, collection?: boolean, addItem?: boolean }) => {

    const SHOWCASE = useSelector((state: GlobalState) => state.dashboard.showcase);
    const ITEMS = useSelector(getFilteredItems(items));

    return (
        <Grid container spacing={2}>
            <Grid item xs={12} sm={12} md={4} lg={3} xl={2}>
                <Grid container spacing={2}>
                    <Grid item xs={12} sm={12} md={12} lg={12} xl={12}>
                        <Filterbox collection={collection} />
                    </Grid>
                </Grid>
            </Grid>
            <Grid item xs={12} sm={12} md={8} lg={9} xl={10}>
                <Grid container spacing={2}>
                    <Grid item xs={12} sm={12} md={12} lg={12} xl={12}>
                        <Pager items={items} />
                    </Grid>
                    <Grid item xs={12} sm={12} md={12} lg={12} xl={12}>
                        <Grid container spacing={2}>
                            {ITEMS && ITEMS.length > 0 && ITEMS
                                .slice(SHOWCASE.pagination.start, SHOWCASE.pagination.end)
                                .map((item, index) =>
                                    <Item
                                        addItem={addItem}
                                        size={size}
                                        item={item}
                                        key={index}
                                        procentualPriceDiffrence={null}
                                        onClick={onItemClick ? (id) => onItemClick(id) : null}
                                    />)}
                        </Grid>
                    </Grid>
                </Grid>
            </Grid>
        </Grid>
    )
}
export default Items
```

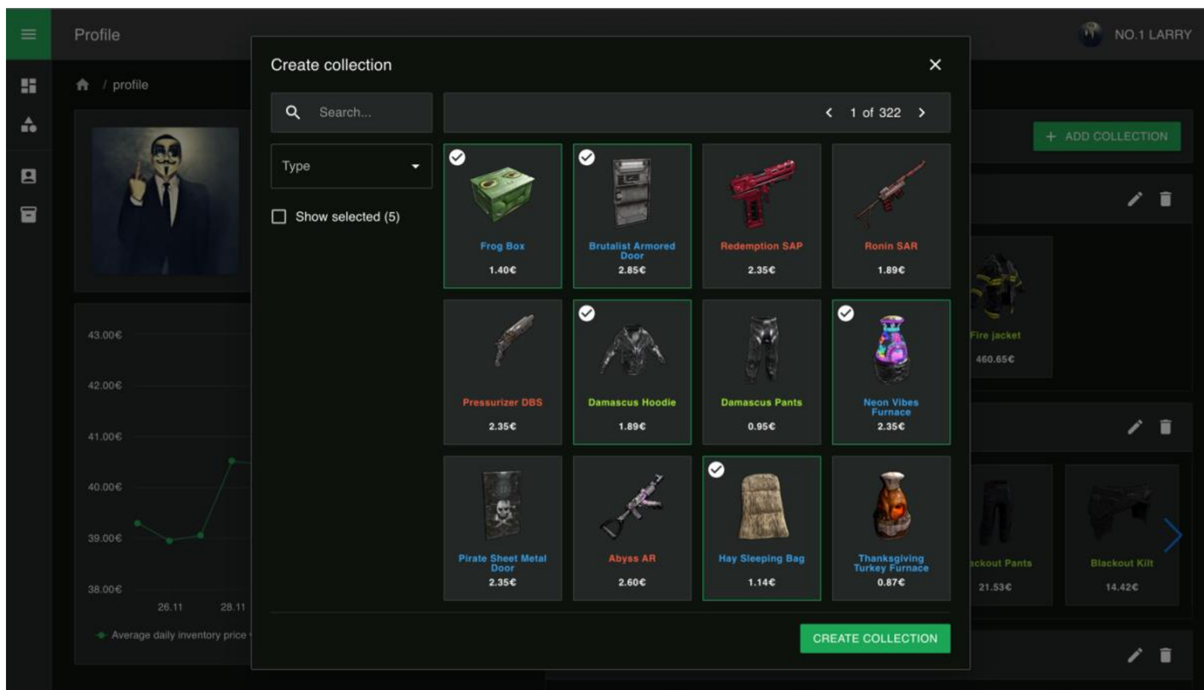
Obrázek 23 – Zdrojový kód komponenty <Items />. Zdroj vlastní.

