FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Front-end systému pro práci s investičními produkty
Student:	Simon Klibi
Vedoucí:	Ing. Jan Blizničenko
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Webové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je vytvořit uživatelské rozhraní k existujícímu systému pro práci s investičními produkty, se kterým bude UI komunikovat pomocí API endpointů.

Požadované funkce:
1. Kompletní správa uživatele (přihlašování, odhlašování, detail uživatele a administrátorská správa uživatelů).
2. Administrace produktů (administrátor bude ručně přiřazovat nezařazené nebo špatně zařazené produkty do správných kategorií).
3. Panel spotů a středových cen (tento panel bude vidět ve všech částech aplikace).
4. Listing drahých kovů a jejich detail (popis produktu, výčet jednotlivých prodejců a jejich ceníku, historie změn ceny v čase).

- Proveďte rešerši relevantních technologií, nástrojů, frameworků a knihoven.
- Seznamte se se současnou podobou architektury back-endu i stávajícího provizorního front-endu.
- Vytvořte návrh funkcionalit a stránek.
- Navrhněte architekturu.
- Navrhněte vizuální design.
- Proveďte implementaci.
- Otestujte a zdokumentujte své řešení.

Elektronicky schválil/a Ing. Jaroslav Kuchař, Ph.D. dne 16. února 2023 v Praze.

Bakalářská práce

FRONT-END SYSTÉMU PRO PRÁCI S INVESTIČNÍMI PRODUKTY

Simon Klibi

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jan Blizničenko
9. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Simon Klibi. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Simon Klibi. *Front-end systému pro práci s investičními produkty.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Seznam obrázků

Seznam výpisů kódu

Chtěl bych poděkovat vedoucímu práce Ing. Janu Blizničenko za jeho cenné rady a skvělý přístup při vedení této bakalářské práce.

Prohlášení

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

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

V Praze dne 9. května 2023 .....

Abstrakt

Tato práce se zaměřuje na návrh a implementaci frontendu pro investiční scraper. Cílem je vytvořit uživatelsky přívětivé rozhraní pro sledování cen drahých kovů. V rámci práce je analyzován doposud používaný frontend a to z pohledu zvolených technologií a architektury. Na základě informací zjištěných z analýzy je následně navržen frontend nový. Práce popisuje proces implementace SPA aplikace včetně návrhu architektury, wireframu a konečného UI. Dále práce popisuje specifické využití frameworku Next.js, které má za cíl maximalizovat optimalizaci pro vyhledávání i přes použití knihovny React a single-page application přístupu při vývoji a zároveň udržovat plynulost a dobrou user experience při užívání aplikace.

Klíčová slova frontend, React.js, Typescript, Next.js, SEO, SPA, single-page aplikace, scraper

Abstract

This thesis focuses on the design and implementation of a frontend for an investment scraper. The goal is to create a user-friendly interface for tracking precious metals prices. The thesis analyses the frontend used so far in terms of the chosen technologies and architecture. Based on the information found from the analysis, a new frontend is then designed. The thesis describes the implementation process of the SPA application including architecture design, wireframe and final UI. In addition, it also describes the specific use of the Next.js framework to maximize search optimization despite the use of the React library and single-page application development approach, while maintaining smooth running of the application and good user-experience.

Keywords frontend, React.js, Typescript, Next.js, SEO, SPA, single-page application, scraper

Seznam zkratek

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CMS	Content Management System
CSS	Cascading Style Sheet
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation
MPA	Multi-page Application
REST	representational state transfer
SASS	Syntactically Awsome Style Sheets
SEO	Search Engine Optimization
SPA	Single-page Application
SSG	Static Site Generation
SSR	Server Side Rendering
SW	Software
UI	User Interface
URL	Uniform Resource Locator
XML	The Extensible Markup Language

Úvod

V dnešní době se finanční trhy neustále vyvíjejí, a investování se tak stává stále populárnější a dostupnější variantou k běžnému bankovnímu spoření i pro širokou veřejnost. Pro investory je velmi důležité prakticky nonstop sledovat nejnovější tržní trendy a pohyby pozicí cen drahých kovů. Pro jejich správné rozhodování je tedy naprosto zásadní mít po ruce přesné a aktuální informace o cenách a celkové náladě na trhu.

Dostat se k informacím o cenách drahých kovů na burzách není v dnešní době internetu složité, co se však jeví jako velký problém, je srovnávání informací mezi jednotlivými burzami. Dohledávání těchto rozdílů a porovnávání jednotlivých aspektů je pak pro investora velmi zdlouhavé a náročné a zdržuje investora od samotné analýzy trhu a následného rozhodnutí či volby strategie.

Pro vyřešení tohoto problému proto vyvíjíme investiční scraper, který bude stahovat data o aktuálních pohybech napříč obchodníky a následně je sjednocovat do přehledného celku. Mým cílem bude uživateli poskytnout intuitivní a přehledné uživatelské rozhraní, které mu umožní procházet si konkrétní aktiva. Stěžejními částmi aplikace tedy budou přehledy jednotlivých aktiv včetně detailu a historie vývoje ceny, možnost registrace, přihlášení pro zachování uživatelského profilu a administrační panel uživatelů aplikace.

Vyvíjený investiční scraper (nástroj k vytěžování dat z webových stránek) se skládá ze dvou částí, frontendové a backendové, backendová část je vyvíjena v bakalářské práci Mikoláše Holého, částí frontendovou se bude zabývat tato bakalářská práce.

Cíle

Hlavním cílem práce je navrhnout a implementovat funkční frontend investičního scraperu, který bude poskytovat srovnávání cen drahých kovů a akcií napříč burzami.

Zprvu se práce zaměří na zadefinování funkčních a nefunkčních požadavků, které by aplikace po dokončení měla splňovat.

Poté se práce bude věnovat průzkumu, a to jak již existujícího frontendového řešení, tak API serverové části, pomocí kterého bude frontend s backendem komunikovat. Existující frontendové řešení má značné nedostatky, a to jak po stránce technologické, tak po stránce pokrytí funkcionalit. Tento průzkum se tedy bude snažit všechny tyto nedostatky identifikovat a předložit alternativní a technologicky rozumnější řešení. Na základě tohoto průzkumu se bude budovat frontend nový.

V další části práce se tedy zaměřím na rešerši relevantních nástrojů, technologií a frameworků tak, aby byl nový frontend postaven na moderních základech a zůstal tak po co nejdelší dobu plně funkční bez nutnost časté údržby či zásahu člověka.

V další kapitole se práce zaměří na návrh architektury. Zde přiblížím framework Next.js, navrhnu způsob komunikace s backendem a pokusím se vyřešit veškeré problémy, se kterými se dosavadní frontend potýkal.

Následovat bude vytvoření návrhu funkcionalit a wireframu, které udají základní představu o finálním vzhledu a funkcích výstupu této práce. Důraz bude kladen na intuitivní a jednoduché uživatelské rozhraní splňující všechny náležitosti správného UI. Na základě wireframu poté vznikne ve vhodném grafickém softwaru konečný vizuální design celé webové aplikace.

Dále budu popisovat proces implementace aplikace, kdy se zaměřím se zejména na zajímavé části kódu a na největší výzvy, které mě během vývoje potkaly.

Na závěr pak podrobím aplikaci uživatelskému testování, a to za pomoci předem zadefinovaných scénářů, které budou uživatelé následovat. Po uživatelském testu pak každý z uživatelů vyplní dotazník, jehož výsledky budou v práci zveřejněny.

Kapitola 1

Funkční a nefunkční požadavky

Na úplném začátku je potřeba vymezit si funkční a nefunkční požadavky kladené na aplikaci, ze kterých bude celý proces vývoje vycházet.

Funkční požadavky slouží k popisu vlastností systému z pohledu běžného uživatele. Popisují očekávané funkce a chování, které by měl konečný produkt splňovat [1]. Určují tedy, co by měl software umět, jak by měl reagovat na vstup uživatele a jak by se měl chovat v definovaných scénářích.

Nefunkční požadavky popisují vlastnosti softwaru jakožto celku a nezaměřují se na konkrétní funkce systému. Řadí se mezi ně například výkonnost, spolehlivost, rozšiřitelnost atd. [2].

1.1 Funkční požadavky

1.1.1 Registrace a přihlášení do systému

Uživatel se bude moci zaregistrovat do systému, registrace proběhne pomocí zadání potřebných údajů (jméno, příjmení, email, heslo). Po registraci bude uživateli umožněno přihlásit se do systému pomocí údajů zadaných při registraci.

Uživatel bude mít přístup k detailu svého účtu kde bude moci měnit informace o své osobě, případně svůj účet smazat.

1.1.2 Procházení produktů a detail produktů

Uživatel bude moci procházet jednotlivými produkty a filtrovat je. Každý produkt bude mít svoji stránku s detailem a s grafy vývoje cen v historii.

1.1.3 Panel spotů a středových cen

Ve všech částech aplikace bude viditelný panel se středovými cenami aktiv a kurzů měn.

1.1.4 Administrátorská správa uživatelů

Uživatel s rolí administrátor bude mít přístup do administrátorského rozhraní uživatelů, kde bude moci s uživateli provádět různé akce. Administrátor bude moci uživatelům mazat účty ze systému, obnovovat jim hesla a měnit role.

1.1.5 Administrace produktů

Uživatel s rolí administrátor bude mít přístup do administrátorského rozhraní produktů, ve kterém bude moci upravovat instance jednotlivých produktů, které byly scraperem zařazeny do chybné kategorie nebo nebyly zařazeny vůbec. Administrátor bude moci instanci produktu přeřadit pod správný produkt nebo přerušit vazbu s produktem původním a vytvořit tak produkt nový.

1.2 Nefunkční požadavky

Narozdíl od funkčních požadavků se nefunkční požadavky nezaměřují na to, co by měla aplikace umět z pohledu uživatele, tj. co aplikace dělá, jak reaguje na vstup od uživatele a tak dále, ale zaměřují se na vlastnosti jako jsou spolehlivost, bezpečnost, výkon či nárok na hardware. [2]

1.2.1 Dostupnost aplikace na URL

Tento požadavek zahrnuje potřebu, aby byla aplikace dostupná jakožto webová aplikace na určené webové adrese. Je požadováno, aby uživatel pro používání aplikace nemusel stahovat žádné dodatečné soubory ani software a aby interakce s aplikací probíhala čistě ve webovém prohlížeči.

1.2.2 Typescript

Je požadováno, aby byl z důvodu lepší orientace při vývoji v aplikaci použit Typescript, nadstavba nad jazykem JavaScript.

Kapitola 2
Analýza

Následující část textu se bude věnovat průzkumu aktuálního stavu frontendu a průzkumu backendu.

Průzkum frontendu se zaměří na analýzu užitých technologií a navrženou architekturu. Analyzovány budou stěžejní technologie a jejich vhodnost pro tento typ webové aplikace, vhodné technologie budou zachovány a u technologií nevyhovujících bude předložena vhodná alternativa.

V Průzkumu backendové části nebudou podrobně popisovány všechny endpointy, stavové kódy a odpovědi na ně. Namísto toho se zaměří na zobecněný popis jeho implementace, na knihovnu Swagger UI, kterou backend implementuje, na nejpodstatnější poznatky a na seznámení s pojmem mockování a jeho využitím v aplikaci.

2.1 Analýza – frontend

Při analýze frontendu jsem objevil zásadní nedostatky v návrhu architektury, dalším problémem je použití knihovny React bez jakéhokoliv podpůrného frameworku. Menším nedostatkem pak je použití knihovny MUI pro projekt s unikátním designem.

2.1.1 React

Aktuální frontend používá React, moderní javascriptovou knihovnu, která v mnoha směrech velmi ulehčuje práci a zpřehledňuje kód.

Použití této knihovny bych rád při vývoji nového frontendu zachoval, nicméně vzhledem k tomu, že investiční scraper je vyvíjen pro komerční účely, tak není podle mého názoru použití samotného Reactu bez podpůrných frameworků ideální volbou, a to hlavně z důvodů SEO (Search Engine Optimization). Nejvhodnější tedy bude čistý React doplnit frameworkem, který výrazně vylepší dohledatelnost aplikace ve vyhledávačích. Této problematice se věnuje kapitola [4.1 Server-side rendering: řešení problematiky React a SEO].

2.1.2 MUI

Při tvorbě vzhledu a uživatelského rozhraní byla v aplikaci použita UI (User Interface) knihovna MUI.

MUI je open-source UI knihovna pro React nabízející širokou škálu přizpůsobitelných komponent. MUI implementuje Google Material Design [3] (systém pravidel, který následuje osvědčené postupy pro návrh uživatelského rozhraní, vyvinutý společností Google [4]).

UI knihovny se hodí hlavně pro návrh informačních systému pro firmy, to jest například správa skladu, pokladny či docházkové systémy, při jejichž implementaci se dá za velmi krátkou dobu vybudovat seriózně vypadající uživatelské rozhraní [5]. Na stranu druhou pro implementaci komerčních webových aplikací, ve kterých je kladen velký důraz na to, aby byl vzhled uživatelského rozhraní unikátní a osobitý, mohou právě předstylované komponenty zbytečně ztěžovat a omezovat vývoj [6].

Pro vývoj aplikací zaměřujících se na unikátní design však nabízí MUI verzi MUI Base, která poskytuje všechny komponenty bez stylů, a design nechává čistě na vývojáři. [3]

Při implementaci frontendu jsem se ale rozhodl MUI vynechat. MUI implementující Material Design je pro vývoj unikátního designu zcela nevyhovující a MUI Base, který by mohl práci částečně usnadnit, by podle mého názoru pouze zbytečně přidal další technologii, kterou by se musel nový programátor pracující na projektu naučit. Namísto toho využiji pro stylování aplikace tradiční stylesheet soubory.

2.1.3 Architektura

Vzhledem k předpokládanému pokračování vývoje fronendu do budoucna je architektura značně nevyhovující, a to jak po stránce technologické, tak po stránce samotného návrhu.

Jeden z hlavních nedostatků architektury je pak jeho neuniverzálnost. Aktuální frontend ve všech částech aplikace počítá s předem daným počtem webů, ze kterých budou jednotlivé produkty scrapovány. Tyto weby jsou pak v aplikaci tzv. „hardcoded"(napevno vepsané do zdrojového kódu aplikace), a v případě jakékoliv změny seznamu webů na backendu, je nutno opětovně zasahovat do kódu klienta a přizpůsobovat ho nově přidaným webům.

Mezi další neduhy pak například patří nedostatečné dělení kódu do souborů, kdy kód jednotlivých komponent obsahuje i funkce, které by měly být separovány do samostatných souborů. Toto nedostatečné dělení pak komplikuje přehlednost a srozumitelnost. [7]

Při vývoji bych se tedy rád zaměřil jak na univerzálnost kódu (vyšší abstraktnost co se jednotlivých scrapovaných webů týče), tak lepší přehlednost (rozdělení kódů do logických celků a jednotlivých souborů).

```
        switch (price.dealer) {
            case Dealer.BESSERGOLD_CZ: {
                    chartData.Bessergold = price.price; break;
                    }
            case Dealer.BESSERGOLD_DE: {
                    chartData["Bessergold.de"] = price.price; break;
                    }
            case Dealer.ZLATAKY: {chartData.Zlataky = price.price; break;}
            case Dealer.SILVERUM: {chartData.Silverum = price.price; break;}
        }
```

Výpis kódu 2.1 Scrapované weby jsou fixně vepsané do zdrojového kódu

2.2 Analýza – backend

Backend implementuje klasické REST API, blíže popsáno v kapitole [4.2.1 Rest API]. Komunikace bude probíhat jak na bázi získávání zdrojů (GET), tak na bázi upravování zdrojů na straně backendu (POST, DELETE, PATCH).

2.2.1 Swagger UI

Swagger UI je framework používaný k vizualizaci API zdrojů a slouží jako skvělý doplněk při komunikaci mezi frontend vývojářem a backend vývojářem. Pro každý endpoint Swagger UI podrobně popisuje implemntované metody, způsob jejich použití a status kódy, které lze při komunikaci očekávat. [8]

Díky jeho použití na backendu jsem tak velmi rychle bez zbytečného zatěžování backend vývojáře pochopil, jak bude komunikace s backendem probíhat a na co si při implementaci dát pozor.

2.2.2 Mocked endpointy

Při procesu vývoje frontendu nebyly naimplementovány některé API endpointy. U těchto nenaimplementovaných endpointů byl použit tzv. „API Mocking“.

Při API Mockingu se na straně serveru implementuje endpoint bez potřebné logiky a imituje se jeho funkčnost. Endpoint pak vrací na jednotlivé requesty falešná data, která na první pohled vypadají realisticky. [9]

API Mocking se používá v případě, že je frontend vývojář s implementací aplikace oproti backend vývojáři napřed. V tuto chvíli by tak musel s implementací frontendu vývojář čekat na dodatečnou implementaci backendu.

Z endpointů, se kterými pracuji, byly namockovány následující:

GET /api/v2/person/me Endpoint vracející přihlášeného uživatele (identifikován podle JWT tokenu). Tento endpoint vrací nezávisle na jwt tokenu stále stejného uživatele (uživatel se jménem „Tomáš Dummy“).

POST /api/v2/auth Endpoint určený k autentizaci do aplikace v těle requestu očekává přihlašovací údaje uživatele. Backend pak ověří přihlašovací údaje a v závislosti na jejich správnosti vrací 200 OK (autentizace proběhla v pořádku) nebo 401 Unauthorized (špatně zadané přihlašovací údaje). V tomto případě je kontrétně namockován JWT token, který aplikace vrací. (Je vracen stále stejný jwt token pro všechny uživatele.).

2.2.3 Absence filtrace na straně serveru

Jedním z požadavků na bakalářskou práci bylo implementovat filtraci jednotlivcýh produktů.

Běžnou praxí bývá, že proces samotné filtrace probíhá na straně serveru a jednotlivé parametry filtrace vkládá frontend do query parametrů za URL. Díky tomuto návrhu pak klientská strana nemusí stahovat veškerá data ze serveru, a dostává tak jen to, co v danou chvíli potřebuje.

Na straně serveru však tento postup implementován není, a filtrace jednotlivých produktů podle váhy, ceny či výkupní ceny tak musí probíhat na straně klienta. Filtrace je blíže popsána v kapitole [6.2 Komponenta <Filter/>]

Kapitola 3
Rešerše technologií

3.1 MPA vs. SPA

Při návrhu atchitektury je zprvu důležité rozmyslet si způsob implementace. Zde se nabízí dvě možnosti.

První možností je moderní SPA (single page application), ve které si klient stáhne prázdnou html stránku s balíčkem javascriptu, který poté na straně klienta vyrenderuje veškerý obsah stránky.

Druhou možností je pak MPA (multi page application), ve které se většina obsahu generuje na straně serveru a klient dostává již finální verzi stránky.

Obě tyto možnosti v následující část přiblížím a popíšu jejich výhody a nevýhody. Na závěr pak popíšu technologii, kterou jsem pro implementaci zvolil, a důvody proč.

3.2 SPA (single page application)

SPA (Single Page Application) je webová aplikace, ve které klient při požadavku na konkrétní URL obdrží pouze prázdnou HTML stránku a javascriptový kód. Po načtení souborů se na straně klienta spustí stažený javascriptový kód a obsah stránky se vyrenderuje. Součástí prvotního spuštění javascriptového kódu mohou být také požadavky na API, které poskytuje aplikaci potřebný obsah, co se dat týče. Tato implementace umožňuje uživateli aplikaci používat, aniž by bylo potřeba při každé interakci stránku znovu načítat. [10]

3.2.1 Výhody SPA

SPA aplikace má mnoho výhod, mezi hlavní lze zařadit rychlost. SPA aplikace se načítá pouze jednou, a to při prvním navštívení stránky. Díky tomu jsou všechny zdrojové kódy (HTML, CSS, Javascript) načteny pouze jednou.

Dále SPA aplikace umožňuje efektivní cachování na straně klienta (klient si stáhne celou aplikaci do paměti), v případě že v aplikaci není nutná komunikace s backendem, lze díky této vlastnosti aplikaci používat i bez připojení k internetu.

Díky faktu, že se strana klienta stará o veškeré vykreslování a sběr dat, lze aplikaci jednoduše debugovat pomocí propracované vývojářské konzole v Google Chrome. [11]

3.2.2 Nevýhody SPA

Mezi hlavní nevýhody SPA patří horší SEO (Search Engine Optimization). Důvodem je pak to, že celý koncept SPA stojí na javascriptu a bez jeho podpory aplikace nemá šanci správně fungovat. Naprostá většina vyhledávačů má však stále velké problémy se spouštěním scriptů [12] a tento fakt se pak výrazně podepisuje na finálním SEO.

Další nevýhodou je, že SPA aplikace mají pomalejší prvotní načítání, jelikož je potřeba, aby klient z hostujícího serveru stáhl v podstatě celou aplikaci, a to i v případě, že chce uživatel přistoupit pouze na jednu jedinou stránku [11].

3.3 MPA (multi page application)

MPA (Multi-Page Application) je tradičnější přistup k tvorbě webových stránek. V případě jakékoliv změny na stránce, ať už z důvodu interakce uživatele s obsahem, nebo přechodu na jinou podstránku, je potřeba zaslat request na server a vyžádat si render nové stránky, obsahující aktualizovaný obsah. [11]

3.3.1 Výhody MPA

Výhod MPA oproti SPA je hned několik. Jednou z hlavních je právě výrazně lepší SEO, a to díky tomu, že SEO crawler (bot, který navštěvuje stránky a shromažďuje o nich informace [13].) při requstu na webovou stránku obdrží již zcela vyrenderovanou stránku, a rovnou tak může začít s indexováním, bez nutnosti spouštění jakéhokoliv javascriptového kódu.

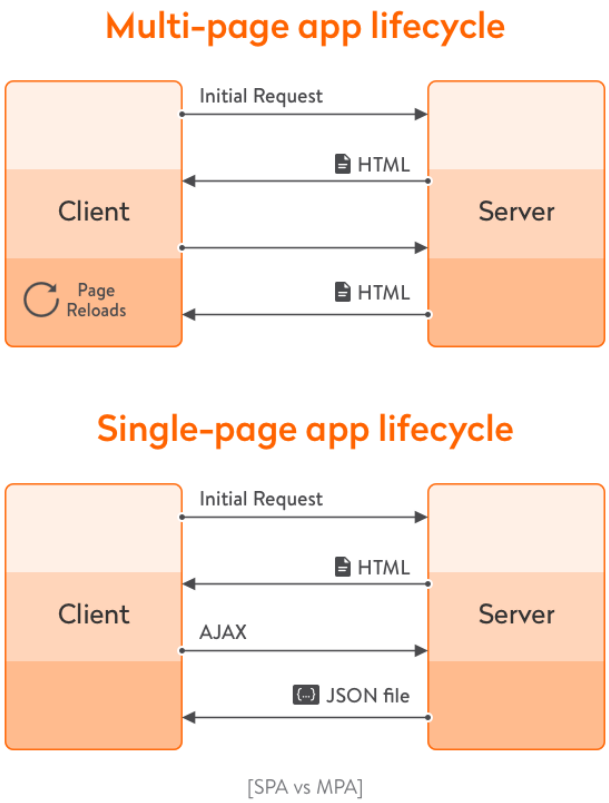
Další výhodou může být například přirozenější navigace pro uživatele, jelikož má každá navštívená stránka svoji vlastní URL a chová se jako samostatná entita.

Dále poskytuje rychlejší prvotní načítání stránky, protože se stahují pouze data, která jsou potřebná pro běh konkrétní načtené sekce. [11]

3.3.2 Nevýhody MPA

Hlavní nevýhodou MPA je, že při jakékoliv změně obsahu na stránce, vyvolané například interakcí uživatele, si klient musí vyžádat od serveru znovu kompletní webovou stránku, která musí být serverem znovu vyrenderována, tentokrát však se změněným obsahem. Tento způsob značně znepohodlňuje uživatelskou interakci s aplikací a zpomaluje práci s ní.

Velké nevýhody z MPA plynou i pro vývojáře, v případě MPA je frontend velmi blízce provázán s backendem, což vede k v dnešní době nežádoucím monolitickým aplikacím. Při výrazné změně na backendu (změna frameworku, přechod k modernějšímu programovacímu jazyku …), je pak u MPA, na rozdíl od SPA, potřeba velkých zásahů do frontendu. [11]



Obrázek 3.1 Multi-page aplikace vs Single-page aplikace [14]

3.4 Závěr

Pro implementaci aplikace jsem vybral SPA, jelikož se jedná o moderní přístup k tvorbě webových stránek a v dnešní době, kdy se od MPA neustále více ustupuje, bude aplikace v následujících letech mnohem lépe udržitelná.

Protože se však jedná o aplikaci s potenciálem komerčního využití, nabízí se otázka SEO problematiky. Aktuálně již existují technologie, které si dokaží se SEO poradit i u single page aplikací. Jedna z nich, konkrétně server-side rendering, bude využita při vývoji této aplikace a bude se jí věnovat kapitola [4.1 Server-side rendering: řešení problematiky React a SEO].

3.5 Single page nástroje

Pro vývojáře může být tvorba SPA složitá a náročná. Naštěstí již v současné době existuje široká škála nástrojů, které zjednodušují vývoj těchto aplikací. Mezi nejpopulárnější z nich patří React.js, Angular a Vue.js [15]. Každý z těchto nástrojů má své výhody a nevýhody, proto každou z těchto technologií podrobněji popíšu a na závěr vyberu nejvhodnější z nich pro tento projekt.

Při porovnávání těchto tří technologií jsem vycházel hlavně z článku [16] a z osobních zkušeností.

3.5.1 Angular

Angular je framework pro webové aplikace dostupný zdarma, vyvíjený společností Google, kompletně založený na typescriptu. Jedná se o nástupce frameworku AngularJs (vyvinutý v roce 2010). Tento framework je ze všech tří zmíněných nejmladší. Světlo světa spatřil v roce 2016 a rychle nabyl popularity napříč komunitou webových vývojářů [17]. Mezi výhody Angularu patří:

Nativní podpora SSR Angular nativně podporuje renderování obsahu na straně serveru (SSR) [18].

Dokumentace Angular má detailně popsanou dokumentaci a koncepty, na kterých tato technologie stojí, jsou v ní jednoduše a srozumitelně popsány.

3.5.2 React

React je opensource knihovna dostupná zdarma a vyvinuta společností Meta (dříve Facebook) v roce 2013.

Používá se jako základ při tvorbě webových aplikací a je založena na komponentizaci. Ze všech tří zmíněných nástrojů se těší největší popularitě mezi vývojáři (42.1%) [15].

Paralelně k Reactu sloužícímu k vývoji webových aplikací vyvíjí společnost meta i React Native, framework používaný k vývoji aplikací pro Android TV, IOS, Android, Windows etc. React Native funguje na podobných principech jako React, díky čemuž stačí ovládnout pouze jednu z těchto technologií. Naučit se druhou je pak velmi snadné. [19]

Hlavní výhody jsou:

Oblíbenost a rozšíření React je mezi vývojáři nejoblíbenějším frontendovým nástrojem. Díky tomu je na internetu k dispozici spousta návodů a postupů pro řešení problému. Z jeho oblíbenosti zároveň plyne vysoká poptávka po této technologii na pracovním trhu.

Cross platform React Native s Reactem sdílí velkou část konceptů (hooks, komponenty…) [20]. Díky tomu je pro vývojáře relativně jednoduché přejít od tvorby webového rozhraní k tvorbě mobilních aplikací bez nutnosti učit se novou technologii a zároveň je možno velkou část kódu webové aplikace jednoduše použít i při tvorbě mobilní aplikace.

3.5.3 Vue.js

Vue.js je javascriptový open-source framework inspirovaný Angularem vytvořený v roce 2014 bývalým pracovníkem ve společnosti Google Evanem You. Na rozdíl od většiny tehdejších technologií bylo velmi lehké naučit se s ním pracovat. Zároveň v něm bylo obsaženo to nejlepší z Angularu a byly odstraněny jeho nedostatky. Díky tomu si VUE.js získal velkou popularitu napříč vývojáři.

Jedna z jeho hlavních výhod je kompaktnost, jeho velikost je cca 20 kb. [21]

3.5.4 Závěr

Každá z výše uvedených technologií se zakládá na podobném principu (sdružování částí aplikace do komponent) a každá má své klady a zápory. Pro implementaci webové aplikace jsem však vybral knihovnu React. A to právě z důvodu jeho širokého rozšíření ve vývojářské komunitě, díky čemuž se můžu spolehnout, že React bude ještě řadu let jedním z hlavních nástrojů pro tvorbu webových aplikací. Z toho plyne dlouhodobá udržovatelnost aplikace a zároveň široké pokrytí problémů na internetových fórech.

3.6 React frameworky s podporou SSR (Server-Side Rendering)

Při implementaci aplikace využiji renderování na straně serveru (SSR). Jak SSR funguje a proč bude při implementaci klíčový, je popsáno v kapitole [4.1 Server-side rendering: řešení problematiky React a SEO]. V následující pasáži se budou věnovat popisu některých frameworků nativně podporujících SSR v Reactu.

3.6.1 Next.js

Next.js je framework postavený na Reactu, kombinuje řadu nastrojů a funkcí usnadňující vyývoj single page aplikací. Jendou z hlavních funkcí Next.js je právě server-side rendering.

Mezi další výhody patří integrace s nástroji třetích stran, díky čemuž lze využívat knihovny, jako je například Redux pro správu stavu aplikace nebo knihovna Styled Components, která usnadňuje stylování komponent používáných na vícero místech.

Next.js je oblíbený framework v komunitě vývojářu a byl použít při vývoji mnoha populárních aplikací jako je například Airbnb či Netflix. [22]

3.6.2 Remix

Remix je React framework umožnující server-side rendering. Na rozdíl od Next.js se Remix zaměřuje čistě na SSR bez podpory SSG (Static Site Generation), kdy se stránka generuje během buildu aplikace [23]. Oproti Next.js má zároveň menší uživatelskou základnu (souzeno podle počtu hvězdiček na github repozitářích).

3.6.3 Gatsby

Gatsby je framework postavený na reactu. Primárně se zaměřuje na static site generation. Mezi jeho výhody patří například dostupnost pluginů, které umožňují integraci s CMS (Content Management System), jako jsou například Wordpress nebo Contentful. Jeho obliba je stejně jako u frameworku Remix nižší než u Next.js.

3.6.4 Závěr

Při rozhodování se mezi Next.js, Remix a Gatsby.js jsem se rozhodl pro Next.js, a to zejména z důvodu předchozích zkušeností s tímto frameworkem a zároveň z důvodu popularity mezi vývojáři, která u Next.js násobně převyšuje zbylé dva frameworky.

Obrázek 3.2 Porovnání popularity frameworků podle github repozitářů, Duben 2023

Kapitola 4

Návrh architektury

Při návrhu architektury se snažím dosáhnout co největší efektivity, co se vývoje týče, a zároveň co nejpřehlednějšího členění a struktury kódu a adresářů. Všechny koncepty a návrhy, které považuji za důležité nebo něčím zajímavé, jsem se proto pokusil co nejsrozumitelněji popsat pomocí textu a diagramů v následujících pasážích.

Nejprve se v této kapitole pokusím přiblížit technologii, na které celá aplikace stoji. Touto technologií je SSR (server-side rendering) popíšu samotné SSR a pozitiva, která z jeho využití plynou. Následně přiblížím způsob komunikace backednu s klientem, zaměřím se na strukturu adresářů a na závěr popíšu typy toků dat, které aplikace používá.

4.1 Server-side rendering: řešení problematiky React a SEO

Jak již bylo zmíněno v [3.2.2 Nevýhody SPA], jedna ze zásadních nevýhod SPA je problém s optimalizací pro vyhledávače (SEO).

Je potřeba, aby byla stránka před započetím indexace a procházení obsahu kompletně načtena, což je v případě SPA, potažmo React.js zásadně zkomplikováno tím, že crawler při požadavku na SPA aplikaci obdrží prázdnou webovou stánku s javascript kódem, který ji vyrenderuje až u klienta.

Velká část SEO crawlerů se ale bohužel v případě velkého objemu aplikace neobtěžuje s jeho exekucí vůbec nebo dává stránce snížené ohodnocení z důvodu dlouhého načítání stránky. (Crawler čeká, než se obsah na straně klienta vyrenderuje.) [12] Naštěstí již existuje technologie, která právě tento problém řeší. Nazývá se server-side rendering.

4.1.1 Server-side rendering (SSR)

Jak již z názvu vyplývá, server-side rendering umožňuje aplikaci renderovat obsah na straně serveru. V této kapitole se budu věnovat SSR u single-page aplikací, nikoliv u multi-page aplikací, u kterých je SSR použito implicitně. Celá technologie funguje tak, že server při požadavku klienta „poskládá“ statické HTML u sebe a pak ho společně s javascriptovým kódem předává klientu. Ten pak místo dosavadní prázdné stránky dostává předrenderované statické HTML s obsahem. [24]

V praxi pak server-side rendering funguje následovně [25]:

1. **Požadavek klienta** – Klient zasílá HTTP request na server.

2. **Zpracování požadavku serverem** – Server obdrží request od klienta a začne s renderováním prvotního HTML. V případě, že je pro render stránky vyžadováno fetchování dat z API endpointu, provede ho a poté obsah doplní do HTML souboru.

3. **Odeslání odpovědi klientu** – Server zasílá klientu odpověď, jejíž součástí je vyrenderované statické HTML.

4. **Obdržení odpovědi** – Klient obdrží odpověď serveru a zobrazí statické HTML.

5. **Hydratace** – Statické HTML jako takové by bez javascriptu nebylo interaktivní (uživatel by dělat nemohl nic jiného než si stránku pouze prohlédnout, případně zaslat na server běžný HTML formulář), klient proto stáhne potřebné javascriptové soubory a na používaných elementech nastaví event listenery.

6. **Hydratace II.** – V případě, že server používá cache, klient né vždy obdrží nejaktuálnější obsah. V tomto případě dochází ještě k dodatečnému fetchování dat z api endpointů

Jak je nejspíše zřejmé z popisu výše, výhody server-side renderingu jsou značné. Klient při requestu obdrží již hotový HTML dokument, který může okamžitě ukázat uživateli. Díky tomu se zrychlí celkové načítání stránky, uživatel nemusí čekat, než mu prohlížeč stránku vyrenderuje, a SEO crawler, který stránku navštíví, dostává hotový HTML dokument. Z toho pak může začít se samotnou indexací.

4.2 Komunikace s backendem

Jak již bylo popsáno v [2.2 Analýza – backend], backend implementuje REST API, pomocí kterého bude frontend s backendem komunikovat. Na straně frontendu jsem zároveň použil knihovnu React Query, která značně zjednodušuje správu dat pomocí cachování.

4.2.1 Rest API

Komunikace frontendu s backendem bude probíhat pomocí takzvaného REST API neboli Representational State Transfer Application Programming Interface. Podle knihy [26] je Rest API populární architektura používaná ve vývoji webových stránek, jejímž hlavním cílem je komunikace a získávání dat ze serveru pomocí standartních HTTP metod (GET, POST, PUT, DELETE). Jednotlivé zdroje jsou identifikovány pomocí unikátního URI a data jsou v naprosté většině přenášena ve formátu JSON nebo XML.

Hlavní výhoda REST API je jeho nezávislost na platformě, jedná se pouze o jakousi sadu pravidel popisující, jak má frontend s backendem komunikovat, a jeho implementace je plně ponechána na vývojářích.

GET safe metoda (metoda, která neupravuje zdroj na straně serveru) sloužící k získaní zdroje ze serveru

POST metoda používaná k vytvoření nového zdroje na straně serveru

PUT idempotentní metoda (vícero stejných požadavků má stejný účinek jako jeden) k upravování již existujícího zdroje na serveru

DELETE metoda, která smaže určený zdroj ze serveru

4.2.2 React Query

React query je knihovna zjednodušující správu dat v React aplikacích. Umožňuje snadno stahovat, ukládat a aktualizovat data do paměti a odstiňuje vývojáře od složité správy stavů.

React query se stará o to, aby byla data vždy aktuální, a pomocí chytré správy minimalizuje počet requestů na server, čímž zároveň snižuje zatěžování serveru a datový přenos mezi klientem a serverem. [27]

Zejména z důvodu ulehčení práce s daty jsem pro komunikaci skrze REST API vybral právě knihovnu React Query.

Způsob použití knihovny React Query při implementaci je blíže popsán v [6.1.2 React Query].

4.3 Stylování

Stylování je zásadní součástí tvorby webových stránek a aplikací. Namísto klasického CSS jsem při stylování použil preprocesor SASS a doplnil ho o vlastní UI Komponenty. Díky tomu je kód přehledný a zbytečně se v něm neopakují stejné koncepty, které jsou využity na vícero místech.

4.3.1 SASS

SASS (Syntactically Awsome Style Sheets) je CSS preprocesor umožňující stylovat aplikaci a za pomocí pokročilých nástrojů, které nejsou v běžném CSS k dispozici, výrazně ulehčovat práci.

Mezi klíčové vlastnosti patří například zjednodušený zápis selektorů, definování proměnných či používání tzv. mixinů, díky kterým lze zadefinovat opakovaně používané styly a na jednom místě je pak hromadně upravovat. [28]

SAAS je velmi užitečný nástroj, který v dnešní době společně s ostatními CSS preprocesory preferuje jako náhražku tradičního CSS naprostá většina vývojářů [29].

4.3.2 UI komponenty

Pro často používané UI prvky jsem se rozhodl vytvářet samostatné komponenty s oddělenými styly a částečně tak kopírovat způsob používání UI knihoven. V praxi jsou pak UI komponenty striktně odděleny od zbytku kódu a fungují jako samostatná jednotka.

Oddělení UI komponent umožňuje jednodušší orientaci v projektu, z toho plynoucí rychlé úpravy bez zbytečného procházení zdrojového kódu a jednoduchou znovupoužitelnost v jiných projektech, kdy stačí překopírovat složku s komponentami či vytvořit samostatný npm modul.

```
import React from "react";
import styles from "./TextField.module.scss"

interface IProps extends React.HTMLProps<HTMLInputElement>{
    label?:string
}
const TextField = (props:IProps) => {
    const {label:inputLabel} = props
    return (
        <div className={styles["text-field"]}>
            { inputLabel &&
                    <label className={"subSubtitle"} HTMLFor={inputLabel}>
                        {inputLabel}
                    </label>
                }
            <input id={inputLabel} {...props}/>
        </div>
    )
}

export default TextField
```

Výpis kódu 4.1 Příklad UI komponenty TextField

4.4 Data Flow

Jak již bylo zmíněno, existují dva hlavní způsoby, jak získávat data v React aplikacích – pomocí server-side renderingu způsobem, který používá framework Next.js, nebo pomocí fetchování všech dat a renderu na straně klienta. Každý z přístupů má své klady a zápory.

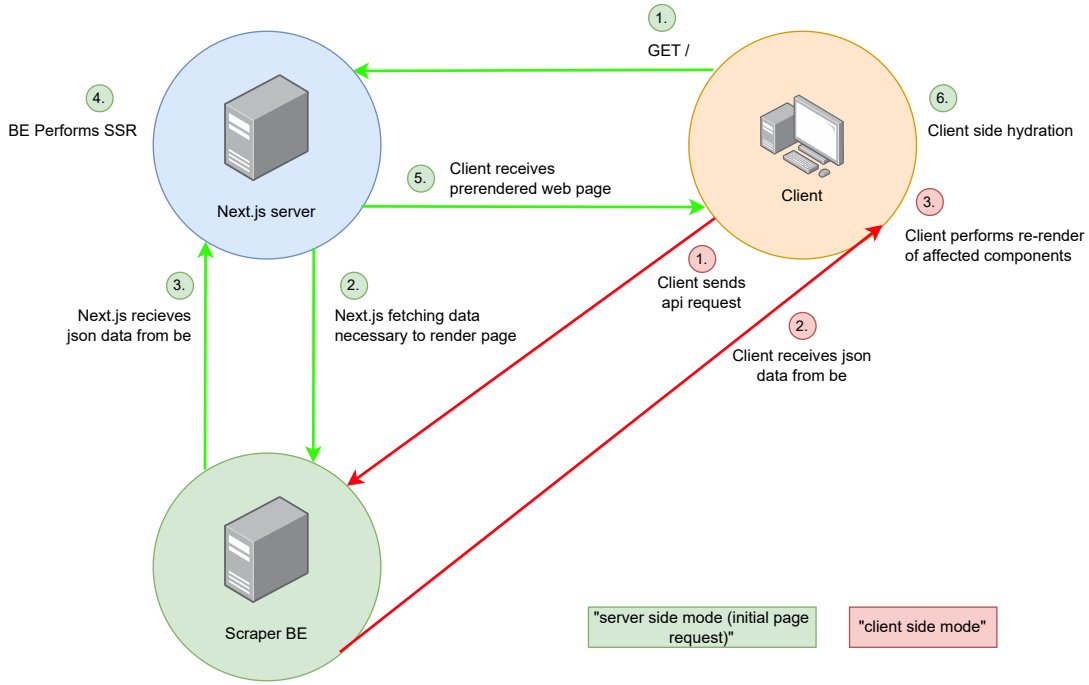
Díky vlastnostem čistého Reactu je user experience velmi dobrá, uživatel nemusí při každé změně čekat na načtení nové stránky a re-render probíhá pouze u obsahu, který byl uživatelovou interakcí změněn.