Jako své parametry přijímá sadu předmětů, celkovou velikost, funkci spouštějící se po kliknutí na určitý předmět a dva přepínače ovlivňující vnitřní komponenty pro použití více způsoby. Komponenta pracuje také s dvěma stavy. První z nich „SHOWCASE“ je zde využíván k výběru

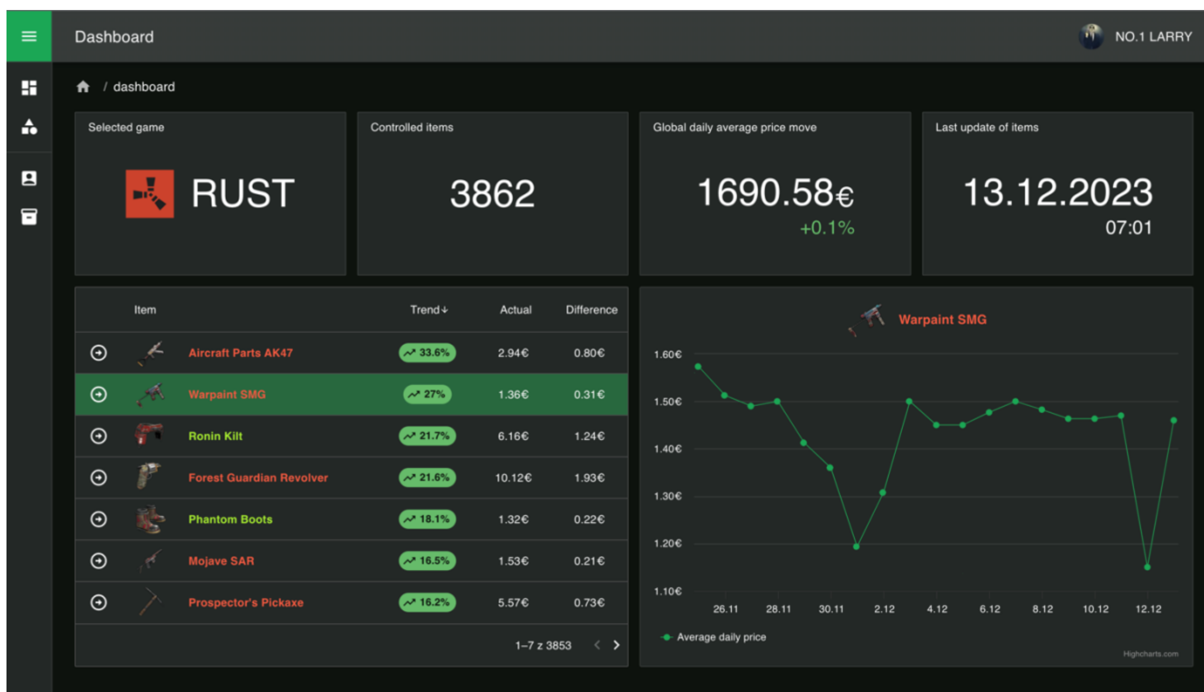
intervalu z filtrované sady poskytnutých předmětů, který se dále iteruje do gridové sestavy. Druhý „ITEMS" je zpřístupněn vytvořeným selektorem, který vrací sadu předmětů vyhovující podmínkám nastavených uživatelem v komponentě *<Filterbox />*. Jakákoliv změna těchto dvou stavů pak přinutí komponentu *<Items />* znovu vykreslovat určitou vnořenou komponentu a její potomky využívající tohoto stavu za účelem poskytnutí dynamického a konzistentního uživatelského rozhraní. Výstupem této komponenty je zobrazení používané například v sub-aplikaci Showcase, jako přehlídka všech předmětů nebo uživatelského inventáře a vypomáhá i jako výběrový katalog při tvoření nebo editaci uživatelských kolekcí.