Next.js pak sice částečně pokulhává, co se plynulosti týče, ale oproti čistému Reactu má mnohem lepší výsledky SEO.

Ideálně by tedy byla nejlepší aplikace disponující jak plynulostí Reactu, tak i dobrou SEO, kterou nám poskytuje framerwork Next.js. Při navrhování aplikace jsem se tedy rozhodl obě tyto výhody zkombinovat.

Zkombinování obou výhod tedy funguje následovně: Při prvotním načtení stránky se využije SSR, který nám poskytuje Next.js a celý obsah je vyrenderován na straně serveru, poté se však „přepne“ na client-side režim a veškerá komunikace a render probíhá čistě na straně klienta, jako tomu je v běžné React aplikaci.

Tento návrh přináší značná pozitiva, a to jak menší zatížení serveru hostujícího aplikaci (server musí zpracovávat pouze zlomek requestů vyvolávajících SSR), tak výrazně vyšší plynulost aplikace, a z toho plynoucí lepší user experience. S tímto návrhem však přicházejí i kompromisy, tím hlavním je možné částečné snížení kvality SEO, podle mého názoru ale v tomto případě klady ve formě zvýšené plynulosti a nižší zátěže serveru značně převažují nad zápory.



Obrázek 4.1 Diagram způsobů získávání dat (vlastní tvorba)

4.5 Struktura adresářů

Strukturu adresářů jsem navrhoval tak, aby byl celý projekt rozdělen do logických celků a aby bylo, pokud možno, co nejjednodušší se v něm rychle zorientovat jak pro nově příchozí, tak pro projektu znalé vývojáře.

Při návrhu jsem částečně vycházel jak ze zkušeností z předchozích zaměstnání, tak z dokumentace Next.js (page adresář).

Mezi dva základní adresáře patří adresář `/src` a adresář `/public`, přičemž v adresáři `/src` se nacházejí veškeré zdrojové kódy a styly použité při vývoji aplikace a v adresáři `/public` se nacházejí veřejně dostupné zdroje, které aplikace využívá, zejména pak obrázky a ikony.

4.5.1 /src/components

Adresář pro všechny React komponenty použité při vývoji aplikace, tento adresář je dále rozdělen na následující složky.

atom Malé komponenty, které samy o sobě slouží pouze k zobrazování základních HTML elementů bez složitější logiky. Nachází se zde například komponenty pro zobrazování avatarů či cenovek.

molecule Obsahuje komponenty středního rozsahu, které již obsahují jisté funkcionality a často se skládají právě z komponent z adresáře `atom`.

organism Velké komponenty jak po stránce počtu elementů, tak po stránce logiky, tyto komponenty již samy o sobě tvoří vetší sekce stránek a jsou schopny na stránce fungovat samostatně. Mezi organism komponenty patří například `Filter` komponenta pro filtrování produktů nebo komponenty zobrazující vývoj cen produktů.

ui Soubor s UI komponentami, blíže popsáno v [4.3.2 UI komponenty].

Každou komponentu samostatně pak reprezentuje složka s jejím názvem, tato složka obsahuje 2 soubory:

1. `[ComponentName].tsx` - Typescript soubor s komponentou
2. `[ComponentName].module.scss` – Soubor se styly pro danou komponentu (Nachází se zde pouze styly právě pro komponentu, jejíž jméno soubor nese, tudíž nenastává situace, kdy soubory se styly přidružené jedné komponentě nějakým způsobem ovlivňují styly komponenty druhé).

4.5.2 /src/page

Tento adresář musí pro funkčnost aplikace dodržovat strukturu zadefinovanou frameworkem Next.js [22]. Právě ze struktury této složky se poté berou jednotlivé URL cesty.

Tato složka se skládá z podsložek, které obsahují jednotlivé obsahy stránek, tyto podsložky pak definují cestu, pomocí které se lze ke stránce dostat. Mapování cesty v adresáři se pak 1:1 podobá finální cestě za URL.

Při routování se nám v Next.js nabízejí 3 možnosti. V případě že se ve složce nachází soubor se jménem index.tsx, router ho považuje za defaultní stránku pro danou cestu.

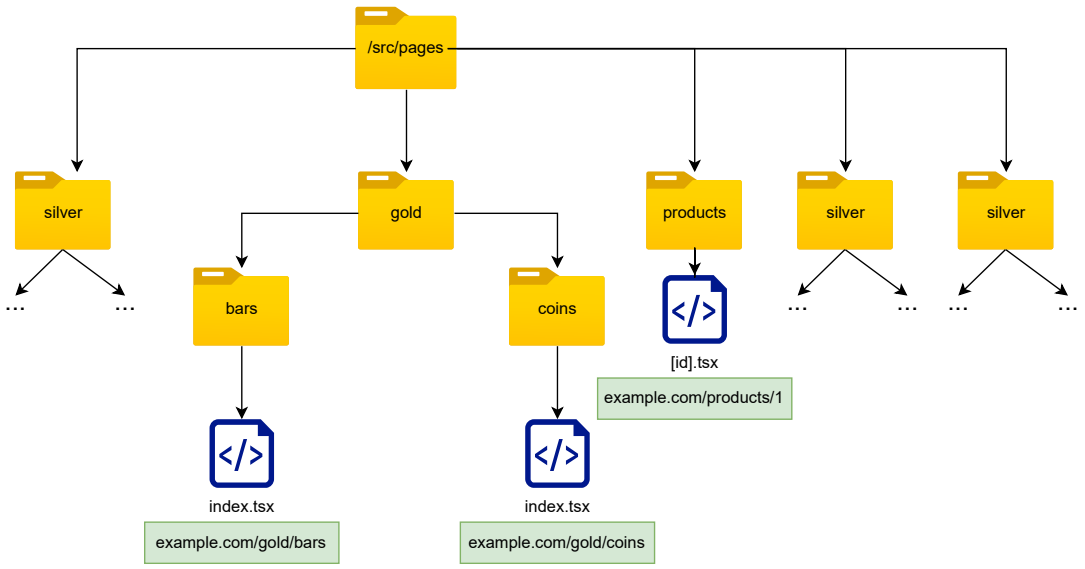
- `/src/page/index.tsx → example.com/`

Pro dynamické routování (například při potřebě definovat v cestě id produktu) se do finálního adresáře píše obsah do souboru `[ [nazev_parametru] ].tsx`, k tomuto parametru pak lze přistupovat při vývoji v komponentě stránky.

- `/src/page/products/[id].tsx → example.com/products/{1,2,3,...}`

Třetí možností pak je vložit do jedné ze složek soubor s libovolným názvem, Next.js pak vezme název tohoto souboru jako další část cesty.

- `/src/page/about.tsx → example.com/about`



Obrázek 4.2 Diagram použití page directory v projektu (vlastní tvorba)

4.5.3 Další adresáře

/src/helper Nachází se zde soubory obsahující pomocné funkce, užívané napříč aplikací. Jedná se například o funkce pro řazení produktů, filter funkce či formátování ceny do stringu.

/src/static Tento adresář obsahuje veškeré statické proměnné užívané napříč aplikací. To umožňuje měnit všechny parametry, ze kterých aplikace čerpá, na jednom místě.

/src/types Nachází se zde definice Typescript typů, enumů a interfaců, které jsou používány na více než jednom místě.

/src/styles Globální styly aplikace, styly jednotlivých stránek, SCSS proměnné a mixiny.

Kapitola 5
UI Návrh

Návrh UI (User Interface) je při vývoji naprosto stěžejní částí, je to hlavní prostředek komunikace mezi aplikací a uživatelem, a je tedy velmi důležité, aby na uživatele udělalo pozitivní dojem, a aplikace mu tak reálně ulehčovala práci namísto přidělávání problémů. I naprosto průlomová aplikace s perfektně navrženým backendem pak může zcela ztroskotat na neochotě uživatelů ji používat z důvodu nevhodně navrženého uživatelského rozhraní.

V Této kapitole proto popíšu osvědčené postupy, které jsem při návrhu UI aplikoval.

Nejprve budu citovat zásadní pravidla pro tvorbu UI popsaných již v minulém století, kterých by se měl každý designer držet.

Následovat bude podkapitola o wireframu, hrubém náčrtu aplikace. Vysvětlím, co to wireframe je a proč je při návrhu UI důležitý.

Závěrem se budu věnovat UI. Popíšu software, který jsem při návrhu použil, a představím samotnou finální podobu aplikace.

5.1 Pravidla správného UI

Při návrhu UI (User interface) jsem se snažil dodržovat 10 zásadních pravidel pro správné UI, jejichž cílem je, pokud možno co nejvíce zlepšit user experience při interakci s aplikací. Tyto zásadní pravidla byla definována Jakobem Nielsenem a Rolfem Molichem [30].

1. **Simple and natural dialogue** *Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility. All information should appear in a natural and logical order.*

2. **Speak the user's language** *The dialogue should be expressed clearly in words, phrases and concepts familiar to the user, rather than in system-oriented terms.*

3. **Minimize the users' memory load** *The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.*

4. **Consistency** *Users should not have to wonder whether different words, situations, or actions mean the same thing.*

5. **Feedback** *The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.*

6. **Clearly marked exits:** *Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue.*

7. **Shortcuts** *Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users.*

8. **Good error messages:** *They should be expressed in plain language (no codes), precisely indicate the problem. and constructively suggest a solution.*

9. **Prevent errors** *Even better than good error messages is a careful design that prevents a problem from occurring in the first place*

10. **Help and documentation** *Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, be focused on the user's task, list concrete steps to be carried out, and not be too large.*

5.2 Wireframe

Podle [31] je wireframe grafický návrh, který se zaměřuje na to, na jaké části bude aplikace rozdělena. Udává priority jednotlivých částí, pozici jednotlivých prvků (tlačítka, inputy, formuláře) a popisuje funkcionality a chování, které by měla aplikace po dokončení splňovat.

Po dokončení wireframu bychom tedy měli mít jasno o:

1. počtu a obsahu stránek, které bude aplikace po dokončení mít,
2. informaci o přiděleném prostoru pro jednotlivé komponenty,
3. funkcionalitách jednotlivých prvků na stránce,
4. vizuálním zobrazení cest mezi stránkami (na jakou stránku se dostanu ze stránky),
5. rozložení prvků na stránce,
6. celcích, které se v aplikaci budou opakovat na vícero místech.

Wireframe se nezaměřuje na grafický vzhled stránky jako takový, tj. neřeší barvy, fonty ani vzhledy prvků, slouží pouze k hrubé představě o finálním vizuálu aplikace. Výstup poté lze použít jako pomyslné vodítko, ze kterého se vychází při návrhu samotného UI designu.

Zvláště u React aplikací pak spočívá další velká výhoda existence dobře vytvořeného wireframu ve faktu, že programátoři mohou začít vývojem jednotlivých komponent, které stránka bude mít, aniž by měli k dispozici vizuální návrh.

Při návrhu wireframu jsem se snažil uživatele co nejvíce odstínit od faktu, že se nachází na webové stránce, a co nejvíce vzbuzovat pocit plnohodnotné desktopové aplikace. Důležité prvky, jako jsou například ceny jednotlivých produktů, typy produktu vizualizované obrázkem či důležitá tlačítka mají, co se přiřazeného prostoru na stránce týče velkou prioritu. Pro zjednodušení UI jsou často místo popisků užívány ikony, které podle mého názoru intuitivně popisují, čeho se daný prvek týká.

Zároveň jsem se wireframe snažil navrhnout minimalisticky, tj. ponechat na stránce pouze důležité informace a vyvarovat se přehlcení uživatele zbytečnými informacemi, které by mohly potenciálně strhávat jeho pozornost.

5.3 UI Design

Na rozdíl od wireframu se UI (User Interface) Design věnuje celkové estetice a finálnímu vzhledu aplikace, tj. barvy prvků, vizuální styl, typografie, grafika a další detaily.

Při jeho návrhu je důležité mít po ruce již hotový a zpracovaný wireframe, díky čemuž získá designer přehled o rozložení aplikace a může mu celkový design přizpůsobit. UI Design se samozřejmě dá dělat i bez zpracovaného náčrtu, tento postup však velmi nedoporučuji, protože se člověk při návrhu samotného designu až moc zaměřuje na estetický vzhled a samotné logické rozdělení aplikace často vypouští. Při zjištění nepotřebnosti prvku či nevyhovujícího návrhu prvku na stránce je pak zbytečně smazána několikahodinová, často dosti titěrná práce (na rozdíl od wireframu, kde jsou všechny prvky načrtnuté abstraktněji, a jejich smazání či přepracování je méně bolestivé).

Návrh UI Designu již nelze zvládnout za pomocí pouhého papíru a tužky (zhruba 94 % respondentů z více než 4000 dotazovaných uvedlo, že při návrhu UI designu používají nějaký software [32]). Je třeba sáhnout po profesionálním softwaru, který se přímo na tento obor zaměřuje.

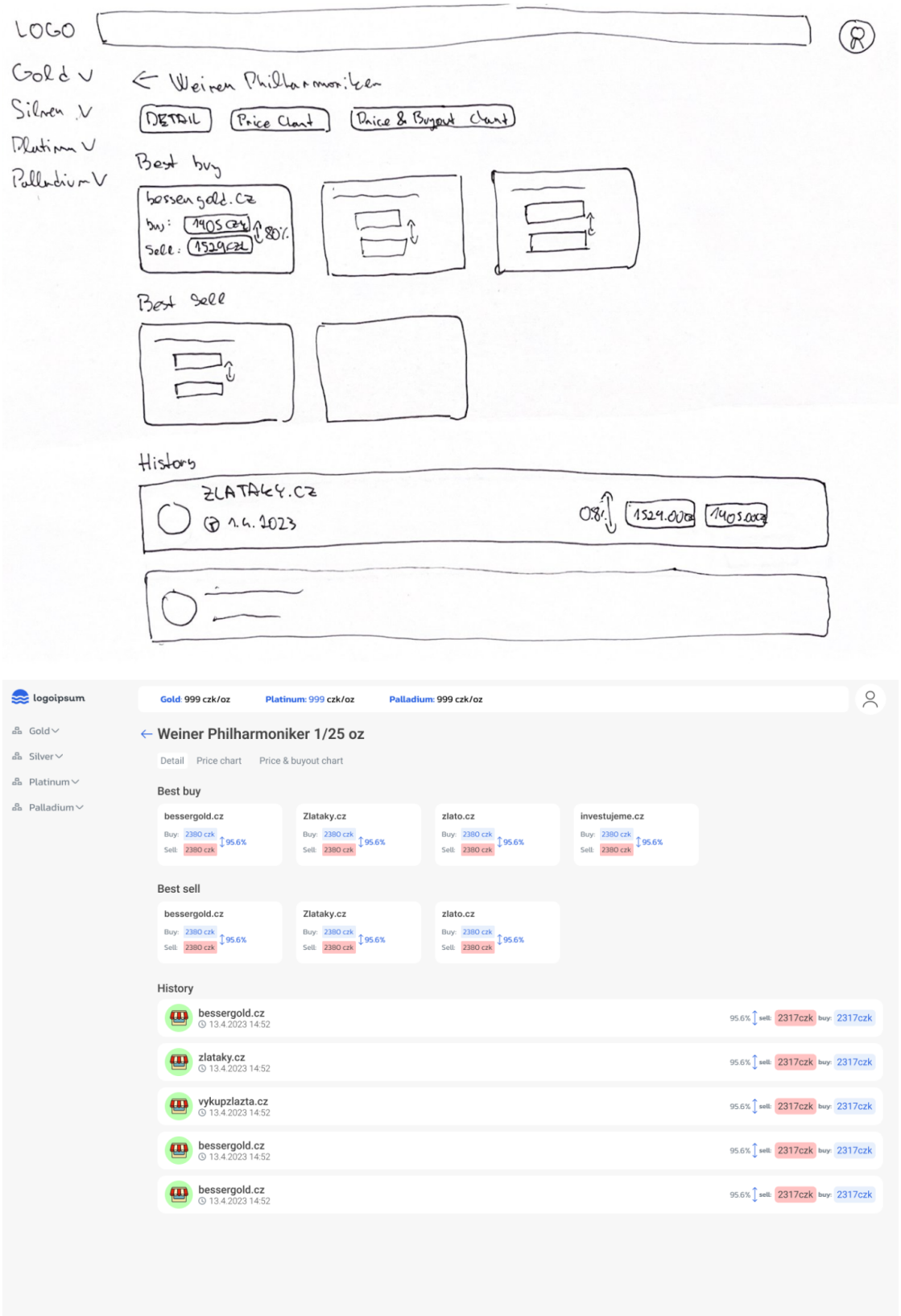
Mezi dva nejvíce používané SW pro tvorbu UI Designu pak patří Figma a Adobe XD [32]. Oba tyto softwary poskytují nepřebernou sadu velmi užitečných nástrojů, a to od komponentizace jednotlivých prvků (vytváření jakési šablony pro často opakované prvky) po generování css stylů připravených k použití.

Pro návrh designu jsem si vybral software Figma, a to nejen díky velmi pozitivní předchozí zkušenosti (při mé práci v seznam.cz byl používaný pro naprosto všechny designové návrhy), tak i díky faktu, že je pro osobní použití zcela zdarma.

Při návrhu uživatelského rozhraní jsem se snažil dodržovat moderní trendy v oblasti tvarů barev a celkového vzhledu, a vytvořil jsem tak esteticky přitažlivé pracovní prostředí.

Zároveň jsem se zaměřil na to, aby výraznost prvků na stránce odpovídala jejich důležitosti, tj. pro důležitější prvky jsem použil větší písmo a výraznější barvy, pro méně důležité naopak.

Částečně jsem se zároveň inspiroval designem úspěšné finanční aplikace Revolut [33], a to zejména v barvách a segmentaci obsahu do bloků.



Obrázek 5.1 Wireframe listingu produktů a výsledný design

Kapitola 6
Implementace

V této kapitole se zaměřím na samotnou implementaci webové aplikace. Popsány budou implementace těch nejdůležitějších částí, zajímavé koncepty, které jsem při vývoji použil, a největší výzvy, které mě během samotného programování aplikace potkaly.

Nejprve objasním způsob komunikace frontendu s backendem, nastíním použití Next.js a popíšu, jak jsem využil nástroje React query pro chytřejší správu dat v paměti. Díky tomu jsem dosáhl minimalizace počtu requestů na backend.

Společně s tím se změřím na největší výzvu, kterou jsem si sám pro sebe připravil, a tím je použití knihovny react query na straně backendu i frontendu zároveň a implementace rozdílného chování aplikace, co se renderu a fetchování dat týče, před a po prvotním načtením stránky.

Na závěr přiblížím komponentu `<Filter/>`, ve které jsem se se snažil dosáhnout pokud možno co nevyšší abstrakce a možnosti použitelnosti na vícero místech nezávisle na vstupních datech.

6.1 Komunikace mezi frontendem a serverem

Jak bylo popsáno v kapitole [4.4 Data Flow], komunikace s Next.js serverem probíhá ve dvou režimech, client-side a server-side. Při vývoji jsem tudíž musel zajistit, aby všechny datové interakce v aplikaci fungovaly přesně podle diagramu 4.1. Před vysvětlením samotného řešení tohoto problému je ale třeba alespoň povrchově popsat koncepty dvou technologií, které pracování s daty při vývoji značně ovlivnily. Konkrétně se jedná o již zmíněné Next.js a React Query

6.1.1 Práce se fetchováním dat v Next.js

Při práci s daty v aplikaci je stěžejní již zmiňovaný server side rendering, na jehož koncep se zaměřovala kapitola [4.1.1 Server-side rendering (SSR)]. Server-side rendering v next.js nefunguje automaticky. Při programování je třeba nějakým způsobem frameworku sdělit, jaká data a jakým způsobem mají být zpracována a fetchována na straně serveru, a naopak, jaká data mají být zpracována a fetchována na straně klienta. Pro tuto situaci nabízí Next.js 3 způsoby přístupu k problematice. Přístup, který jsem použil při vývoji, podrobně popíši a zbylé dva krátce shrnu. Tyto přístupy jsou následující.

1. `getStaticProps` – Přístup používaný v případě, že jsou data, se kterými aplikace pracuje, neměnná. V praxi pak tento přístup funguje tak, že Next.js při procesu buildu finální aplikace všechna potřebná data stáhne z backendu a na stálo je uloží. V případě requestu klienta na server pak vrací stále stejná data, která byla získána v průběhu buildu.

2. `getStaticPaths` – Přístup velmi podobný getStaticProps. Server taktéž získává data pouze při buildu aplikace, a navíc umožňuje ukládat data pro stránky s dynamickými cestami (všechny možné kombinace pro dynamické routy, vedoucí na existující stránku, se musejí deklarovat před procesem buildu aplikace).

3. `getServerSideProps` – Přístup používaný v případě, že se potřebná data v reálném čase mění, a je tedy potřeba je pravidelně fechovat. Tento přístup je implementovaný tak, že se Next.js server při každém requestu klienta dotazuje na backend pro potřená data.

Z těchto tří přístupů jsem při vývoji použil `getServerSideProps`, jelikož jako jediný vyhovuje scénáři, kdy se data na straně backendu mění v reálném čase. V případě užití `getServerSideProps`, (u ostatních přístupů je postup velmi podobný), je třeba v souboru s next.js stránkou (soubory v adresáři /src/page, struktura podrobněji popsána v kapitole [4.5.2 /src/page]) společně s komponentou reprezentující tuto stránku exportovat i funkci s názvem `getServerSideProps`.

Funkce `getServerSideProps` je stejná jako všechny ostatní javascriptové funkce s rozdílem toho, že se její obsah vykonává pouze na straně serveru. Z toho nám plynou značné výhody. Mezi ty, co považuji za hlavní, patří například následující:

Node.js environment Kód neběží v prohlížeči, ale v prostředí Node.js. Jsou zde tedy dostupné systémové funkce, které by v prohlížeči běžně dostupné nebyly (manipulace se soubory, přístup k procesům atd.).

Přístup k databázím Díky bodu výše v této funkci můžeme k databázím přistupovat napřímo, a ne pouze pomocí API Endpointů.

Bezpečnost Na straně backendu je možno k důležitým api endpointům, či api endpointům s potenciálními zranitelnostmi povolit přístup pouze Next.js serveru, který se na ně pak dotáže právě v `getServerSideProps`.

Funkce `getServerSideProps` přijímá argument `context`, pomocí kterého lze přistupovat například k url parametrům v requestu klienta nebo k samotné odpovědi klienta. Vracet pak musí objekt obsahující props, které jsou následně předány komponentě stránky, a ta k nim pak může volně přistupovat.

```
function Page({ data }) {
     // Komponenta stránky (zde můžeme pracovat se získanými daty)
}
export async function getServerSideProps(context) {

  // tento kód se provádí pouze na straně serveru
  const res = await fetch(`http://example.com/api/data`)
  const data = await res.json()

  // props, které obdrží komponenta stránky
  return { props: { data } }
}

export default Page
```

Výpis kódu 6.1 Ukázka použití funkce getServerSideProps a způsob získávání dat v komponentě stránky

6.1.2 React Query

React Query jsem již obecně popisoval v kapitole [4.2.2 React Query], nyní ale podrobněji popíšu, jak tato technologie funguje v praxi, jak zvládá ušetřit aplikaci zbytečné requesty pomocí cachování a jak se používá.

Pro použití je nejprve potřeba inicializovat `QueryClient`, ten slouží jako úložiště všech aktuálních requestů na api, které jsou v aplikaci provedeny, a udržuje informace o stavu těchto requestů a odpovědí na ně. Zároveň je třeba inicializaci Query klientu spojit s obalením zpravidla celé aplikace komponentou `<QueryClientProvider/>`. Právě díky této komponentě má pak aplikace přístup ke všem datům uloženým v cache ve všech částech kódu.

Každý get request je pak nutno provádět pomocí funkce `useQuery`, která v argumentech přijímá:

1. `queryKey` klíč který identifikuje request, pomocí něj se pak identifikují stejné queries v cache,
2. funkci provádějící request,
3. `options` konfiguraci requestu, ve které se dá nastavit například doba platnosti requestu před nutností re-fetche.

```
const fetchProducts = async () => {
    const res = await fetch("http://example.com/api/products");
    return res.json();
};
const response = useQuery("ALL_PRODUCTS", fetchProducts, {staleTime: 3000});
```

Výpis kódu 6.2 Příklad použití useQuery

Samotná Data se z cache postupně mažou, a to buď po uplynutí doby platnosti definované při odesílání requestu, nebo ručně pomocí funkce `invalidateQueries`, která bere jako argument právě `queryKey` zadaný v době requestu. Ve chvíli požadavku na api pak React Query zkontroluje, jestli již není v cache uložen platný request se stejným `queryKey`. Jestliže ano, vrátí data z cache, v opačném případě data stáhne a uloží do cache.

6.1.2.1 Použití React Query

React Query v aplikaci používám jak na straně serveru ve funkci `getServerSideProps`, tak na straně klienta. Bylo proto třeba zajistit, aby byla data získaná na straně serveru již uložena v instanci `QueryClient`, která je dostupná na straně klienta. Běh kódu na straně serveru a na straně klienta se ale liší a je oddělený, pokud je na straně serveru objekt inicializován, tak je tento objekt přístupný pouze na straně serveru a klient k němu nebude mít přístup.

Pro zachování cache je ale potřeba, aby měla aplikace na straně klienta přístup k instanci QueryClient se zachachovanými requesty, které byly získány na straně serveru. Toto ale z výše uvedeného důvodu není automatické, a aplikace by tak bez patřičného ošetření tohoto problému okamžitě po načtení u klienta automaticky fetchovala všechna data znovu.

Tento problém lze vyřešit pomocí React Query funkce dehydrate, která vyjme veškerá data uložená v cache a vloží je do běžného javascript objektu. Tento objekt poté na straně serveru transformuji do lehce odesílatelných dat pomocí nativní funkce `JSON.stringify`.

```
export const getServerSideProps = withCSR( (context) => {

    const {query} = context

    queryClient.prefetchQuery<IProduct>([QUERY_KEYS.PRODUCT_DETAIL, 1190],
                            async () => ProductService.getById(query.id))

    return {
        props:{
            dehydratedState: JSON.stringify(dehydrate(queryClient))
        }
    }
})
```

Výpis kódu 6.3 Dehydratace QueryClient cache

Data se poté v stringified podobě odesílají klientovi. Tato data jsou následně v komponentě `App` (výchozí komponenta aplikace, kterou Next. js používá k inicializaci stránek a procházejí skrze ni veškerá data vrácená v `getServerSideProps` funkci) načtena do nově vzniklé instance `ReactQuery` pomocí komponenty `Hydrate`.

```
export default function App({ Component, pageProps }: AppProps) {
  return (
      <QueryClientProvider client={queryClient}>
          <Hydrate state={parseJson(pageProps.dehydratedState)}>
                <Layout>
                    <Component {...pageProps} />
                </Layout>
          </Hydrate>
          <ReactQueryDevtools initialIsOpen={false} />
      </QueryClientProvider>
  )
}
```

Výpis kódu 6.4 QueryClient hydratace

6.1.3 Omezení SSR

Závěrem bylo potřeba, aby byla funkce `getServerSideProps` volána pouze při prvotní komunikaci mezi klientem a serverem, jak je popsáno v diagramu 4.1. Tohoto chování jsem docílil pomocí obalení funkce `getServerSideProps` následující funkcí:

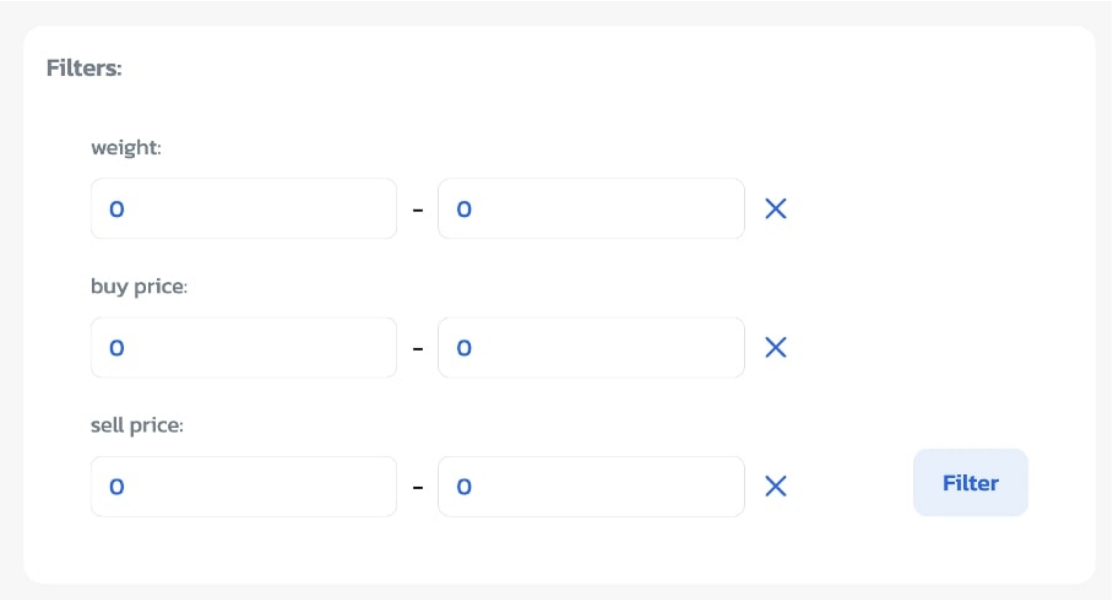
```
export const withCSR = (next) => async (ctx) => {


    const isCSR = ctx.req.url?.startsWith('/_next');

    if (isCSR) {
        return {
            props: {},
        };
    }

    return next?.(ctx)
}
```

Výpis kódu 6.5 Funkce withCSR

Funkce `withCSR` ověřuje, jestli byla navigace mezi stránkami v aplikaci prováděna na straně klienta nebo na straně serveru, a to pomocí podmínky `isCSR`, která zjišťuje, jestli URL adresa, na kterou uživatel přistupuje začíná na `"_next"` [34]. Díky tomu jsou pak zpracovány pomocí SSR pouze prvotní requesty na stránku.

6.2 Komponenta <Filter/>

Filter komponenta v aplikaci slouží k filtrování produktů a uživatelů. Jelikož není na straně backendu implementována filtrace, kde by backend filtroval produkty, například podle query parametrů, musel jsem implementovat na straně klienta i samotný proces filtrování.



Obrázek 6.1 Vizuál filtrovací komponenty

Díky správnému návrhu jsem dosáhl jisté úrovně abstrakce, díky které není třeba pro uživatele a produkty vytvářet filtry zvlášť, ale lze tuto komponentu použít pro obojí. Zároveň v případě, že

v budoucnu přibudou další entity s nutností filtrace, může tato komponenta posloužit i v tomto novém případě bez nutnosti zasahovat do jejího zdrojového kódu.

Následující popis konceptu je kromě popisu samotného použití místy zjednodušený pro snadnější pochopení.

Celý princip filtrování je založen na nativní javascriptové funkci filter, která je dostupná pro každé pole. Funkce `filter` očekává v argumentu callback funkci, do které funkce `filter` postupně dosadí všechny elementy v poli. Callback funkce pak vrací `true` nebo `false` v závisloti na tom, jestli má být daný element přítomný ve finálním vyfiltrovaném poli, či ne. [35]

Komponenta `<Filter/>` očekává v `props` popis jednotlivých filtrovacích polí, které následně vyrenderuje. Každé toto filtrační pole pak při změně v inputu vygeneruje callback funkci, s filtrační podmínkou, kterou si komponenta `<Filter/>` ukládá.

Po stisknutí tlačítka filter pak komponenta `<Filter/>` postupně volá nativní funkci `filter`, na pole poskytnuté od rodiče se všemi uloženými callback funkcemi, které byly poskytnuty od jednotlivých filtračních polí.

Jak již bylo zmíněno, tato implementace umožňuje použít komponentu `<Filter/>` na pole libovolných objektů určených k filtraci.

Pro správnou funkci filtrovací komponenty je nutné předat v props:

Items Pole položek určených k filtraci.

setFilteredItems Callback funkce, která v argumentu přijímá pole vyfiltrovaných položek. Tuto funkci komponenta volá po stisknutí tlačítka filter a do jejího argumentu vkládá vyfiltrované pole.

filterSchemas Pole objektů s interfacem FilterSchema, které popisuje, co má komponenta `<Filter/>` obsahovat za pole a jak tyto pole filtrují seznam.

```
const ProductsList = (props:IProps) => {
    const {products} = props
    const [filteredProducts, setFilteredProducts] = useState(products)
    return(
        <>
        <List>
            <Filter
                items={products}
                filterSchema={FILTER_SCHEMAS.product}
                setFilteredItems={setFilteredProducts}
                />
            {filteredProducts?.map(product =>
                            <ProductThumbnail
                            product={product}
                            key={product.id}
                            />
            )}
        </List>
        </>
    )

}
```

Výpis kódu 6.6 Použití filtrační komponenty pro filtrování seznamu produktů

6.2.1 filterSchema

Na rozdíl od props `items` a `setFiltereditems`, jejichž význam je přímočarý, je prop `filterSchema` komplexnější záležitostí. Pokusím se tedy koncept, na kterém `filterSchema` stojí, podrobněji vysvětlit.

Aktuálně lze ve filtrační komponentě využít dva různé typy filtrace. Těmi jsou `ScaleFilterField` sloužící pro případ, kdy je potřeba položky v seznamu vyfiltrovat podle rozmezí dvou hodnot (např. cena od, cena do), A `ValueFilterField` sloužící pro případy, kdy je potřeba položky filtrovat podle jednoho pole (např. název =). Popisy těchto typů jsou však čistě hypotetické, samotný význam pak určuje vývojář podle přiřazené filtrovací funkce.

Pro oba tyto typy jsem vytvořil Typescript interface, `filterSchema` je pak objekt, který může být jak typu `ScaleFilterField`, tak typu `ValueFilterField`.

```
// /src/types/filterFields.tsx
export interface IScaleFilterField{
    fieldLabel:string,
    minFilter : (lowerBoundValue : any) => (value:any) => boolean,
    maxFilter: (upperBoundValue: any) => (value:any) => boolean

}
export interface IValueFilterField{
    fieldLabel:string,
    filter: (equalTo:any) => boolean

}
export type FilterSchema = (IValueFilterField | IScaleFilterField)[]
```

Výpis kódu 6.7 Interface filtračních polí

Atribut `fieldLabel`, který je pro oba typy stejný a reprezentuje název pole filtru (weight, buy, etc.). Zajímavější jsou však atributy `minFilter` a `maxFilter` v `IScaleFilterField` resp. `filter` ve `IValueFilterField`.

Právě v těchto atributech se očekávají funkce řídící filtrování, nazývejme je funkcemi generujícími. Tyto generující funkce přijímají jako argument libovolnou hodnotu (za tento argument se dosazuje hodnota, kterou uživatel vyplní do inputu filtru). Tato funkce pak v returnu vrací callback funkci, která bude použita při filtrování.

```
function maxSellTotalPriceFilter(value){
    return (product:IProduct) => {
        return  product.bestPrice?.redemption < value
    }}
```

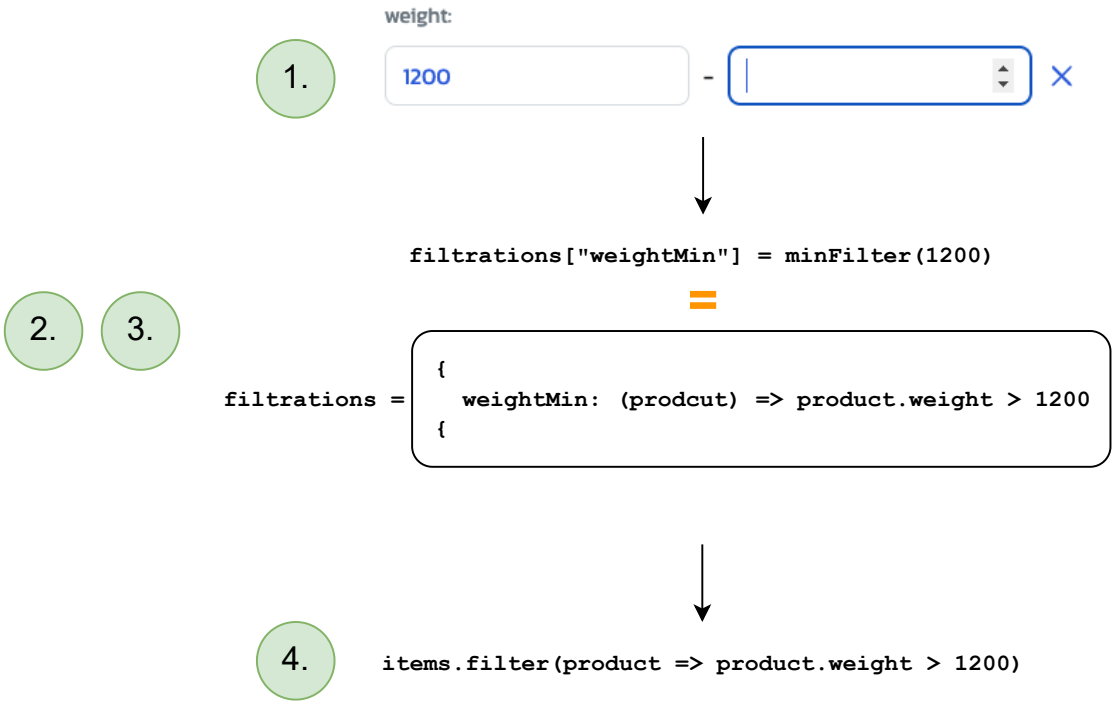
Výpis kódu 6.8 Funkce generující filtrační funkci

Z logiky věci potřebuje `ValueFilterField` jednu filtrovací funkci (obsahuje jediný input) a `ScaleFiletField` dvě (jednu pro dolní hranici a druhou pro horní hranici).

Celý proces tedy funguje následovně:

1. Uživatel zadá hodnotu do inputu filtru.
2. filtrovací pole podle zadané hodnoty vygeneruje callback funkci.

3. `<Filter/>` Komponenta si drží objekt, který obsahujuje všechny callback funkce vygenerované filtračními poli, do tohoto objektu je pak uložena nově vygenerovaná callback funkce.
4. Po kliknutí na tlačítko „filtrovat“ komponenta `<Filter/>` projde pole filtrovacích funkcí, aplikuje je na položky předané v props a pole vyfiltruje.



Obrázek 6.2 Diagram procesu filtrace (vlastní tvorba)

Kapitola 7

Uživatelské testování

Uživatelské testování je stěžejní součástí při vývoji uživatelského rozhraní, pomáhá odhalit nelogicky navržené celky a chyby [36]. Podle [36] by se zároveň mělo jednat o primární způsob, kterým bude jakékoliv uživatelské rozhraní testováno, a to na základě důkladnosti spolehlivosti srozumitelnosti a celkových nákladů.

Dalším možným způsobem testování je tzv. heuristická analýza, která spočívá v analýze uživatelského rozhraní odborníky, kteří mají znalosti o lidském chování, a dokáží tak odhadnout pravděpodobné nedostatky, se nimiž by se uživatelé mohli při užívání aplikace potýkat.