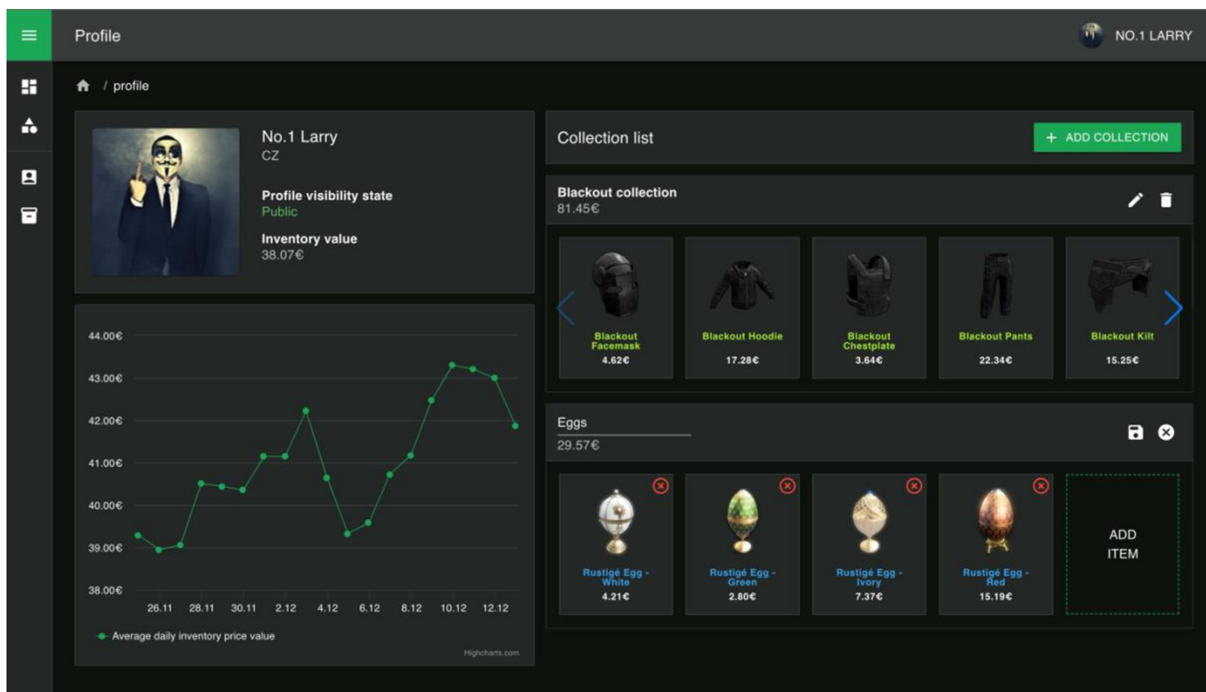
Obrázek 24 - Použití komponenty <Items /> jako přehlídka všech předmětů. Zdroj vlastní.



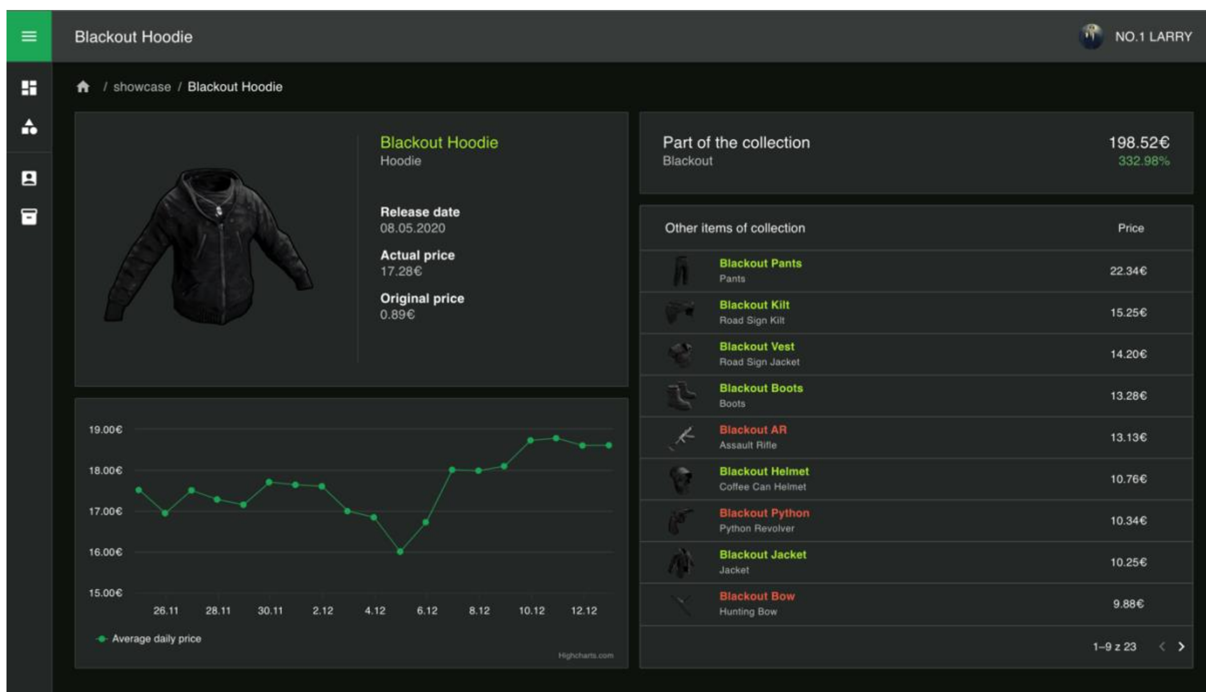
Obrázek 25 - Použití komponenty <Items /> jako katalog pro výběr předmětů kolekce. Zdroj vlastní.