Uživatelské testování mnou vyvíjené aplikace bude probíhat za pomocí předem určených scénářů, které bude muset uživatel při testování následovat. do testu budou zapojeni uživatelé různých věkových kategorií, s různými znalostmi problematiky, kterou vyvíjená aplikace řeší. Před začátkem testu bude uživateli vysvětleno, k čemu aplikace slouží a jaké má funkce (investiční scraper, srovnávání cen…).

U scénáře týkajícího se přesunu linku bude uživateli krátce vysvětleno, co je po něm požadováno, kromě toho nebude mít žádný z uživatelů možnost seznámit se před testem se samotou aplikací či s uživatelským manuálem. testování tedy bude záviset na čisté intuici každého uživatele a intuitivnosti mnou navrženého uživatelského rozhraní.

Po uživatelském testování bude každý z uživatelů instruován k vyplnění krátkého dotazníku, který bude vedle intuitivnosti UI hodnotit i samotné pocity uživatele během obsluhy aplikace.

Pro uživatelské testování jsem vybral následující vzorek uživatelů:

1. Muž, 24 let, žádné znalosti o investování, pokročilé zkušenosti v IT,
2. Žena, 23 let, žádné znalosti o investování, zkušenosti s běžným užíváním pc,
3. Muž, 23 let, pokročilé znalosti o investování, pokročilé zkušenosti v IT,
4. Muž 22 let, žádné znalosti o investování, zkušenosti s běžným užíváním pc,
5. Žena 56 let, žádné znalosti o investování, zkušenosti s běžným užíváním pc.

7.1 Scénaře Testování

Vytvořenými scénáři budou procházet všichni uživatelé, obsaženy jsou jak scénáře určené pro běžného uživatele (přihlášení, listing, atd.), tak scénáře určené pro správce systému (administrace uživatelů, správa produktů).

Před samotným začátkem testování bude systém obsahovat reálná data scrapovaných cen a předem vytvořené uživatele k testování administrace.

1. **Registrace a přihlášení** Přejděte na stránku a zaregistrujte se do systému.
2. **Listing a filtrace produktů** Přejděte do listu zlatých mincí a vyfiltrujte produkty dražší než 1000kč a levnější než 2000 kč (cena prodejní cena).
3. **Detail produktu** Přejděte do detailu kteréhokoliv produktu a najděte následující:
   a. Prodejce s nejvyšší výkupní cenou
   b. Prodejce s nejnižší prodejní cenou
   c. Graf vývoje výkupní ceny v čase
4. **Detail přihlášeného uživatele** Najděte detail uživatele, za kterého jste přihlášen/na, a změňte jeho jméno a příjmení na své.
5. **Administrace uživatelů** Přejděte do administrace uživatelů a za pomoci filtrace najděte uživatele s emailem simon.klibi@gmail.com
6. **Smazání konkrétního uživatele** Uživatele z bodu výše vymažte ze systému.
7. **Změna hesla konkrétního uživatele v administraci uživatele** Změňte heslo libovolnému uživateli.
8. **Přesun linku pod jiný produkt** V detailu produktu vyberte libovolný link a zařaďte ho pod jiný produkt.

7.2 Průběh testování

Testování probíhalo na počítači Macbook Pro 16“. Před začátkem testu byly uživatelům vysvětleny velmi hrubé základy investování do drahých kovů, byl jim popsán problém, který aplikace řeší a byl jim vysvětlen proces přesunu linku pod jiný produkt.

Následně uživatelé dostali na výběr z prohlížečů Google Chrome, Safari a Mozilla Firefox a byl jim poskytnut scénář, kterým se během testování musí řídit. Všichni uživatelé si pro průchod aplikací vybrali prohlížeč Google Chrome, u posledního uživatele jsem udělal výjimku a test byl pro širší pokrytí prohlížečů proveden v prohlížeči Safari.

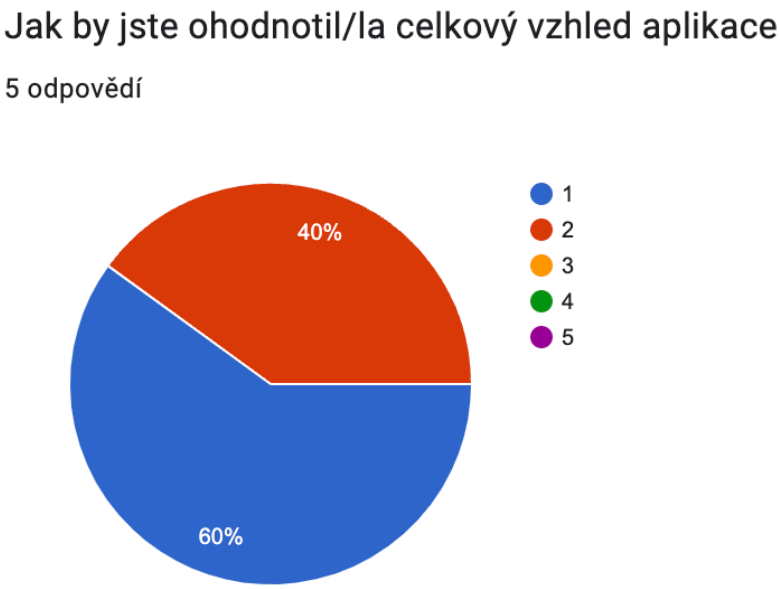
Po absolvování testování uživatelé dostali krátký dotazník, který vyplnili. Dotazník obsahoval otázky týkající se vzhledu aplikace a pocitu při užívání. V Další části pak hodnotili složitost jednotlivých kroků na stupnici 1–5 (školní známkování) a zdůvodňovali svoji volbu.

U otázky „Jak by jste ohodnotil/la celkový vzhled aplikace“ oznámkovali aplikaci 3 uživatelé známkou 1 a 2 uživatelé známkou dva, u otázky „Jaký byl váš celkový pocit při užívání aplikace“ pak všichni uživatelé oznámkovali aplikaci známkou 1.

Mezi největší nedostatky aplikace pak uživatelé řadili následující:

1. chybějící fotografie produktů,
2. nekonzistentost v jazyce použitém v celé aplikaci a v názvech produktů (angličtina x čeština),
3. chybějící lokalizace aplikace do českého jazyka,
4. v listingu produktů chybí řazení produktů (podle ceny, váhy…),
5. po kliknutí na logo aplikace není uživatel přesměrován na homepage,
6. proces přesunu linku pod jiný produkt je složitější,
7. součástí prodktů, na které je možno link přesunou je i produkt na kterém se link aktuálně nachází, po zvolení tohoto produktu a kliknutí na tlačítko „relink“ aplikace vypisuje chybu.

Naopak nejvíce oceňovaná byla intuitivnost aplikace, uživatelé v aplikaci vždy našli to, co hledali a jednotlivé scénáře zvládali takřka bez problému.



Obrázek 7.1 Graf odpovědí - vzhled aplikace

Obrázek 7.2 Graf odpovědí - pocit při užívání

7.3 Závěr

O chybějící fotografii produktu jsem věděl již před začátkem uživatelského testování. Tento problém bohužel nebylo možno v době implementace vyřešit, jelikož backend investičního scraperu neposkytuje při volání endpointu URL na fotografii produktu.

Nekonzistentnost v jazyce je způsobena názvy produktů, tyto produkty jsou scrapovány z českých e-shopů a jejich názvy určují jednotliví vlastníci webu. Jako možnost se nabízí překlad názvů do angličtiny. Překlad názvu mi však nepřipadá, z důvodu zachování správnosti údajů na webu, jako správná volba.

Aplikace je plně lokalizovaná do angličtiny, a při vývoji aplikace se nepočítalo s jejím překladem do českého jazyka, nicméně bych do budoucna v závislosti na zadefinování cílového zákazníka zvážil její lokalizaci do českého jazyka.

Chybějící řazení v listingu produktů by mělo být do budoucna implementováno. Aktuální stav backendu neumožňuje řazení na straně serveru, a implementace by tak musela být ponechána na frontendu.

Přesměrování na homepage po kliknutí na logo aplikace bylo v aplikaci po uživatelském testování doimplementováno.

Co se procesu přesunu linku pod jiný produkt týče, je jako takový komplexnější záležitostí a vyžaduje rozšířené znalosti uživatele o aplikaci. Tato funkce je nicméně v aplikaci dostupná pouze pro uživatele s rolí administrátor. Tomuto uživateli by byla v reálném provozu tato funkcionalita podrobně vysvětlena. Po vysvětlení by pak uživatel neměl mít problém tuto funkcionalitu používat.

Problém s chybnou možností přesunu linku pod produktu, ke kterému je link aktuálně přiřazen byl vyřešen odstraněním tohoto produktu ze seznamu možných voleb.

Závěr

Cílem práce byla realizace frontendu pro investiční scraper. Koncept porovnávání cen drahých kovů se mi zalíbil již od okamžiku, kdy jsem o něm slyšel poprvé, a jsem moc rád, že jsem se mohl stát jeho součástí.

Ve své práci jsem zprvu podle zadání zadefinoval funkční a nefunkční požadavky, díky kterým jsem dostal hrubý přehled o finálních funkcionalitách, které vznikající aplikace musí umět.

Dále jsem provedl analýzu stavu dosavadního (starého) frontendu a backendu, se kterým aplikace komunikuje. Frontend jsem analyzoval jak po stránce zvolených technologií, tak i po stránce samotné architektury. Během této analýzy jsem objevil mnoho potenciálních chyb, které by v budoucnu mohly představovat problém. Analyzované problémy jsem poté v nově vznikající aplikaci vyřešil.

V rámci rešerše technologií jsem zvažoval, jestli aplikaci implementovat jako SPA, nebo MPA a rozmyslel jsem si, jaké technologie budu při implementaci používat.

Následně jsem provedl návrh architektury, kde jsem se snažil vyřešit problémy dosavadního frontendu a vytvořit solidní návrh tak, aby byla aplikace do budoucna udržitelná a rozšířitelná ve všech směrech.

Po dokončení výše zmíněného, jsem začal s návrhem finální podoby aplikace. Nejprve jsem vytvořil wireframe ve formě jednoduchého náčrtu, ze kterého následně vznikl koneční UI design aplikace.

Při implementaci se mi následně podařilo zrealizovat jednotlivé body požadavků na aplikaci a vyřešit problémy, se kterými se dosavadní frontend potýkal. Tím jsem splnil hlavní cíl této práce.

Výslednou aplikaci jsem poté podrobil uživatelskému testování, které odhalilo drobné nedostatky, každý z nedostatků byl analyzován, a některé z nich pak opraveny.

Při zpracovávání závěrečné práce jsem si výrazně rozšířil obzory, co se technologie Next.js a server-side renderingu týče, dále jsem si uvědomil důležitost jednotlivých kroků vedoucích k samotné implementaci (funkční požadavky, rešerše etc.) díky kterým jsem se vyvaroval mnoha chybám a zbytečným opravám kódu.

7.3.1 Budoucí vývoj

V případě pokračovaní projektu bych obzvláště doporučil zaměřit se na přenesení zodpovědnosti filtrace na serverovou část. Po doimplementování namockovaných částí backendu bude dále potřeba přizpůsobit části aplikace, které tyto mockované endpointy používají, obzvláště se jedná o uživatelský profil, přihlášení, panel středových cen a přiřazování nezařazených produktů. U přiřazování nezařazených produktů a panelu spotů a středových cen endpoint na backendu zcela chybí.

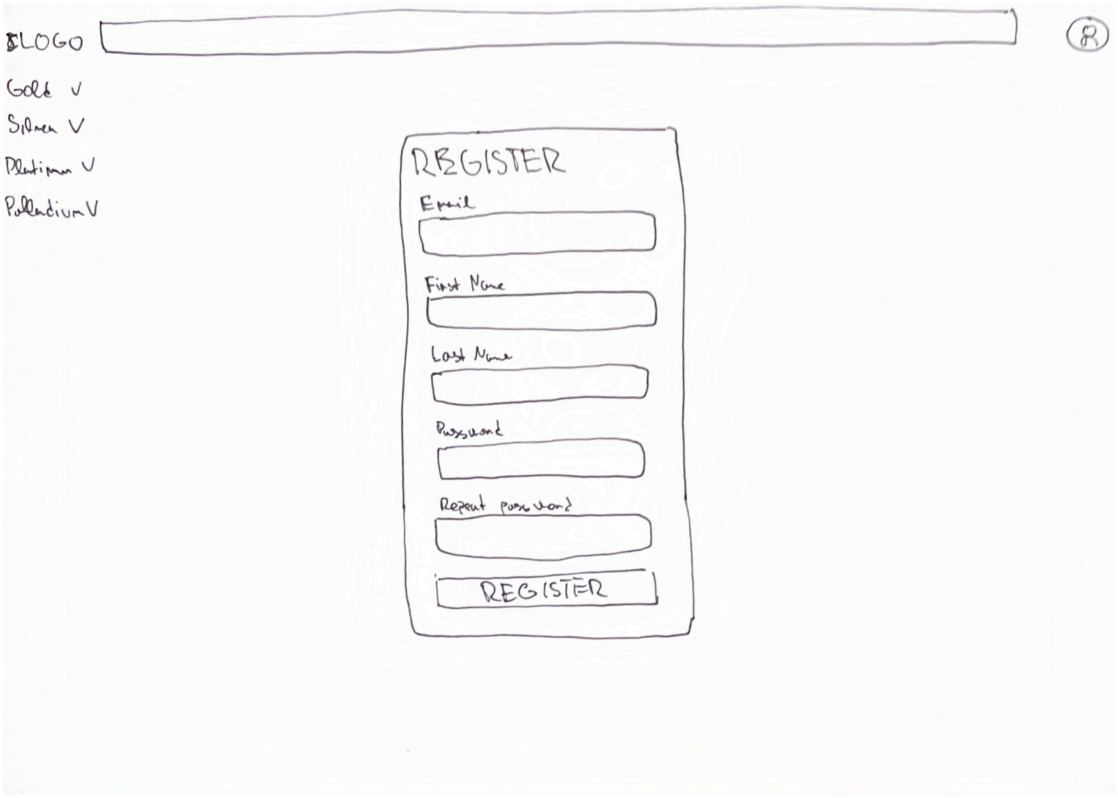
Dále bych se pak zaměřil na návrh homepage, která je doposud zcela prázdná. Na homepage by bylo podle mého názoru nejvhodnější navrhnout uživatelský dashboard, který by mohl obsahovat souhrny jednotlivých drahých kovů a produktů.

Aplikace byla navržena tak, aby měla po technické stránce co nejlepší základy pro dobré SEO ohodnocení. Samotné SEO (title tagy, meta tag descriptio…) však v aplikaci řešeno nebylo. Bylo by proto vhodné se na tuto část zaměřit a vyladit ji pro co nejlepší výsledky.

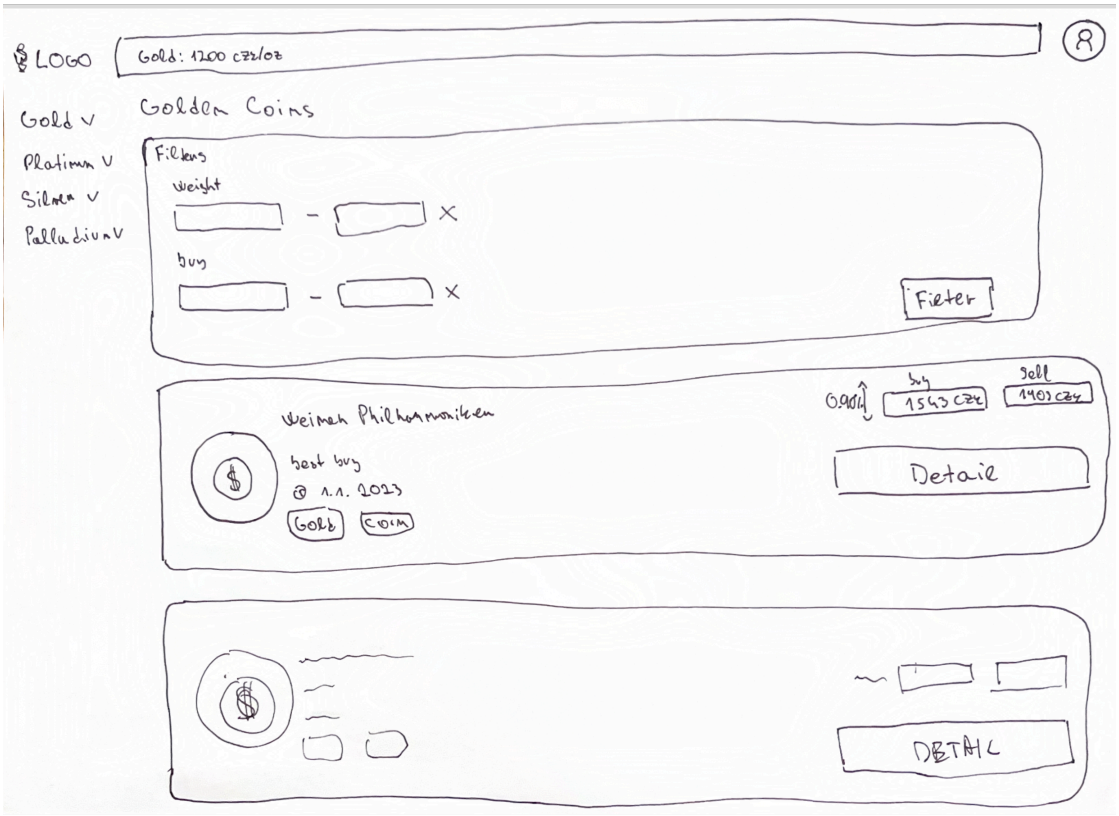
V poslední řádě bych pak zvážil překlady aplikace do různých jazyků v závislosti na cílovém zákazníkovi. K této lokalizaci bych využil její nativní podpory frameworkem Next.js více v [37].

Příloha A

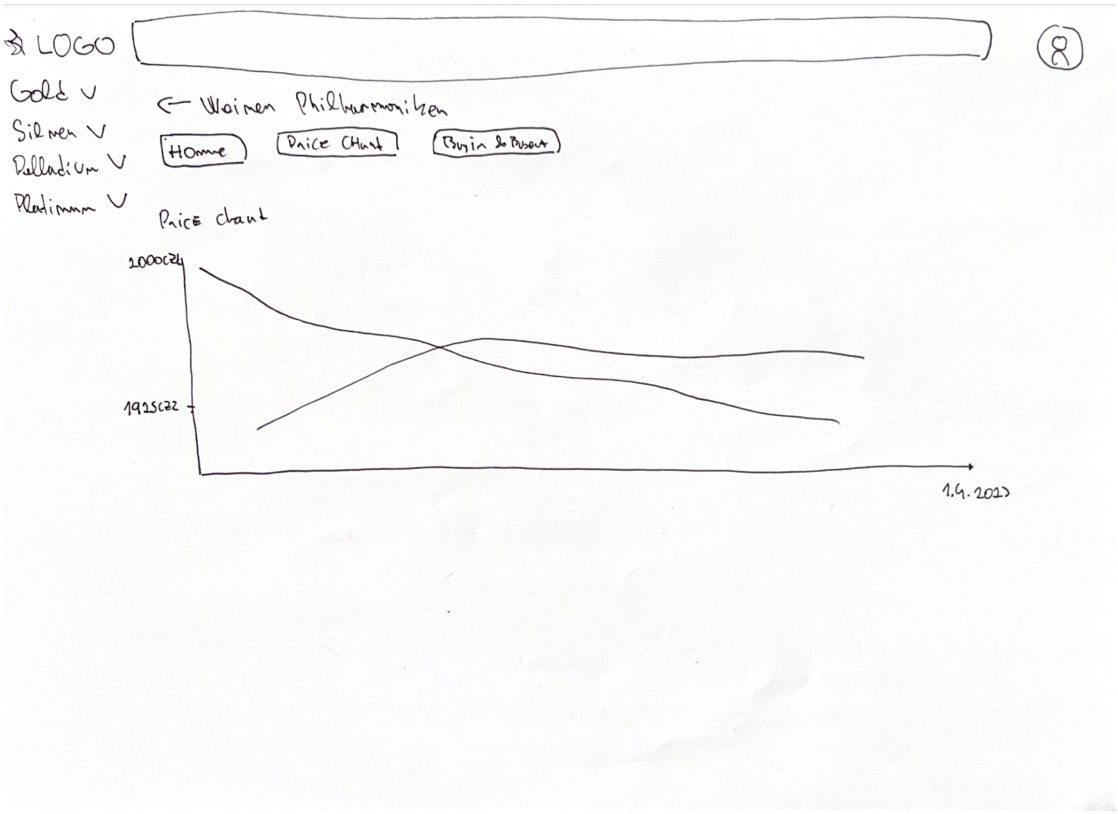
Wireframe aplikace



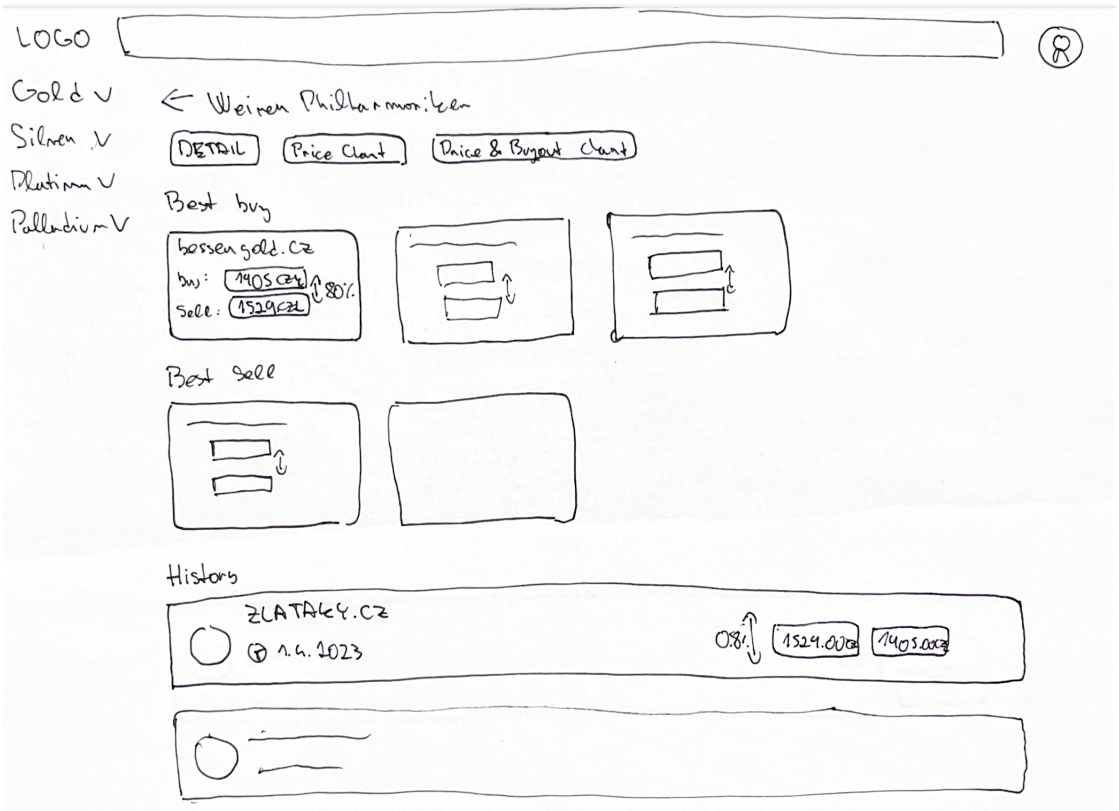
Obrázek A.1 Wireframe Registrace



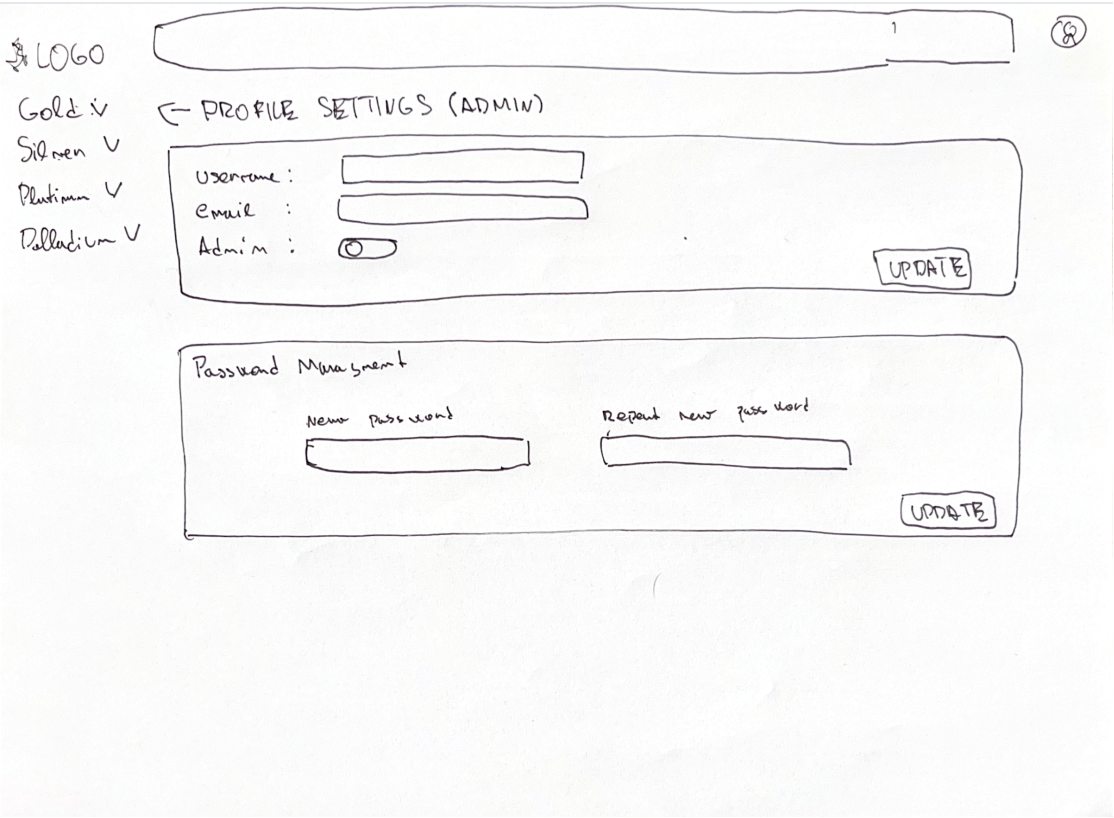
Obrázek A.2 Wireframe listingu produktů



Obrázek A.3 Wireframe grafu vývoje cen v historii v detailu produktu



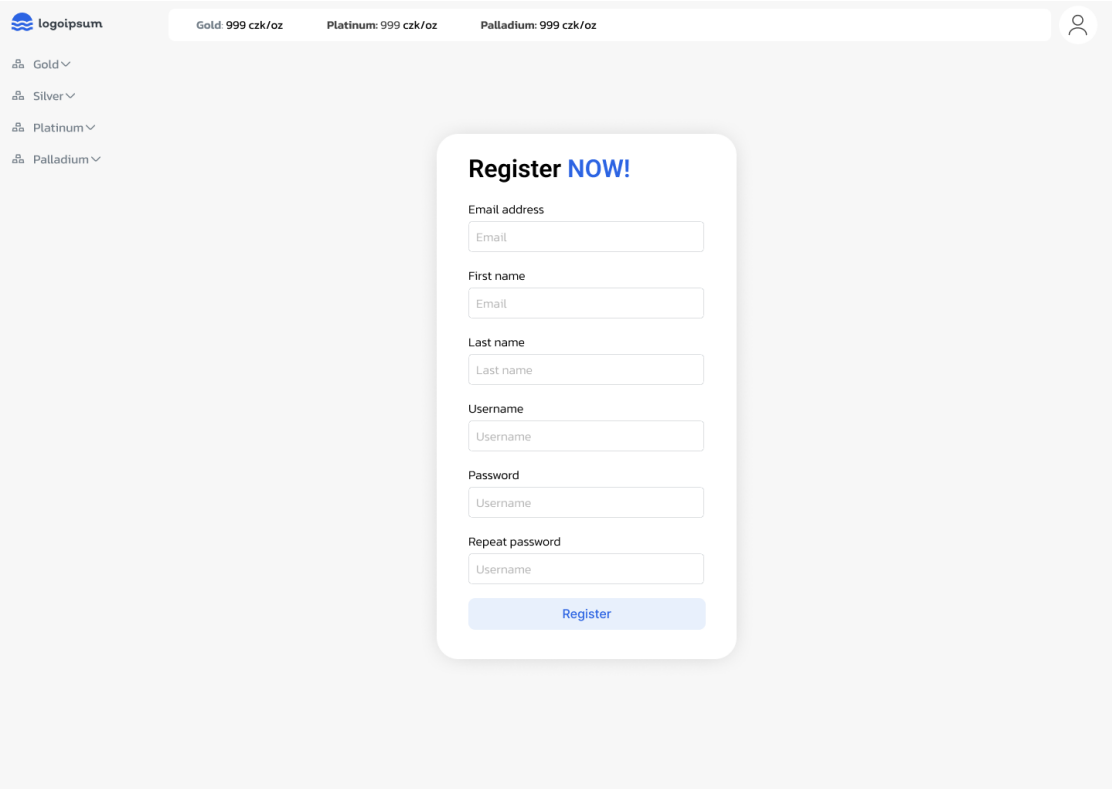
Obrázek A.4 Wireframe detailu produktu



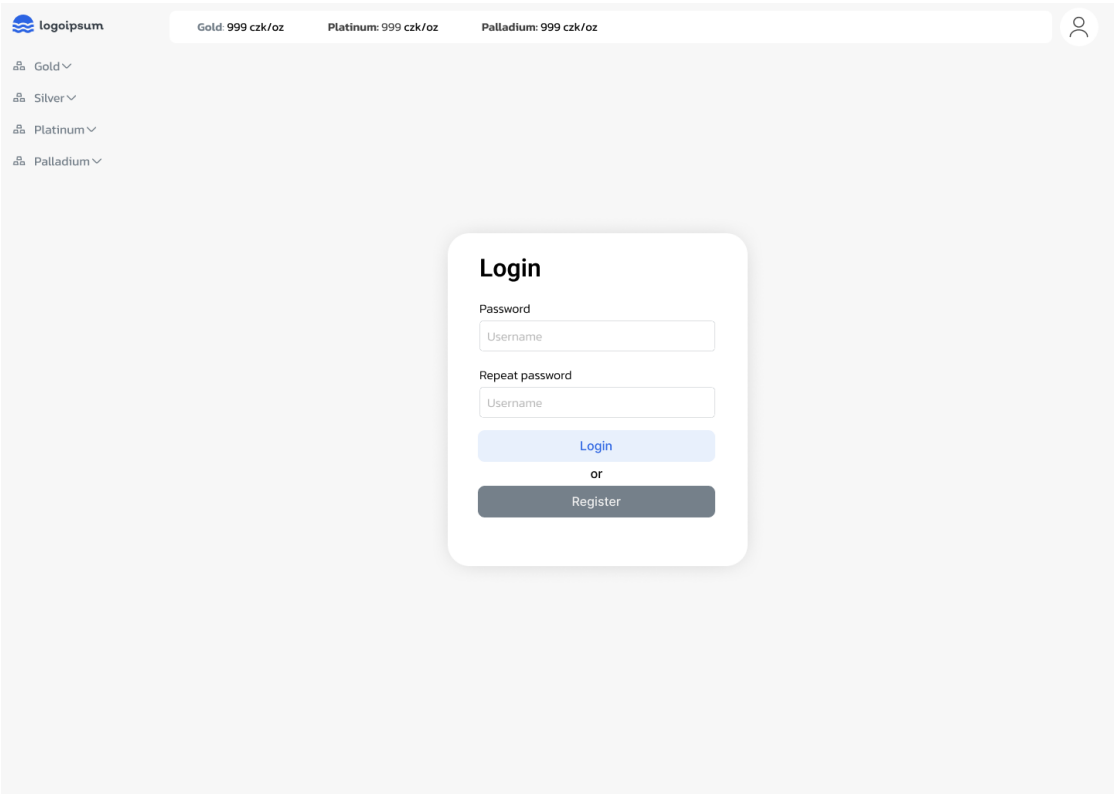
Obrázek A.5 Wireframe nastavení uživatele