Obrázek 26 - Zobrazení sub-aplikace Dashboard. Zdroj vlastní.



Obrázek 27 - Zobrazení sub-aplikace Profil. Zdroj vlastní.



Obrázek 28 - Zobrazení sub-aplikace Detail předmětu. Zdroj vlastní.

5 TESTOVÁNÍ A BUDOUCNOST NASAZENÍ

Aplikace byla již během svého vývoje postupně testována s každým přidaným uceleným zdrojovým kódem, který ji rozšiřoval. Vývoj aplikace bez provádění takovýchto testů by vedl k zhoršení celkové stability aplikace. Byla zvažována i možnost použití automatických testů, jak na straně klienta, tak na straně serveru. Nicméně implementace těchto automatických testů by vyžadovala hodně času kvůli potřebě pečlivého definování testovaných procesů a správných výstupů a z toho důvodu byl potřebný čas věnován spíše vývoji.

Klientská, serverová i databázová část aplikace byla od počátku nasazena v lokálním prostředí vývojového stroje. Testování serverové části, hlavně funkčnosti vytvořeného API, probíhalo za pomocí nástroje Postman [57], ve kterém byly vytvořeny a postupně simulovány všechny dostupné dotazy. K testování rozšířených funkcionalit byl využit osobní účet herní platformy Steam, na kterém byly zakoupeny určité předměty pro demonstraci funkcionalit uživatelského inventáře.

Pro nasazení celkové aplikace do online prostředí je zapotřebí několika kroků. Aplikace je rozdělena do třech jednotlivých částí, kde každá může být umístěna na jiném místě. Důležitým krokem je proto výběr a konfigurace jednotlivých hostingových služeb, které umožnují nasazení konkrétních částí. Pro nasazení frontendové části je zapotřebí upravit server URL odpovídající adrese, na které je zpřístupněná serverová část a provést připravený „buildovací script“ *npm run build*, který shromáždí a spojí všechny zdrojové kódy do několika výsledných souborů. Mezi nimi je i soubor *index.html*, který představuje vstupní bod klientské části. V serverové části je zapotřebí upravit konfigurační soubor *.env*, konkrétně řetězec určený pro připojení k externí MongoDB.

ZÁVĚR

Účelem této práce bylo vytvořit aplikaci, která umožní svým uživatelům získat lepší přehled o předmětech určených k obchodování z herní platformy Steam. Při určování funkčních požadavků kladených na aplikaci, bylo přihlídnuto k osobním potřebám, které agenda oficiální aplikace sama nenabízí a zároveň zpracovat a přizpůsobit ty, které již poskytuje.

Návrh serverové části odolává stanoveným požadavkům na výkon, stabilitu i rozšiřitelnost díky rozdělení její struktury do oddělených modulů. Návrh klientské části poskytuje přehledné informace v jednotném stylu, které vynikají dodržováním stanovených pravidel na designovou stránku aplikace.

Implementace obou částí zahrnuje povědomí o tom, jak celá aplikace vzniká od samého počátku. Zprostředkovává detailní popis počátečních inicializací obou částí. V serverové části zodpovídá otázku tvorby aplikačního rozhraní a komunikaci s databází. V klientské části zase tvorbu aktivních prvků, které uživatel používá pro interakci se systémem.

Vytvořená aplikace splňuje všechny stanovené požadavky, díky kterým funguje jako podpůrný nástroj pro analýzu dat, kterou by mohli její uživatelé použít v případě určování své obchodní strategie s těmito předměty. Aplikace aktuálně zpracovává a poskytuje funkcionality spojené s hrou Rust, která nabízí aktivní trh předmětů dostačující pro demonstraci funkčnosti aplikace. Systém bude v budoucnu nasazen na online servery, aby zajistil nepřetržitý přístup odkudkoliv na světě s připojením k internetu a bude implementovat i ostatní hry jako jsou například CSGO nebo Dota2, které poskytují obchodovatelné předměty, za účelem rozšíření okruhu svých uživatelů.

POUŽITÁ LITERATURA

[1] DHADUK, Hiren. An Ultimate Guide to Web Application Architecture. In: SIMFORM [online]. c2023, Last Updated April 27, 2023 [cit. 2023-11-07]. Dostupné z: https://www.simform.com/blog/web-application-architecture/

[2] What do client side and server side mean? | Client side vs. server side. CLOUDFLARE [online]. c2023 [cit. 2023-11-07]. Dostupné z: https://www.cloudflare.com/learning/serverless/glossary/client-side-vs-server-side/

[3] BANGA, Swapnil. What is Web Application Architecture? Components, Models, and Types. In: Hack.io [online]. [cit. 2023-11-07]. Dostupné z: https://hackr.io/blog/web-application-architecture-definition-models-types-and-more

[4] Client-Server Application. In: OOSE [online]. [cit. 2023-11-07]. Dostupné z: https://madooei.github.io/cs421_sp20_homepage/client-server-app/

[5] VERNEROVÁ, Sára. Front end vs. Back end - jaký je mezi nimi rozdíl? In: APITREE [online]. c2020 [cit. 2023-11-09]. Dostupné z: https://www.apitree.cz/blog/front-end-vs-back-end-jaky-je-mezi-nimi-rozdil

[6] COURSERA. What Does a Back-End Developer Do? In: Coursera [online]. c2023 [cit. 2023-11-09]. Dostupné z: https://www.coursera.org/articles/back-end-developer

[7] What is an API? POSTMAN, INC. Postman [online]. c2023 [cit. 2023-11-11]. Dostupné z: https://www.postman.com/what-is-an-api/

[8] What Is API and How Does API Work? Quick introduction. In: Flatlogic [online]. c2014-2023 [cit. 2023-11-13]. Dostupné z: https://flatlogic.com/blog/what-is-api-and-how-api-works/

[9] API Types and Protocols. Stoplight [online]. c2023 [cit. 2023-11-16]. Dostupné z: https://stoplight.io/api-types

[10] Types of APIs and Use Cases. In: Kong [online]. c2023 [cit. 2023-11-16]. Dostupné z: https://konghq.com/learning-center/api-management/different-api-types-and-use-cases

[11] GILLIS, Alexander. SOAP (Simple Object Access Protocol). In: TechTarget [online]. c2019-2023 [cit. 2023-11-16]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/SOAP-Simple-Object-Access-Protocol

[12] The SOAP Message Structure. In: *Flylib.com* [online]. c2008-2017 [cit. 2023-11-16]. Dostupné z: https://flylib.com/books/en/2.439.1.22/1/

[13] What Is SOAP API and Does It Still Work? In: MAKEUSEOF [online]. c2023 [cit. 2023-11-17]. Dostupné z: https://www.makeuseof.com/what-is-soap-api-does-it-work/

[14] What is the Difference Between SOAP and REST? AWS [online]. c2023 [cit. 2023-11-17]. Dostupné z: https://aws.amazon.com/compare/the-difference-between-soap-rest/

[15] What is a RESTful API? AWS [online]. c2023 [cit. 2023-11-18]. Dostupné z: https://aws.amazon.com/what-is/restful-api/

[16] GUPTA, Lokesh. What is REST. In: Restfulapi [online]. c2023 [cit. 2023-11-18]. Dostupné z: https://restfulapi.net

[17] What is REST API and how does it work? In: Geekboots [online]. c2023 [cit. 2023-11-18]. Dostupné z: https://www.geekboots.com/story/what-is-rest-api-and-how-does-it-work

[18] PANG, Avelon. REST Architectural Constraints. In: Medium [online]. 2021 [cit. 2023-11-18]. Dostupné z: https://medium.com/geekculture/rest-architectural-constraints-495a50ae0651

[19] MARICA, Sorin-Gabriel. Rest API architectural constraints. In: Web Scraping API [online]. c2021-c2023 [cit. 2023-11-18]. Dostupné z: https://www.webscrapingapi.com/rest-api-architecture-constraints

[20] CLINE, Brian. What are REST API's constraints? In: Brian Cline [online]. [cit. 2023-11-18]. Dostupné z: https://www.brcline.com/blog/what-are-rest-apis-constraints

[21] IJSTE - International Journal of Science Technology & Engineering [online]. I.J.S.T.E , INDIA, April 2016 [cit. 2023-11-19]. ISSN 2349-784X.