Příloha B

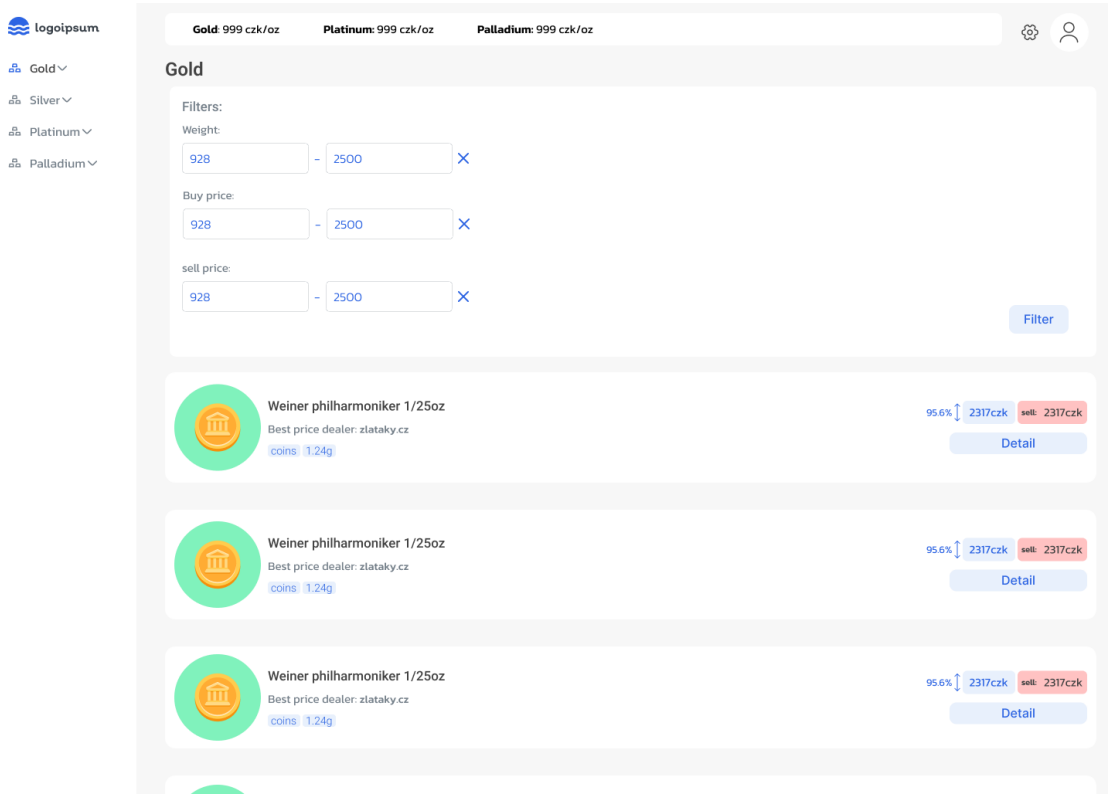
UI design aplikace



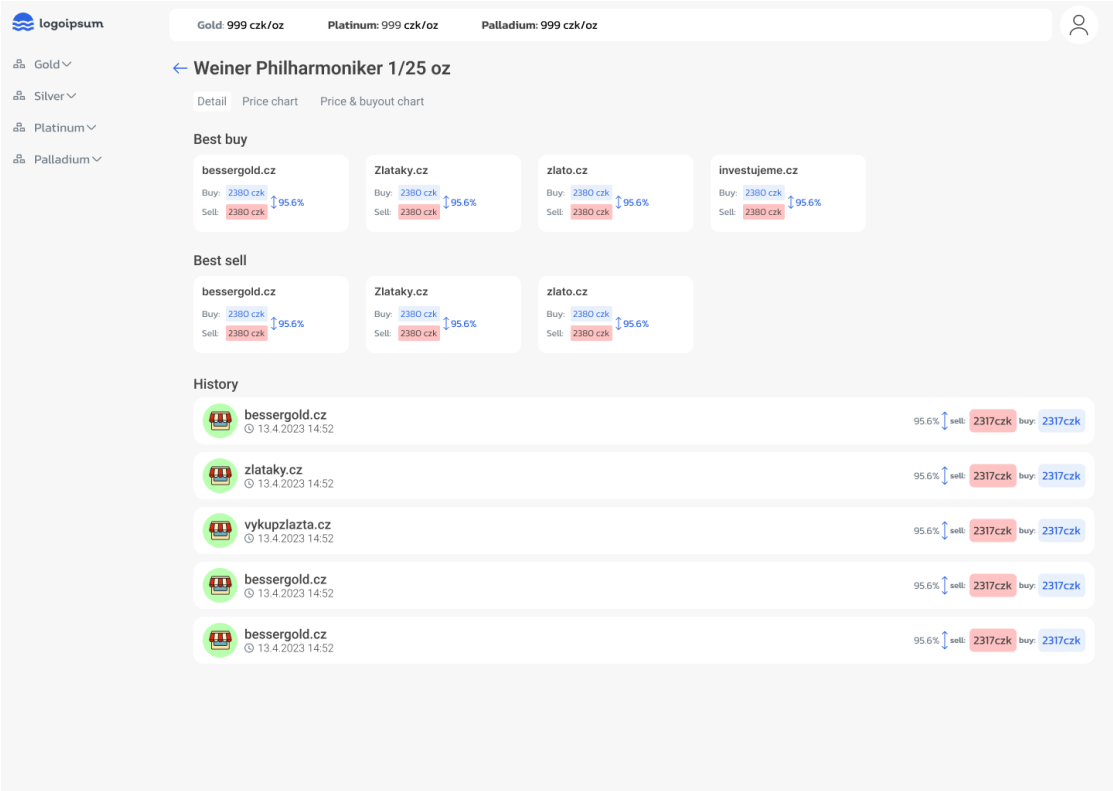
Obrázek B.1 UI design registrace



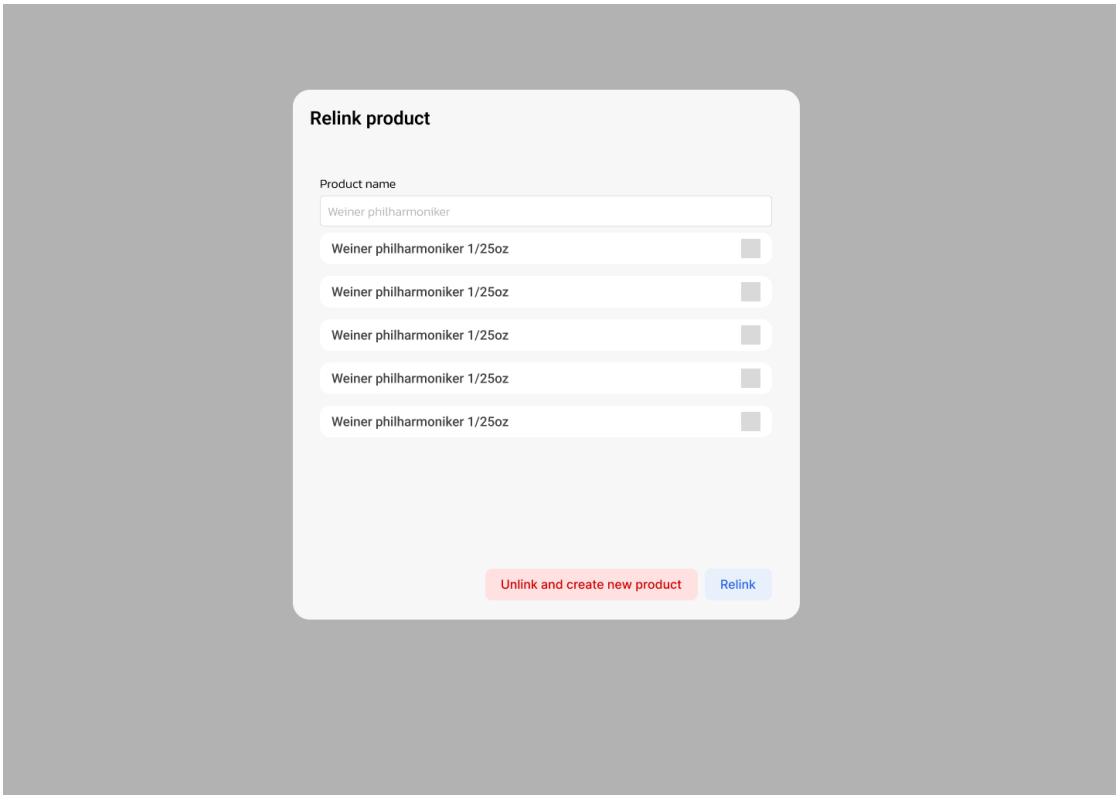
Obrázek B.2 UI design login



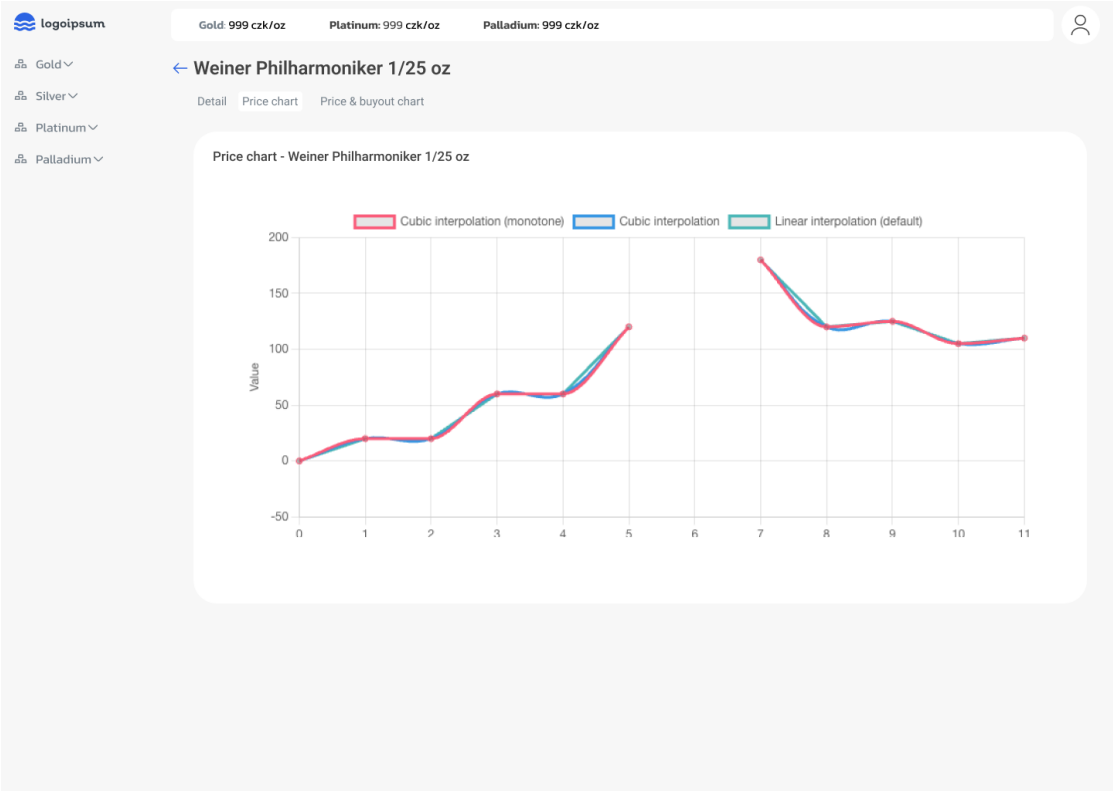
Obrázek B.3 UI design listingu produktů



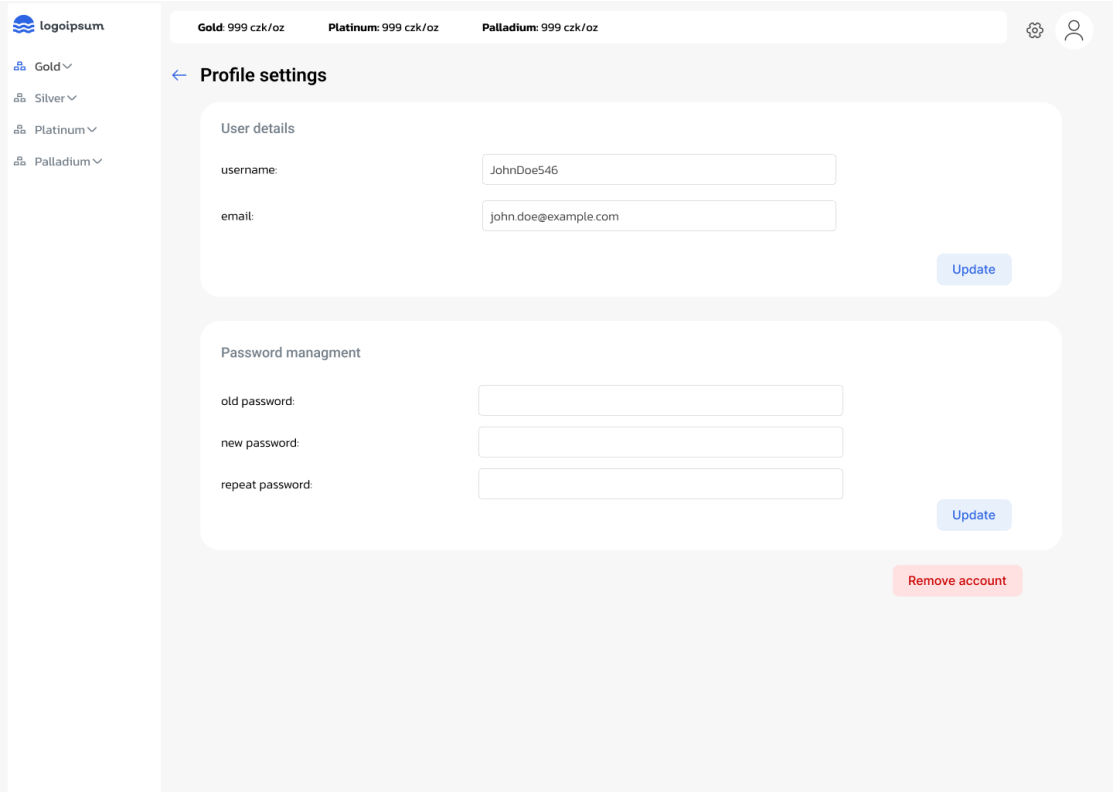
Obrázek B.4 UI design detailu produktu



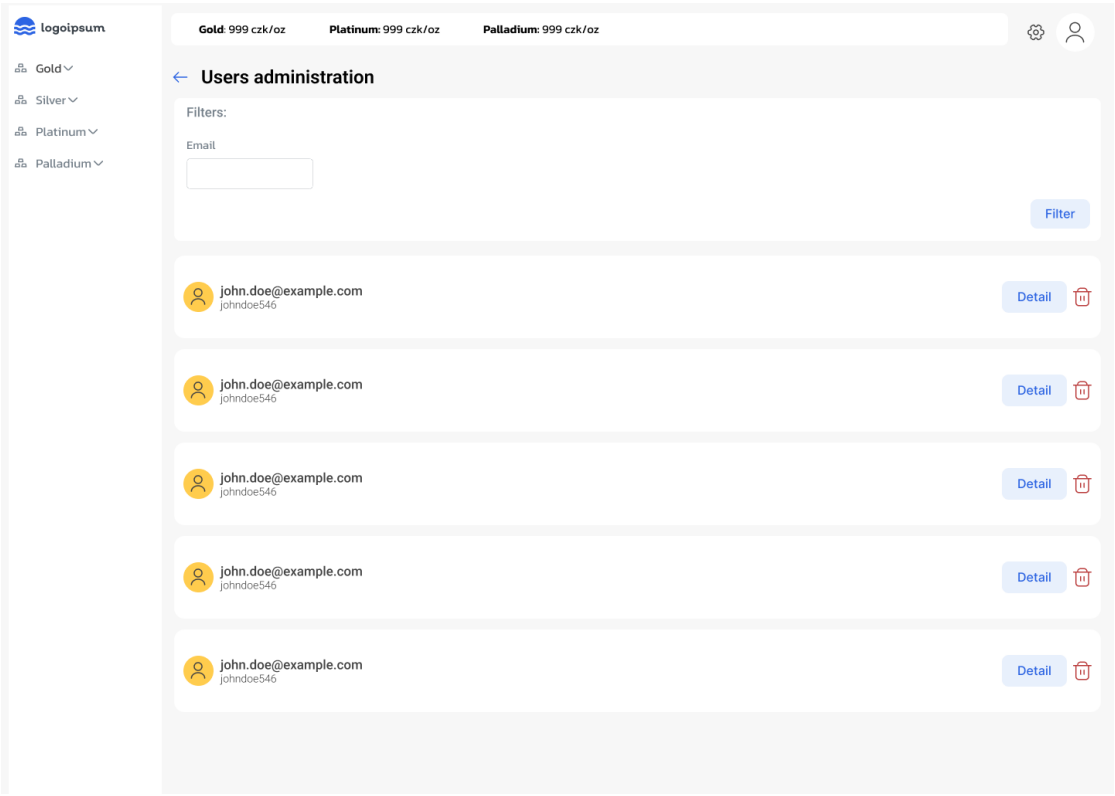
Obrázek B.5 Ui design modal pro přesunutí odkazu



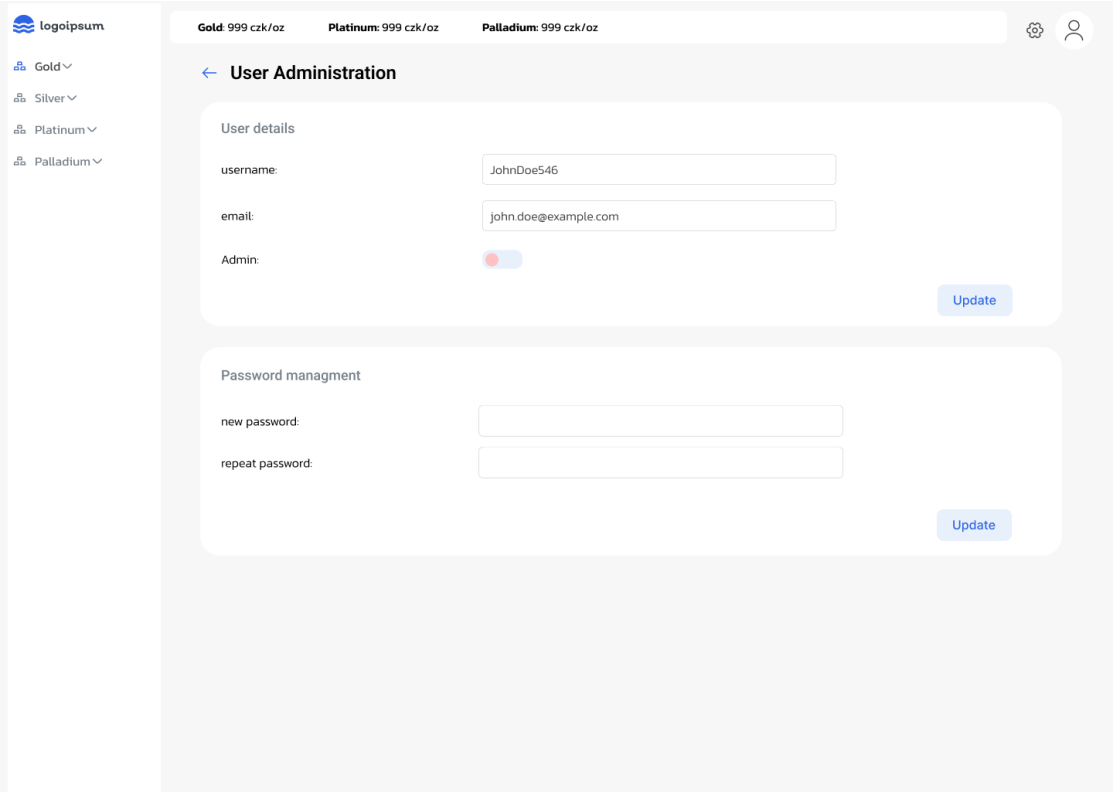
Obrázek B.6 UI design vývoje cen v historii v detailu produktu



Obrázek B.7 UI design nastavení uživatele



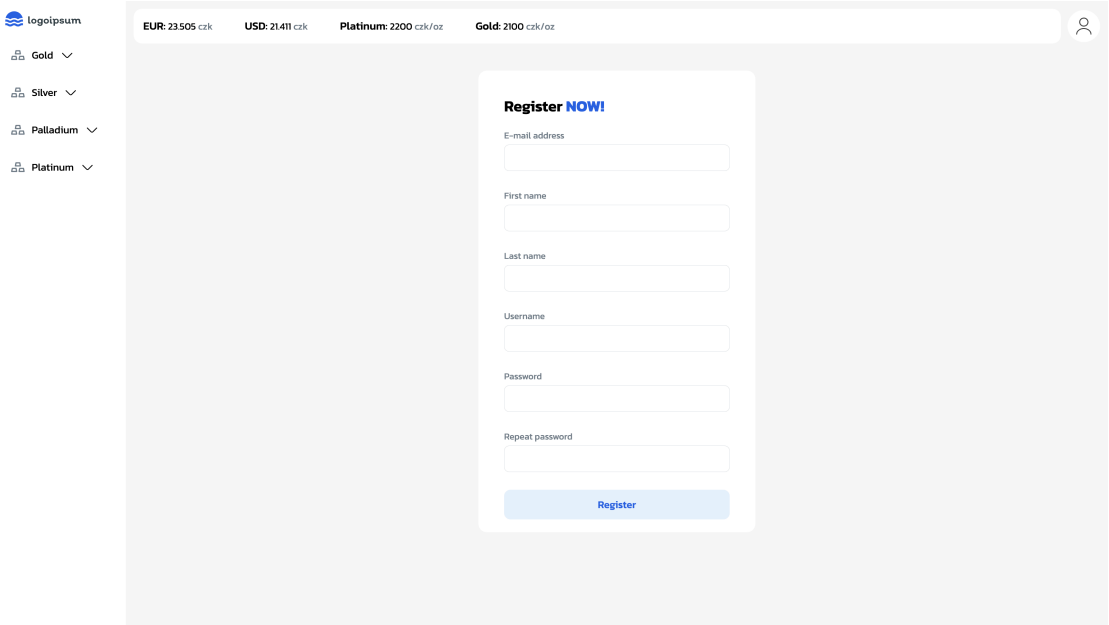
Obrázek B.8 UI design administrátorský list uživatelů



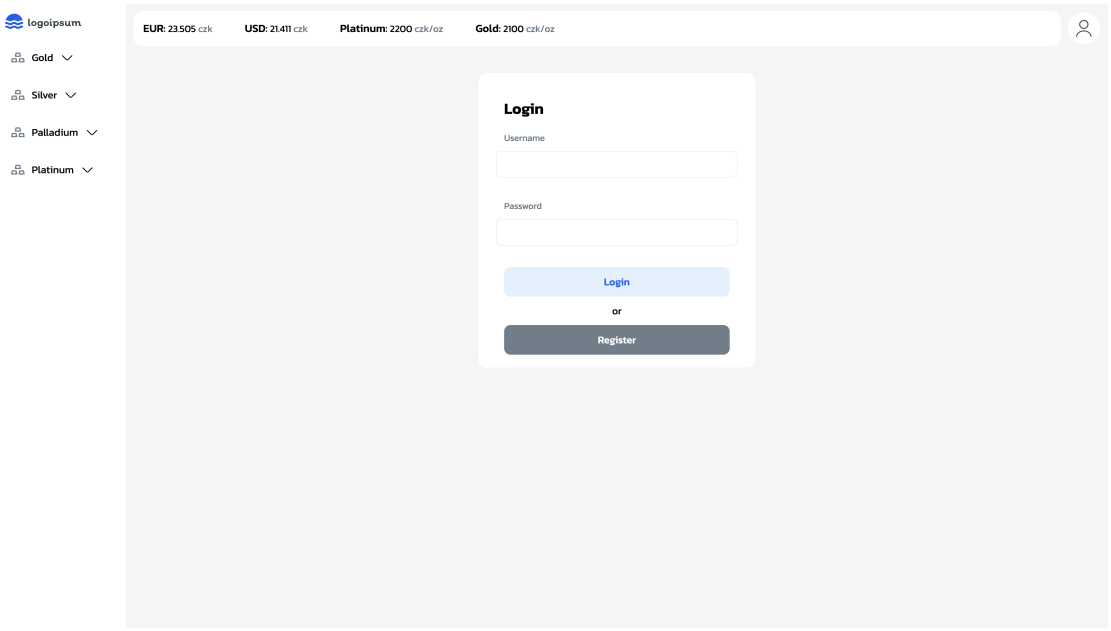
Obrázek B.9 UI design administrace uživatele