[22] JUVILER, Jamie. REST APIs: How They Work and What You Need to Know. In: HubSpot [online]. c2023 [cit. 2023-11-24]. Dostupné z: https://blog.hubspot.com/website/what-is-rest-api

[23] GraphQL [online]. c2023 [cit. 2023-11-24]. Dostupné z: https://graphql.org

[24] GraphQL is the better REST. In: *How to GraphQL* [online]. [cit. 2023-11-24]. Dostupné z: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

[25] BALDANIYA, Hitesh. Why and When to Use GraphQL. In: DZone [online]. [cit. 2023-11-24]. Dostupné z: https://dzone.com/articles/why-and-when-to-use-graphql-1

[26] What Is Database. Oracle [online]. c2023 [cit. 2023-12-07]. Dostupné z: https://www.oracle.com/database/what-is-database/

[27] 7 Key Factors to Consider When Choosing a Database for Your Application. In: Medium [online]. 2017 [cit. 2023-12-08]. Dostupné z: https://bootcamp.uxdesign.cc/7-key-factors-to-consider-when-choosing-a-database-for-your-application-a8ff14cd8305

[28] Database Scalability. ScyllaDB [online]. c2023 [cit. 2023-12-08]. Dostupné z: https://www.scylladb.com/glossary/database-scalability/

[29] ARNOLD, Joseph. Data Consistency Versus Data Integrity: Similarities And Differences. In: Databand [online]. c2023 [cit. 2023-12-08]. Dostupné z: https://databand.ai/blog/data-consistency-vs-data-integrity/

[30] ANDERSON, Benjamin a Brad NICHOLSON. SQL vs. NoSQL Databases: What's the Difference? In: IBM [online]. [cit. 2023-12-08]. Dostupné z: https://www.ibm.com/blog/sql-vs-nosql/

[31] What is a Relational Database (RDBMS)? Oracle Cloud Infrastructure [online]. c2023 [cit. 2023-12-08]. Dostupné z: https://www.oracle.com/database/what-is-a-relational-database/

[32] PAWLAN, David. Relational vs. Non-Relational Database: Pros & Cons. In: Aloa [online]. c2023 [cit. 2023-12-08]. Dostupné z: https://aloa.co/blog/relational-vs-non-relational-database-pros-cons

[33] SHELDON, Robert. ACID (atomicity, consistency, isolation, and durability). In: TechTarget [online]. c2005-2023 [cit. 2023-12-08]. Dostupné z: https://www.techtarget.com/searchdatamanagement/definition/ACID

[34] Relational vs. Non-Relational Databases. MongoDB [online]. c2023 [cit. 2023-12-09]. Dostupné z: https://www.mongodb.com/compare/relational-vs-non-relational-databases

[35] What is a Distributed Database? MongoDB [online]. c2023 [cit. 2023-12-09]. Dostupné z: https://www.mongodb.com/basics/distributed-database

[36] Distributed Database System. GeeksforGeeks [online]. 2023 [cit. 2023-12-09]. Dostupné z: https://www.geeksforgeeks.org/distributed-database-system/

[37] Data Replication in DBMS. GeeksforGeeks [online]. 2023 [cit. 2023-12-09]. Dostupné z: https://www.geeksforgeeks.org/data-replication-in-dbms/

[38] KUMAR, Barmanand. CAP Theorem and NoSQL Databases. In: Medium [online]. 2020 [cit. 2023-12-09]. Dostupné z: https://medium.com/@kumar.barmanand/cap-theorem-and-nosql-databases-589e26e15905

[39] Journal of Advanced Database Management & Systems [online]. 2. 2015 [cit. 2023-12-09]. ISSN 2393-8730. Dostupné z: https://www.researchgate.net/profile/Rakesh-Kumar-34/publication/280622043_Effective_Way_to_Handling_Big_Data_Problems_using_NoSQL_Database_MongoDB/links/55bf4b7208aed621de123708/Effective-Way-to-Handling-Big-Data-Problems-using-NoSQL-Database-MongoDB.pdf

[40] ROUSE, Margaret. JavaScript. In: Techopedia [online]. 2016 [cit. 2023-12-09]. Dostupné z: https://www.techopedia.com/definition/3929/javascript-js

[41] O'GRADY, Brian. JavaScript Demystified: Why You Should Learn This Essential Language. In: Code institute [online]. [2022] [cit. 2023-12-09]. Dostupné z: https://codeinstitute.net/global/blog/what-is-javascript-and-why-should-i-learn-it/

[42] ZOLOTAREV, Julia. What Is TypeScript? In: Built in [online]. c2023 [cit. 2023-12-09]. Dostupné z: https://builtin.com/software-engineering-perspectives/typescript

[43] SUN, Yan. Types vs. interfaces in TypeScript. In: LogRocket [online]. c2023 [cit. 2023-12-09]. Dostupné z: https://blog.logrocket.com/types-vs-interfaces-typescript/

[44] Introduction to Node.js. Node.js [online]. [2009] [cit. 2023-12-10]. Dostupné z: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

[45] DZIUBA, Anna. When and Why to Use Node.js: An In-Depth Guide for 2023. In: Relevant [online]. c2023 [cit. 2023-12-10]. Dostupné z: https://relevant.software/blog/why-and-when-to-use-node-js/#What_is_Nodejs_used_for

[46] HERBERT, David. What is React.js? Uses, Examples, & More. In: HubSpot [online]. c2023 [cit. 2023-12-10]. Dostupné z: https://blog.hubspot.com/website/react-js

[47] IGHORADO, Neo. Understanding Redux: A tutorial with examples. In: LogRocket [online]. 2023 [cit. 2023-12-10]. Dostupné z: https://blog.logrocket.com/understanding-redux-tutorial-examples/

[48] What is Git and Why Should You Use it? Noble desktop [online]. c1998-2023 [cit. 2023-12-10]. Dostupné z: https://www.nobledesktop.com/learn/git/what-is-git

[49] MERN Stack Explained. MongoDB [online]. c2023 [cit. 2023-12-13]. Dostupné z: https://www.mongodb.com/mern-stack

[50] User Authentication and Ownership. Steamworks [online]. [cit. 2023-12-13]. Dostupné z: https://partner.steamgames.com/doc/features/auth

[51] Passport-Steam. Npm [online]. [cit. 2023-12-13]. Dostupné z: https://www.npmjs.com/package/passport-steam

[52] Material Design [online]. [cit. 2023-12-13]. Dostupné z: https://m3.material.io

[53] Npm [online]. [cit. 2023-12-13]. Dostupné z: https://www.npmjs.com

[54] Install MongoDB. MongoDB [online]. c2023 [cit. 2023-12-13]. Dostupné z: https://www.mongodb.com/docs/manual/installation/

[55] MVC. In: Mdn [online]. c1998-2023 [cit. 2023-12-13]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/MVC

[56] MUI [online]. c2023 [cit. 2023-12-13]. Dostupné z: https://mui.com

[57] Postman [online]. c2023 [cit. 2023-12-13]. Dostupné z: https://www.postman.com