Příloha C

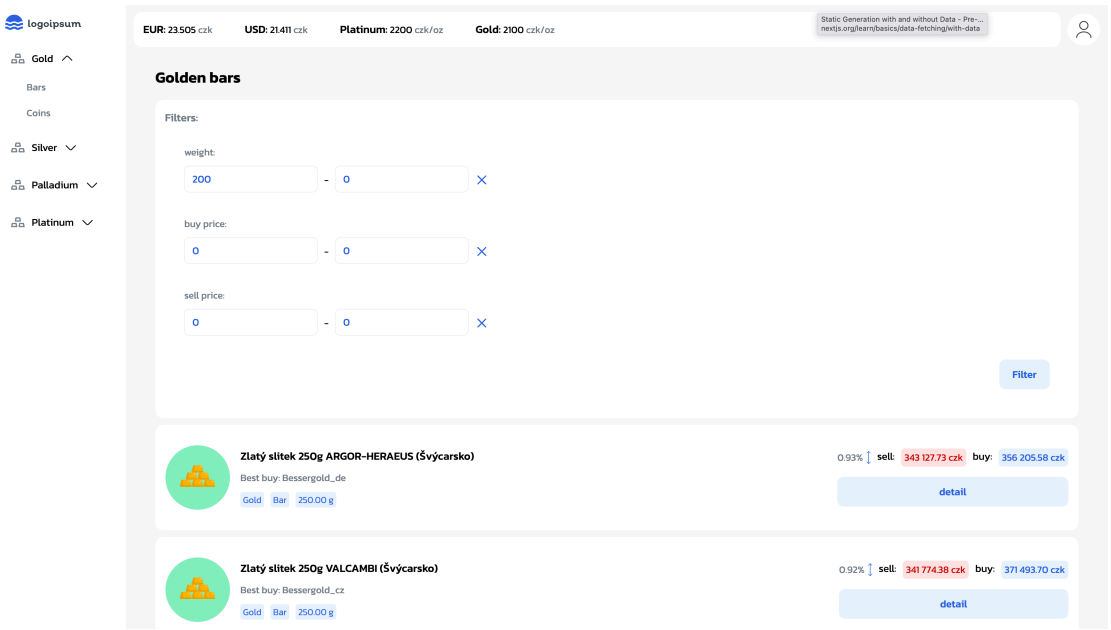
Výsledný vzhled aplikace



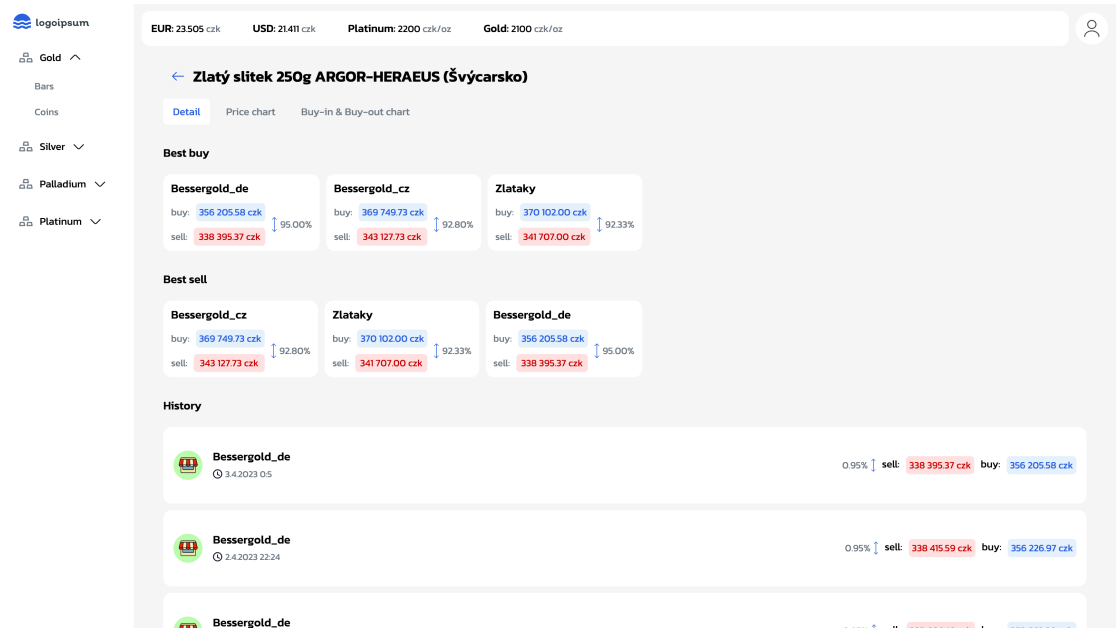
Obrázek C.1 Výsledný vzhled registrace



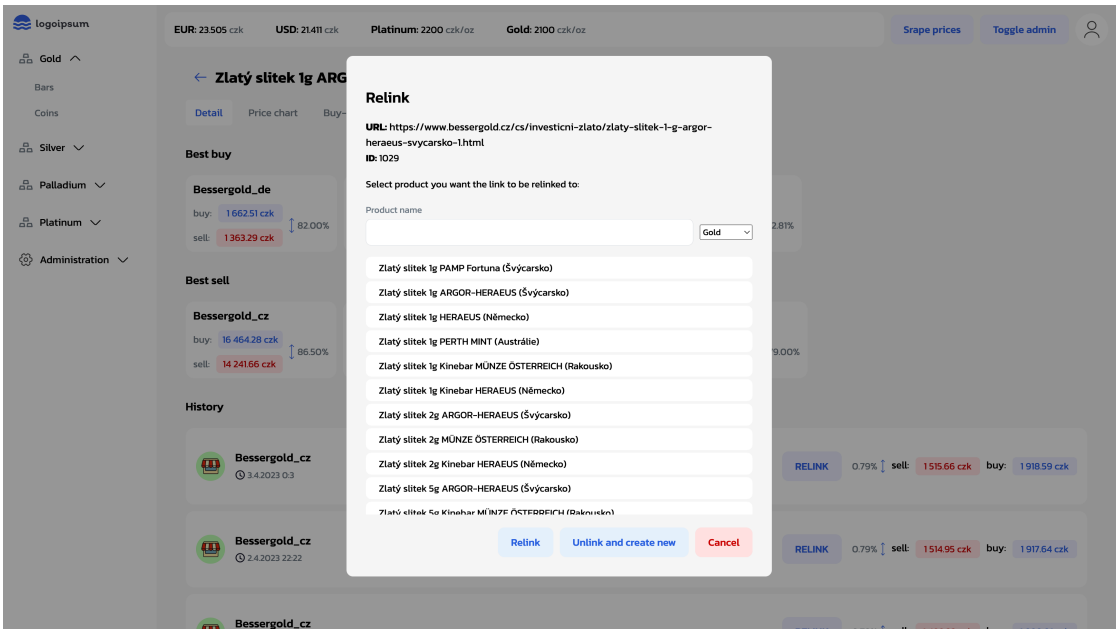
Obrázek C.2 Výsledný vzhled login stránky



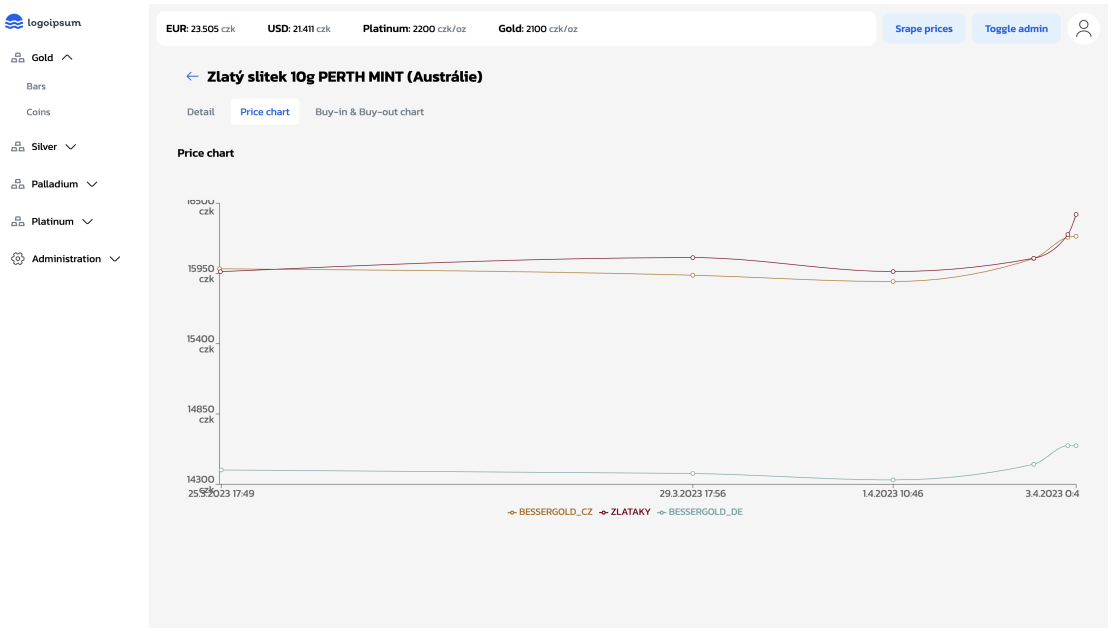
Obrázek C.3 Výsledný vzhled listingu produktů



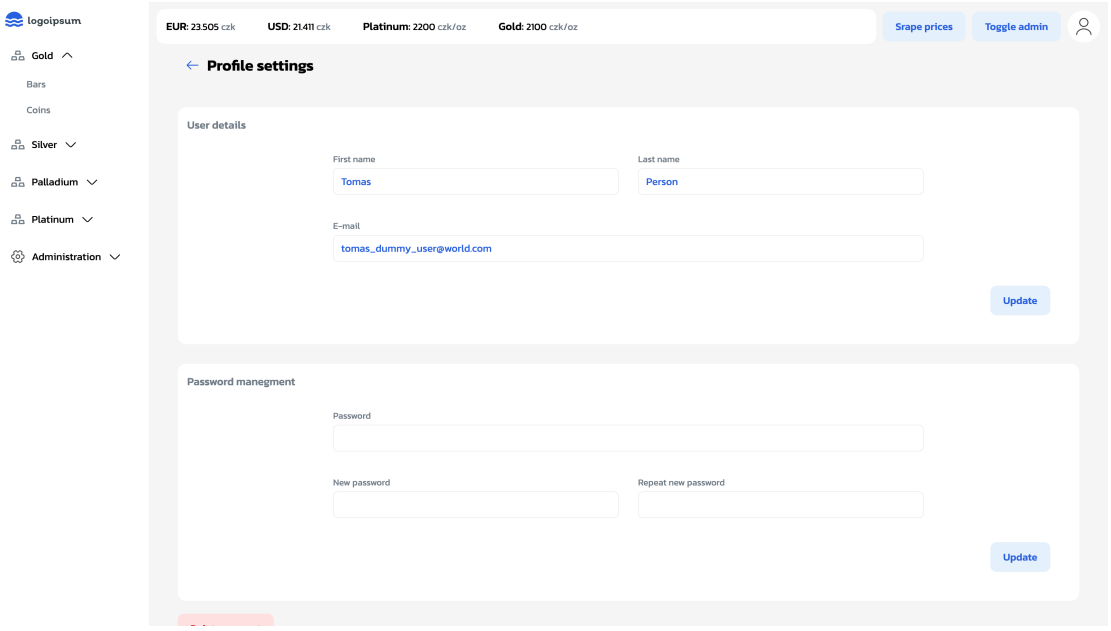
Obrázek C.4 Výsledný vzhled detailu produktu



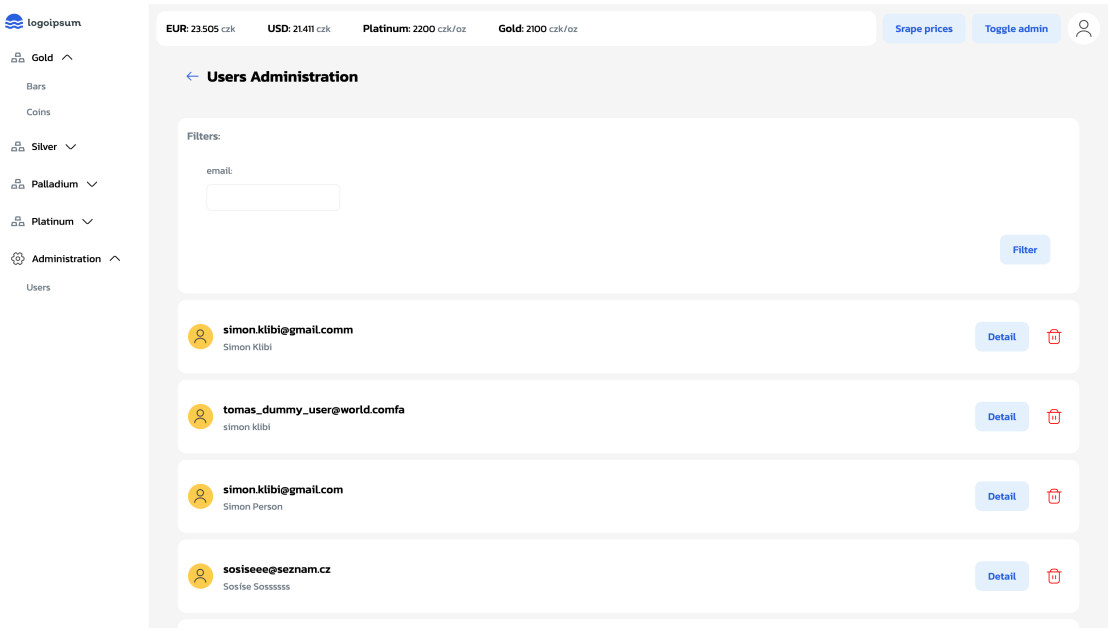
Obrázek C.5 Výsledný vzhled modal pro přesunutí odkazu



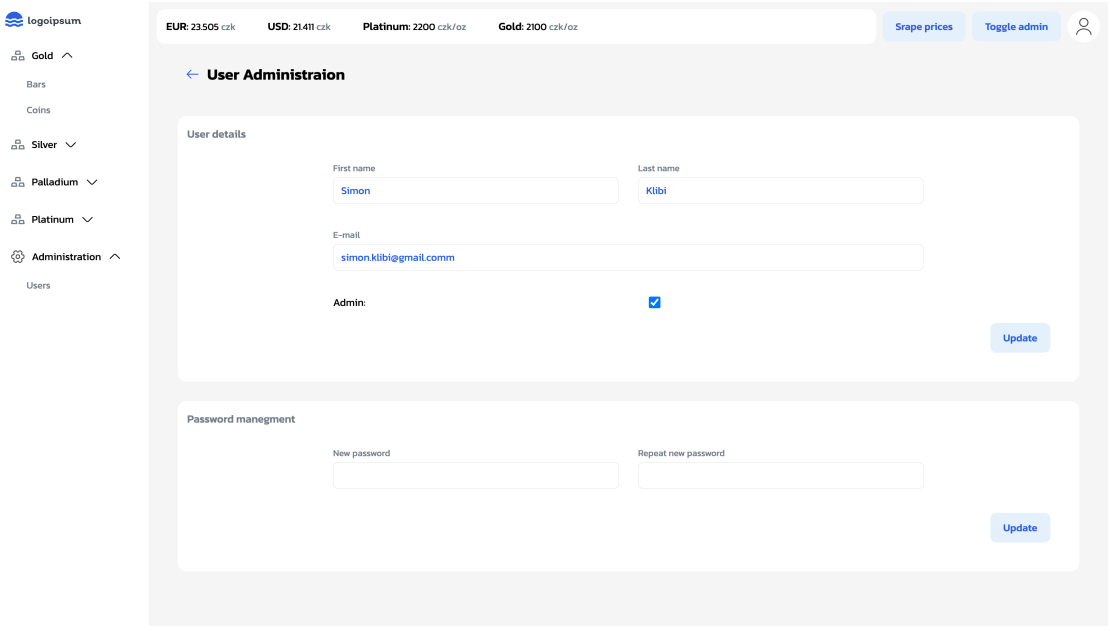
Obrázek C.6 Výsledný vzhled vývoje cen v historii v detailu produktu



Obrázek C.7 Výsledný vzhled nastavení uživatele



Obrázek C.8 Výsledný vzhled administrátorského listu uživatelů



Obrázek C.9 Výsledný vzhled administrace uživatele

Bibliografie

1. MALAN, Ruth; BREDEMEYER, Dana et al. Functional requirements and use cases. *Bredemeyer Consulting*. 2001.
2. KOTONYA, Gerald; SOMMERVILLE, Ian. *Requirements Engineering*. Nashville, TN: John Wiley & Sons, 1998. Worldwide Series in Computer Science. ISBN 9780471972082.
3. *MUI: The React component library you always wanted* [online]. [B.r.]. [cit. 2023-03-02]. Dostupné z: `https://mui.com/`.
4. *Material Design* [online]. [B.r.]. [cit. 2015-03-02]. Dostupné z: `https://m2.material.io/`.
5. VETTER-NEO, Natalia. *When To Use A Ui Component Library In A React Project? | Sunscrapers* [online]. 2020. [cit. 2023-04-27]. Dostupné z: `https://sunscrapers.com/blog/ui-component-library-in-a-react-project/`.
6. BEN-YAIR, Nir. *Headless components in React and why I stopped using a UI library for our design system* [online]. 2022. [cit. 2023-04-27]. Dostupné z: `https://medium.com/@nirbenyair/headless-components-in-react-and-why-i-stopped-using-ui-libraries-a8208197c268`.
7. *Separation of concerns with React hooks | Felix Gerschau* [online]. 2021. [cit. 2023-04-27]. Dostupné z: `https://felixgerschau.com/react-hooks-separation-of-concerns/`.
8. *REST API Documentation Tool | Swagger UI* [online]. [B.r.]. [cit. 2023-03-20]. Dostupné z: `https://swagger.io/tools/swagger-ui/`.
9. *Mockoon - Mockoon's guide to API mocking* [online]. [B.r.]. [cit. 2023-03-20]. Dostupné z: `https://mockoon.com/articles/what-is-api-mocking/`.
10. SCOTT JR, Emmit A. *SPA Design and Architecture: Understanding single-page web applications*. Simon a Schuster, 2015. ISBN 9781617292439.
11. SKÓLSKI, Paweł. *Single page vs. multi page applications - differences* [online]. 2016. [cit. 2023-03-12]. Dostupné z: `https://neoteric.eu/blog/single-page-application-vs-multiple-page-application/`.
12. GÓRALEWICZ, Bartosz. *Single page vs. multi page applications - differences* [online]. 2017. [cit. 2023-03-12]. Dostupné z: `https://moz.com/blog/search-engines-ready-for-javascript-crawling`.
13. ŠTRÁFELDA, Jan. *Co je crawling* [online]. [B.r.]. [cit. 2023-04-27]. Dostupné z: `https://www.strafelda.cz/crawling`.
14. TEAM, Altamira expert. *Single page application vs Multi page application* [online]. 2022. [cit. 2023-05-03]. Dostupné z: `https://www.altamira.ai/blog/single-page-app-vs-multi-page-app/`.

15. *Stack Overflow Developer Survey 2022* [online]. [B.r.]. [cit. 2023-03-12]. Dostupné z: https://survey.stackoverflow.co/2022/%5C#most-popular-technologies-webframe.
16. JOSHI, Mohit. *Angular vs react vs Vue: Core differences* [online]. 2022. [cit. 2023-03-07]. Dostupné z: https://www.browserstack.com/guide/angular-vs-react-vs-vue.
17. *Angular* [online]. [B.r.]. [cit. 2023-03-07]. Dostupné z: https://angular.io/guide/what-is-angular.
18. *Angular universal* [online]. 2022. [cit. 2023-04-29]. Dostupné z: https://angular.io/guide/universal.
19. *React Native · Learn once, write anywhere* [online]. [B.r.]. [cit. 2023-03-10]. Dostupné z: https://reactnative.dev/.
20. PATADIYA, Jaydeep. *React vs React Native - Key Difference, Features, Advantages* [online]. 2022. [cit. 2023-03-12]. Dostupné z: https://radixweb.com/blog/react-vs-react-native.
21. ARKHIPOV, Artem. *Benefits of Vue.js for Web Development* [online]. 2022. [cit. 2023-04-29]. Dostupné z: https://www.techmagic.co/blog/benefits-ofvuejs/.
22. *Next.js by Vercel - The React Framework for the Web* [online]. [B.r.]. [cit. 2023-03-12]. Dostupné z: https://nextjs.org/.
23. DEOSTHALE, Atharva. *Remix: A guide to the React framework taking on Next.js* [online]. 2022. [cit. 2023-04-29]. Dostupné z: https://blog.logrocket.com/guide-to-remix-react-framework/.
24. KONSHIN, Kirill. *Next. js Quick Start Guide: Server-side rendering done right*. Packt Publishing Ltd, 2018. ISBN 9781788993661.
25. MONUS, Anna. *What Is Server-side Rendering And How Does It Improve Site Speed?* [online]. 2022. [cit. 2023-03-14]. Dostupné z: https://www.debugbear.com/blog/server-side-rendering.
26. MASSE, Mark. *REST API design rulebook: designing consistent RESTful web service interfaces*. "O'Reilly Media, Inc.", 2011. ISBN 9781449310509.
27. *TanStack Query | React Query, Solid Query, Svelte Query, Vue Query* [online]. [B.r.]. [cit. 2023-03-18]. Dostupné z: https://tanstack.com/query/v3/.
28. *Sass: Syntactically Awesome Style Sheets* [online]. [B.r.]. [cit. 2023-03-19]. Dostupné z: https://sass-lang.com/.
29. QUEIRÓS, Ricardo. A survey on CSS preprocessors. 2017.
30. NIELSEN, Jakob. Enhancing the explanatory power of usability heuristics. In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. 1994, s. 152–158.
31. HANNAH, Jaye. *What Is A Wireframe? Your Best Guide — careerfoundry.com* [online]. [B.r.]. [cit. 2023-03-19]. Dostupné z: https://careerfoundry.com/en/blog/ux-design/what-is-a-wireframe-guide/.
32. *2022 Design Tools Survey - UI Design* [online]. [B.r.]. [cit. 2023-03-20]. Dostupné z: https://uxtools.co/survey/2022/ui-design/.
33. *One app, all things money | Revolut* [online]. [B.r.]. [cit. 2023-04-27]. Dostupné z: https://www.revolut.com/.
34. HAMDI, Arian. *React Query v4 + SSR in Next JS* [online]. 2022. [cit. 2023-04-29]. Dostupné z: https://dev.to/arianhamdi/react-query-v4-ssr-in-next-js-2ojj.
35. *Array.prototype.filter() - JavaScript | MDN* [online]. [B.r.]. [cit. 2023-03-20]. Dostupné z: https://mockoon.com/articles/what-is-api-mocking/.

36. TAN, Wei-siong; LIU, Dahai; BISHU, Ram. Web evaluation: Heuristic evaluation vs. user testing. *International Journal of Industrial Ergonomics*. 2009, roč. 39, č. 4, s. 621–627.

37. *Advanced Features: Internationalized Routing* [online]. [B.r.]. [cit. 2023-05-04]. Dostupné z: `https://nextjs.org/docs/advanced-features/i18n-routing`.

Obsah přílohy

- `README.md` ..... instrukce ke spuštění aplikace
- `userManual.pdf` ..... uživatelská příručka k aplikaci
- `src`
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `app` ..... zdrojový kód aplikace
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF