Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Vstupní stránky pro osoby a instituce s pokročilým vyhledáváním

Bakalářská práce

Autor práce: Lukáš Hornych
Studijní obor: Aplikovaná Informatika

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Hradec Králové 2022

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 22. dubna 2022 Lukáš Hornych

Poděkování

Tímto bych chtěl poděkovat vedoucímu mé práce prof. RNDr. PhDr. Antonínu Slabému CSc. za odborné vedení práce, poskytnuté rady a přínosné konzultace, také mým přátelům za jejich pomoc a podporu.

Anotace

Bakalářská práce se zabývá návrhem a implementací webové aplikace fungující na všech standardních zařízeních, která bude sloužit ke vzájemnému sdílení a vyhledávání kontaktních údajů, sociálních sítí, firemních struktur a dalších informací mezi osobami, firmami, neziskovými organizacemi, umělci a jinými subjekty či objekty na jednom místě. Hlavním požadavkem je umožnit všem snadný a rychlý přístup ke všem informacím bez omezení, jako je registrace nebo geografická lokace jedince. Součástí je analýza podobných webových aplikací a dostupných webových technologií a nástrojů pro vývoj webových aplikací od nuly až po běh v produkčním prostředí. Práce následně vybírá nejvhodnější technologie a nástroje pro samotnou implementaci moderní webové aplikace, která umožňuje vytváření a prohlížení vstupních stránek a přibližuje se svým chováním nativním desktopovým a mobilním aplikacím.

Annotation

Title: Landing pages for individuals and institutions with advanced search

Bakalářská práce se zabývá návrhem a implementací webové aplikace fungující na všech standardních zařízeních, která bude sloužit ke vzájemnému sdílení a vyhledávání kontaktních údajů, sociálních sítí, firemních struktur a dalších informací mezi osobami, firmami, neziskovými organizacemi, umělci a jinými subjekty či objekty na jednom místě. Hlavním požadavkem je umožnit všem snadný a rychlý přístup ke všem informacím bez omezení, jako je registrace nebo geografická lokace jedince. Součástí je analýza podobných webových aplikací a dostupných webových technologií a nástrojů pro vývoj webových aplikací od nuly až po běh v produkčním prostředí. Práce následně vybírá nejvhodnější technologie a nástroje pro samotnou implementaci moderní webové aplikace, která umožňuje vytváření a prohlížení vstupních stránek a přibližuje se svým chováním nativním desktopovým a mobilním aplikacím.

Obsah

1 Úvod

Tato bakalářská práce se bude zabývat vývojem aplikace pro tvorbu a zobrazování vstupních stránek osob, firem a jiných institucí. Takových aplikací existuje mnoho, nicméně žádná nekombinuje širší možnosti sdílení různých druhů informací a vyhledávání. Proto jsem se rozhodl navrhnout a implementovat vlastní verzi v podobě responzivní webové aplikace, která tyto požadavky bude splňovat.

Jednostránkové vstupní stránky jsou moderní cestou, jak sdílet všechny své veřejné kontaktní a jiné údaje se svými zákazníky. Představují jakousi digitální rozšířenou alternativu k běžným fyzickým vizitkám, které postrádají interaktivnost a spoustu důležitých údajů.

Dále se práce zaměří na analýzu implementačních technologií, hlavně pak na programovací jazyky a frameworky urychlující vývoj. Technologií v dnešní době existuje celá řada a každým dnem přibývají nové. Pro účely této práce se technologie rozdělí na front-endové a back-endové. Front-endové budou pomáhat tvořit responzivní a interaktivní uživatelské prostředí pro koncové uživatele. Back-endové, naproti tomu, budou tvořit konkrétní logiku aplikace. To zahrnuje převážně manipulaci s uživatelskými daty a jejich agregaci. Součástí této práce bude průzkum celosvětově využívaných technologií pro vývoj webových aplikací. V případě front-endu se bude jednat o nějaký z Javascriptových frameworků: Vue, React, Swelte nebo Angular. V rámci back-endu se práce zaměří na programovací jazyky Java, Javascript, C# a PHP, a také na podpůrné nástroje v podobě databázových systémů nebo souborových úložišť.

Po zvolení vhodných technologií bude následovat navržení uživatelského prostředí společně s interním datovým modelem a strukturou aplikace. Hlavním cílem bude tento návrh implementovat a jednotlivé řešené problémy popsat.

Konečným krokem bude spuštění aplikace v produkčním prostředí, která bude přístupná veřejnosti.

2 Cíl práce

V první řadě bude nezbytné zjistit, jaké informace bude vhodné umožňovat uživatelům zobrazovat na jejich vstupních stránkách, a jak s těmito informacemi pracovat v aplikaci. Dále bude následovat analýza a porovnání programovacích jazyků a frameworků, společně s jejich výhodami a nevýhodami. Výsledný výběr bude kombinací požadavků aplikace, výhod jednotlivých technologií a osobních preferencí. Avšak osobní preference budou hrát roli pouze v případě, kdy nebude možné rozhodnout čistě podle výhod nebo požadavků.

Implementace pak bude vycházet z návrhu datového modelu a struktury aplikace. Jednotlivé řešené problémy budou představeny společně s jejich možnými řešeními. Následovat bude popis výsledné implementace a důvod zvolení daného řešení.

Primárním cílem je vytvořit funkční interaktivní webovou aplikaci umožňující intuitivně svým uživatelům tvořit, spravovat a sdílet své vstupní stránky s jinými uživateli. Aplikace by měla také podporovat všechny využívané rozměry zařízení, aby byl zaručen přístup k aplikaci z mobilních i desktopových zařízení s minimálním omezením pro uživatele. Rovněž by aplikace měla být jednoduše spustitelná a udržitelně provozovatelná v produkčním prostředí.

Finálním cílem bude hotovu aplikace spustit v produkčním prostředí a stanovené požadavky otestovat.

3 Vstupní stránky pro osoby a instituce

Vstupní stránky pro osoby a instituce jsou prezentační jednostránkové webové stránky. Tyto stránky slouží k představení daného subjektu/objektu koncovému uživateli, který o něm hledá základní informace. Kromě základních informací, stránky většinou obsahují všemožné kontaktní údaje a odkazy na sociální sítě pro získání mnohem detailnějších informací. Některé stránky umožňují reprezentovat i například firemní struktury nebo geografické lokace. Dostupné druhy informací, které lze definovat na takových stránkách, ale nejsou jasně definované, spíše jsou určeny konkrétní webovou aplikací poskytující tyto služby.

Tento typ stránek je velice vhodný pro sdílení svých osobních údajů, a funguje jako takový rejstřík uživatelových údajů. Tyto stránky by se též daly popsat jako digitální rozšířené vizitky, nejen pro firmy.

Pro tvorbu takovýchto stránek existuje nepřeberné množství webových aplikací, kde lze tyto prezentace jednoduše tvořit a rovnou publikovat. Ne všechny ale nabízí všechny zmíněné možnosti, a je tak na uživatelovi, jakou webovou aplikaci zvolí.

Webové aplikace jsou aplikace, které běží ve webovém prohlížeči koncového uživatele bez nutnosti jakékoliv instalace. Tyto aplikace svá data ukládají do vzdálených serverů, a uživatel tak může ke svým datům jednoduše přistupovat z jakéhokoliv zařízení během okamžiku.

4 Existující webové aplikace

Existuje spousta webových aplikací, které by se dali považovat za řešení popsané problematiky. Nicméně žádná momentálně nekombinuje prvky sdílení kontaktních údajů, sociálních sítí a firemních struktur s plně fulltextovým vyhledáváním bez nutnosti registrace nebo rušivých sociálních prvků. Některé navíc neřeší ani ukládání cízích stránek do oblíbených s možností kategorizace či poznámek. U většiny takových aplikací se totiž spoléhá na navigaci pomocí přímých odkazů, které si daný subjekt vloží na své sociální sítě nebo vizitky.

Mezi současně nejznámější existující webové aplikace řešící podobné problémy patří například následující.

4.1 Linktree

Pravděpodobně nejrozšířenější takovou aplikací v zahraničí je Linktree. Ta se zabývá především zobrazováním odkazů na sociální sítě a rozsáhlým přizpůsobením. Momentálně však neumožňuje fulltextově vyhledávat jednotlivé stránky nebo je ukládat do oblíbených. Není ani jistota, že jednotlivé odkazy směřují opravdu na skutečné sociální sítě, protože uživatelé si mohou ke každému odkazu zvolit vlastní ikony. Nezabývá se ani firemními hierarchiemi nebo otevíracími dobami. Na druhou stranu, poskytuje spoustu napojení na externí aplikace, především pak ty statistické nebo e-commercové.

4.2 LinkedIn

Další významnou aplikací je LinkedIn. Ta však směřuje své zaměření spíše na firemní prostředí a pracovní zkušenosti uživatelů. Informace navíc nejsou veřejně dostupné bez registrace. Oproti Linktree, ale obsahuje fulltextové vyhledávání a možnost sledování ostatních uživatelů. Nevýhodou můžou být všudypřítomné reklamy nebo sociální prvky generující rušivé elementy. I přesto že se zaměřuje na firmy, není snadné dohledat firemní struktury nebo otevírací doby.

4.3 AllMyLinks

Méně známou alternativou je služba AllMyLinks. Ta nabízí velice podobné funkce jako Linktree - též nabízí pouze sdílení odkazů na sociální sítě. Stejně jako Linktree neposkytuje podporu ověřování odkazů oproti sociálním sítím, což má za následek nevěrohodnost odkazů a ošklivé uživatelské ikony.

4.4 Swopi

Swopi je další alternativou Linktree. Navíc se ale zabývá výrobou a prodejem chytrých fyzických karet a štítků pro rychlé zobrazení uživatelských profilů. Software je ale značně neintuitivní a neumožňuje více odkazů na jednu sociální síť.

4.5 Firmy.cz

Mezi české nepřímé alternativy by se dala zařadit i služba Firmy.cz, která se ale podobně jako LinkedIn zabývá spíše firmami, a jejich zobrazením na mapě. Nabízí vyhledávání, otevírací dobu a základní kontakty, externí odkazy a firemní struktury jsou však značně omezené. Služba je navíc cílená pouze na české publikum.

4.6 Zlaté stránky

Obdobou Firmy.cz trpící podobnými problémy je další česká služba Zlaté stránky.

5 Návrh vlastní aplikace

Výsledná webová aplikace bude oproti konkurenčním řešením kombinovat více různých typů informací, aby tak pokryla většinu požadavků soukromých osob, firem a objektů.

5.1 Karty

Základní obecnou jednotkou reprezentující takový subjekt či objekt bude karta; odvozenina z anglického překladu vizitky "business card". Každá karta bude jakási moderní rozšířená digitální náhrada klasické fyzické vizitky. Cílem je umožnit vyhledávat a sdílet informace soukromých osob, firem, událostí, produktů, míst, uměleckých děl a podobně. Jedinou podmínkou pro takový subjekt nebo objekt je existence alespoň jednoho údaje důležitého pro ostatní osoby. Tím může být kontaktní údaj, odkaz na jiné webové stránky (např.: sociální sítě), geografická lokace nebo firemní hierarchie.

5.2 Oblíbené karty

Uživatelé aplikace budou mít možnost uložit karty, které je nějakým způsobem zajímají, do svého profilu. Kromě obyčejného uchování oblíbených karet bude možné karty pro lepší organizaci seskupovat do vlastních upravitelných složek a komentovat vlastními soukromými poznámkami. Tyto funkce umožní uživatelům komunikujícím s velkým počtem subjektů vyznat se v kontaktních údajích a poznámkách, například ohledně poslední komunikace nebo podrobností o daném subjektu bez nutnosti ručně takové informace spravovat v textových editorech nebo papírových poznámkách.

5.3 Viditelnost karet

Protože vyplnění takové karty může být zdlouhavý proces, i za předpokladu intuitivního UI (uživatelské rozhraní), je potřeba umožnit uživatelům skrýt karty do té doby, než budou připravené ke zveřejnění. Zároveň ale takovou rozpracovanou kartu může uživatel chtít někomu vzdáleně ukázat, například pro ověření správnosti informací před zveřejněním. Karty tak budou moct mít dvě viditelnosti: veřejná a skrytá. Veřejná karta bude mít jednoduchou zapamatovatelnou URL (Uniform Resource Locator) adresu a bude ji možné vyhledávat pomocí fulltextového vyhledávače. Skrytá karta nebude vyhledatelná pomocí vyhledávače, nicméně uživatelé vlastnící URL odkaz na danou kartu budou mít přístup k jejímu zobrazení. Skrytá karta bude mít navíc možnost vygenerování náhodné URL pro znemožnění uhádnutí URL adresy nepovolenou osobou.

5.4 Karetní informace

Každá karta bude mít možnost zobrazovat různé informace o uživatelích. Definovatelné informace budou vždy dané, tj. budou mít vlastní strukturu v kódu, validaci správnosti hodnot a specializované UI. Aby se předešlo zhoršení čitelnosti informací karet, nebude možné uživatelem definovat náhodné informace. Toto rozhodnutí by mělo pomoci především uživatelům nepříliš zdatných v dodržování přehlednosti většího množství informací. Je však potřeba strukturu navrhnout dostatečně univerzálně, aby bylo možné v budoucnu implementovat další typy definovatelných informací na kartách.

Momentálně podporovanými typy informací budou:

- fotografie/avatar,

- kategorizační štítky,
- popisek,
- základní kontaktní údaje,
- odkazy na profily na sociálních sítí,
- generické odkazy na jakékoliv externí webové stránky,
- geografická lokace,
- hlavní otevírací doba,
- kanceláře, pobočky zaměstnanci.

Pro rychlé vizuální rozpoznání karty bude mít autor k dispozici možnost nahrát vlastní fotografii nebo avatar s logem. Fotografie nebo avatar dokáže dostatečně odlišit jednotlivé karty bez nutnosti čtení názvu nebo popisku ke zjištění, o jakou kartu se vlastně jedná.

Kategorizační štítky Kategorizační štítky budou sloužit pro globální kategorizaci karet pro úvodní povědomí uživatele, o jakou kartu se jedná a hlavně pro vyhledání karet ve stejné kategorii. To se může hodit pro seskupení například osob s určitým povoláním nebo vyznávaným životním stylem. Konkrétní využití je však na konkrétních uživatelích, definovat nemusí žádný nebo i několik.

Popisek Popisek bude sloužit především pro stručnou definici obsahu karty, tedy co daná karta reprezentuje. Předpokládá se proto, že popisek bude zodpovídat některé z následujících otázek?

- Jedná se o osobu, firmu, událost, dílo či předmět?
- Čím se daný subjekt či objekt zabývá v profesní nebo soukromé sféře?
- Proč je daný subjekt či objekt zajímavý?

Popisek bude mít omezenou velikost a nebude povinný, nicméně silně doporučovaný.

Základní kontaktní údaje Základními kontaktními údaji se v případě těchto karet rozumí: telefonní číslo, hlavní emailová adresa, webové stránky a IČO (Identifikační číslo osoby). Uživatel si pak vybere jaké z těchto údajů bude chtít zveřejnit a jaké nikoli.

Odkazy na profily na sociálních sítích Pro většinu karet, zejména pak pro karty reprezentující soukromé osoby, budou nejdůležitější odkazy na sociální sítě. Pro jednoduché rozlišení mezi jednotlivými sítěmi a poskytnutí koncovým uživatelům jistotu, že daný odkaz skutečně směřuje na ověřenou doménu, bude mít systém předdefinované podporované sociální sítě. Každá taková definice bude jakási šablona pro výsledné odkazy na sociální sítě a bude obsahovat oficiální název, dále pak názvy podle kterých bude možné síť vyhledat, ikonu, šablony validních URL adres a kategorie. Uživatel tedy bude moct při tvorbě karty vybrat ručně ze seznamu konkrétní síť nebo nechat aplikaci nalézt správnou síť podle vložené URL adresy.

Šablony sítí budou kategorizované a bude možné je pro jednodušší nalezení vyhledávat podle názvu. Po vložení konkrétní URL adresy vznikne konkrétní odkaz držící informaci o použité šabloně, cílové URL adrese a popisku pro rozeznání různých odkazů na stejné sociální sítě. Díky definovaným šablonám reprezentující rozsah validních URL adres pro konkrétní sociální síť bude aplikace schopna validovat konkrétní URL adresy, zdali odpovídají alespoň jedné šabloně a jestli zadaná URL adresa skutečně existuje. To jednak umožní autory karet upozornit na nesprávné URL adresy, a koncovým uživatelům poskytne zmiňovanou jistotu pravosti.

Generické odkazy na externí webové stránky Generické odkazy budou podobné odkazům na sociální sítě. Rozdíl však bude ve volnosti zadat jakoukoliv validní URL adresu. Taková adresa sice bude stále validovaná aplikací pro existenci, nebude už však validovaná proti žádnému rozsahu URL adres. To umožní autorovi karty odkazovat na jakékoliv webové stránky, ale znemožní poskytnutí jistoty pravosti odkazu pro koncové uživatele. Bohužel není možné jednoduše strojově zjistit, jestli odkaz nevede na podvodnou stránku. Takové stránky je obtížné rozeznat od originálních webových stránek i pro pravidelné uživatele originálních stránek. Navíc podvodné stránky každým dnem přibývají a autoři takových webů jsou stále sofistikovanější.

Geografická lokace Důležitou informací pro firmy, události, pobočky atd. je geografiká lokace sídla. Ta umožní koncovým uživatelům nalézt subjekt či objekt, reprezentovaný kartou, v reálném světě. Zároveň poskytne možnost zobrazit karty v blízkém okolí daného uživatele pomocí mapy. Uživatel tak jednoduše zjistí, jaké firmy, osoby nebo události se nachází v okolí jeho bydliště, zaměstnaní nebo cílové destinaci.

Otevírací doba Firmy a obchody většinou mají nějakou provozní/otevírací dobu, během níž jsou zaměstnanci schopni obsloužit své zákazníky. Tato informace je navíc uživateli vyhledávaná opakovaně, a je tedy nutné, aby byla vždy po ruce a sdělovala vše potřebné. Každá karta tak bude moct definovat otevírací dobu každého dne v týdnu s volitelnou přestávkou. Uživatelé pak kromě definovaných otevíracích dob uvidí i informaci o tom, jaká otevírací doba pro daný den platí, a za jak dlouho daný subjekt bude mít otevřeno, popřípadě kolik času zbývá do konce otevírací doby.

Kanceláře, pobočky a zaměstnanci Nedílnou součástí větších firem a institucí jsou pobočky a kanceláře rozmístěné v různých koutech světa. Každá taková pobočka má své zaměstnance, geografickou lokaci a v některých případech i vlastní otevírací dobu. Autor karty tak může definovat své pobočky či kanceláře a volitelně je obohatit o zmíněné rozšiřující informace. Každá taková pobočka může mít vlastní otevírací dobu, i v případě, že samotná karta může mít již hlavní otevírací dobu definovanou. Pro klienty je pak vhodné mít definovanou i lokaci dané pobočky. Podstatnou částí bude možnost přiřadit zaměstnance nacházející se v dané pobočce pro jejich snadné kontaktování.

Zaměstnanec bude mít definované jméno a pracovní pozici, volitelně pak ještě fotografii a základní kontaktní údaje v podobě emailové adresy a telefonního čísla. Každý takový zaměstnanec bude reprezentován speciálním typem karty pro možnost uložení mezi ostatní oblíbené karty.

5.5 Typy karet

Aby bylo možné definovat pro karty různého zaměření jiné podporované informace, budou existovat různé typy karet. Každý typ bude definovat účel karty, podporované informace a platné použití. Koncoví uživatelé budou schopni díky této univerzalitě ukládat karty mezi oblíbené bez rozdílů. Momentálně dostupnými typy budou obecná karta a karta zaměstnance. Obecná karta bude umožňovat specifikovat všechny výše zmíněné informace a každý uživatel ji bude moct vytvořit přímo. Karta zaměstnance bude automaticky nepřímo tvořena při tvorbě zaměstnanců poboček. To umožní zobrazit detail jednotlivých zaměstnanců, mít unikátní odkaz na každého zaměstnance nebo vyhledávat přímo karty zaměstnanců s referencí na rodičovskou firmu. Typy karet se ale mohou v budoucnu rozrůst, a je proto nutné s takovým rozšířením při implementaci počítat.

5.6 Vyhledávání karet

Velmi podstatnou součástí výsledné aplikace je schopnost vyhledávat a objevovat karty podle požadavků koncového uživatele. Pro zajištění těchto funkcí bude aplikace umět fulltextově vyhledávat a geograficky zobrazovat karty na mapě světa. Díky fulltextovému vyhledávání bude uživatel schopen vyhledávat chtěné karty podle frází obsažených v názvech, popiskách, kontaktních údajích, odkazech, dále podle názvů poboček a karet zaměstnanců. Výsledky takového vyhledávání budou seřazené podle relevantnosti vzhledem k hledané frázi. Mimo konkrétního vyhledávání bude zobrazena mapa světa se zaměřením na jeho současnou lokaci zobrazující karty a pobočky jako body v mapě. Každý bod bude obsahovat souhrn nejdůležitějších informací z reprezentující karty nebo pobočky. Takovou informací může být název, štítky, otevírací doba konkrétního dne, odkazy nebo zaměstnanci. Díky těmto funkcím budou uživatelé schopni objevovat nové subjekty a objekty bez nutnosti znalosti URL adres konkrétních karet nebo jejich údajů.

5.7 Dostupnost aplikace

Cílem aplikace je poskytnout globálně komukoliv bez omezení přístup k veřejným informacím. Na rozdíl od některé konkurence nebude ve výsledné aplikaci vyžadována po uživatelích pro přístup k vyhledávání nebo samotným kartám a jejich informacím registrace. Jedinou opodstatněnou výjimkou pro nutnost registrace by se v budoucnu mohli stát karty s příznakem 18+. Registrovaný účet uživatele by pak sloužil pro ověření věku daného uživatele podle data narození. Tento požadavek nicméně v současné verzi navrhované aplikace není řešen. Dalším požadavkem je umožnit přístup k informacím z jakékoliv země světa. Budoucím cílem je postupně překládat aplikaci do dalších jazyků pro ještě větší dostupnost v zemích mluvících jiným než anglickým jazykem. Přesto nesmí být chybějící jazyk dané země sám o sobě důvod pro nedostupnost aplikace v dané zemi, uživatelé by měli být vždy schopni používat minimálně výchozí anglickou verzi.

5.8 Uživatelské účty

Jak již bylo v předchozí kapitole nastíněno, uživatelské účty budou volitelnou funkcí aplikace přinášející výhody nemožné implementovat bez existujících účtů. Mezi takové výhody bude z počátku patřit tvoření karet a ukládání cizích karet mezi oblíbené. Tyto funkce vyžadují uživatelský účet, aby bylo možné určit, kdo je má právo upravovat a mazat. Uživatelské účty jsou tak tedy spíše bezpečnostním nástrojem pro správu dat ve tvořené webové aplikaci,

nikoliv nástrojem pro získání osobních dat uživatelů pro vlastní zpracování. Uživatelské účty jako takové nebudou reprezentovat vždy jen jednu kartu. Místo toho každý účet bude moct vytvořit několik karet a účet bude představovat jen neviditelného vlastníka.

Kromě těchto hlavních funkcí, správná aplikace musí svým zaregistrovaným uživatelům poskytnout i nástroje pro správu účtů jako takových. Každý uživatelský účet bude identifikován jednoznačně pouze podle emailové adresy. Aplikace nebude vyžadovat žádné další údaje v podobě jména, bydliště a podobně, protože nejsou potřeba pro provoz aplikace. Emailová adresa místo uživatelského jména byla zvolena z důvodu možnosti informovat uživatele o různých událostech v aplikaci bez nutnosti otevírat samotnou webovou aplikaci. Takové události mohou zahrnovat potvrzení změny emailové adresy, upozornění na změnu hesla nebo třeba vyžádání obnovy zapomenutého hesla. Mimo samotné registrace a přihlášení, bude aplikace umožňovat změnu emailu, bezpečně obnovit zapomenuté heslo, kompletně smazat vytvořený účet se všemi jeho daty nebo odhlásit uživatele ze všech přihlášených zařízení. Přihlášení pomocí emailové adresy a hesla bude navíc rozšířeno o přihlášení a registraci skrze poskytovatele třetích stran. To zjednoduší a hlavně zrychlí registraci a přihlášení uživatelů využívající služby Googlu, Facebooku nebo třeba GitHubu. Při registraci v aplikaci pomocí některého z poskytovatelů poskytovatel předá aplikaci základní údaje automaticky a uživatel pouze registraci potvrdí bez nutnosti cokoliv ručně vyplňovat.

5.9 Prémiové funkce

Aby bylo možné obecně dlouhodobě provozovat jakoukoliv webovou aplikaci, je nutné zajistit financování provozu produkčního prostředí. Možností je spousta a záleží na představivosti vlastníka aplikace. Nejčastějšími praktikami jsou: prodej produktů nebo služeb, zobrazování reklam nebo placené plány s prémiovými funkcemi. V případě této aplikace by bylo možné nabízet a prodávat fyzické produkty, například speciální čipové karty. Avšak prodej jakýchkoliv fyzických produktů není hlavním cílem této aplikace, zejména kvůli přidané zátěži pro provoz. Zobrazování reklam při navigování aplikací by bylo možné a nepřidalo by tomuto řešení nadbytečnou zátěž na provoz, nicméně uživatelské rozhraní by značně trpělo svojí razantně sníženou přehledností, což by mohlo vést k frustraci uživatelů. Existuje navíc značné procento uživatelů využívající speciální nástroje prohlížečů pro blokování takových reklam.

Ideální variantou pro tento typ aplikace se jeví zavedení placených plánů s prémiovými funkcemi. Zaregistrovaný uživatel bude mít standardně bezplatný plán poskytující uživateli základní omezené funkce, ale bude si moct aktivovat prémiový placený plán odemykající všechny zbylé funkce, které bude aplikace nabízet. Toto řešení zamezí zhoršení uživatelského rozhraní a celkově nebude omezovat koncové uživatele. Zároveň řešení nebude ovlivněno nástroji pro blokování reklam a nebude narušovat soukromí uživatelů v podobě sbírání uživatelských dat. Nevýhodou oproti reklamám je přidaná náročnost implementace systému umožňující uživatelům přepínat a platit plány, a hlavně zamezit uživatelům s bezplatným plánem využívat prémiové funkce. Avšak na rozdíl od fyzických produktů, u nichž je přidaná zátěž stálá, týká se tato zátěž především počátku implementace. Další možnou nevýhodou je malý počet uživatelů platících si prémiový plán, protože pokud nebude dostatečný počet platících zákazníků, nebude dostatečný příjem pro produkční provoz a rozvoj.

Konkrétním řešením pro tuto webovou aplikaci bude zavést limity pro bezplatný plán a umožnit uživatelům jednoduše přejít na placený. Kromě samotných plateb bude možné uplatňovat kupóny, které aktivují prémiový plán na konečnou dobu, a poté se plán automaticky přepne zpět na bezplatný. Pokud uživatel bude mít na svém účtu aktivované prémiové

funkce při vypršení prémiového plánu, bude nucen data zahrnující prémiové funkce odstranit, jinak budou automaticky skryta. Samotné limity se budou týkat především karet. V případě bezplatného plánu bude mít uživatel k dispozici omezený počet karet, které může vytvořit. V rámci každé karty pak nebude mít tvořit pobočky a zaměstnance. Důležité je připravit takový systém, aby bylo možné co nejjednodušeji limity upravovat a případně přidávat další s novými prémiovými funkcemi.

5.10 Bezpečnost

V dnešní době s přibývajícími podvodnými stránkami a sofistikovanějšími útočníky je čím dál více důležité myslet i na zabezpečení webových aplikací. Kromě již základní běžné praktiky hashování hesel bezpečným algoritmem, je nutné zabezpečit mnohem více. V první řadě jde komunikaci mezi prohlížečem a serverem, aby nemohl útočník odposlouchávat posílaná data a následně se vydávat za jiného uživatele. S kradení identity uživatelů souvisí i další praktiky, jako jsou CSRF (Cross Site Request Forgery) nebo kradení sezení uživatele. Z pohledu samotného kódu aplikace je zase nutné připravit systém práv, a taková práva uživatelů před vykonáním akcí ověřovat, aby jeden uživatel nemohl upravovat data jiných uživatelů nebo jinak škodit.

5.11 Uživatelské rozhraní

Aby byla aplikace schopna konkurovat, musí kromě své základní funkčnosti poskytovat i přehledné a vzhledné UI. Navržené UI by tak mělo uživateli zjednodušovat přístup k informacím a akcím, které chce provést bez nutnosti zdlouhavého hledání napříč aplikací. Kvůli velkému procentu využívání internetu na mobilních zařízení je žádoucí optimalizovat webovou aplikaci i pro běh na menších zařízení. Hlavním cílem je uživatelům mobilních zařízení poskytnou webovou aplikaci přibližující se svým rozvržením a pohodlím nativním aplikacím, stejně jako pro uživatele osobních počítačů s velkými obrazovkami, a to díky dynamické přizpůsobitelnosti UI. Tento koncept se navíc v budoucnosti dá dále rozšířit o skutečnou mobilní aplikaci za pomoci některých technologií, aniž by bylo nutné tvořit separátní nativní aplikaci se stejnými funkcemi.

6 Technologie pro vývoj

Vývoj jakékoliv aplikace od nuly až po provoz v produkčním prostředí zahrnuje spoustu kroků, a každý takový krok lze pojmout několika způsoby. Před samotným vývojem je tak potřeba důkladně promyslet a naplánovat, jaké má projekt požadavky, co je cílem samotného projektu, a také, jaký je rozpočet pro jeho vývoj. Od těchto cílů se pak odvíjí způsob vývoje a především výběr technologií a vývojových nástrojů. Ty spolu musí být dostatečně kompatibilní a tvořit efektivní celek. Při výběru nevhodných technologií se může totiž stát, že v průběhu vývoje vývojáři narazí na problém, který je buď s použitými technologiemi neřešitelný, nebo je nutné problém obejít, a to může způsobit komplikace v budoucnu v podobě malé výkonnosti aplikace nebo příliš složitého zdrojového kódu.

V dnešním moderním světě existuje spousta webových technologií, které umožňují a většinou i značně usnadňují vývoj webových stránek, aplikací či mikroslužeb. Avšak nové alternativní technologie přibývají každým dnem a může být mnohdy obtížné i pro zkušené vývojáře zorientovat se a vybrat ty správné pro daný projekt. Při rozhodování je důležité vybírat nejen podle popularity, ale především podle typu projektu, ten totiž může zásadně ovlivnit, jaké technologie jsou vhodné a které nikoliv.

Kromě webových technologií použitých přímo pro vývoj aplikace je vhodné se při návrhu zabývat i technologiemi a nástroji pro:

- uchovávání uživatelských dat,
- uchovávání souborů a poskytování jejich variant,
- návrh uživatelského rozhraní,
- komunikaci s uživateli jinými kanály, než je samotná aplikace,
- běh aplikace ve vývojovém a produkčním prostředí.

Webové technologie se primárně rozdělují na: front-endové a back-endové. Obě obsahují rozdílné technologie zejména kvůli odlišnosti zaměření. Existují ale i technologie spadající více či méně do obou kategorií, avšak v každé plní trochu jiný účel. Nejrozšířenějším takovým kandidátem je programovací jazyk Javascript (JS) dominující v současné době oběma kategoriím.

6.1 Technologie pro návrh uživatelského prostředí

Návrh UI je jedním z nejdůležitějších kroků pro úspěšnou aplikaci. Společně s UX (uživatelský zážitek) z velké části rozhodují, zda aplikace nebo webová prezentace uživatele zaujme a bude jí nadále používat nebo přejde ke konkurenci. Je důležité navrhnout moderní, přehledné a hlavně intuitivní UI, aby uživatele podvědomě navádělo k jeho cíli. Dobré je myslet i na uživatele s různými omezeními. Například pro úplnou slepotu je možné do webové stránky přidat speciální metadata umožňující prohlížečům správně předčítat obsah stránky. Pro barvoslepost je zase dobré myslet na správný kontrast vybrané barevné palety. Návrh UI ještě před samotnou implementací může navíc zásadně šetřit čas vývojáře, protože v úvodních krocích se design často mění, a pokud by měl vývojář každou úpravu ještě implementovat, zabralo by to dvojnásobek času.

Pro návrh UI a UX existují specializované grafické nástroje. Použít lze i tradiční grafické nástroje, ty ale nenabízí funkce pro vzájemnou kolaboraci mezi členy v týmu nebo pro přidávání interaktivních prvků. Nejpopulárnějšími nástroji se staly Figma, Sketch a

Adobe DX, a z toho Figma a Adobe XD poskytují dokonce bezplatné varianty s určitými omezeními [1]. S pomocí nich je snadné navrhnout základní drátový model pro návrh struktury, grafický model pro vizualizaci vzhledu, i interaktivní demo. Tyto modely pak značně usnadňují implementaci UI, protože vývojář nemusí vymýšlet vzhled a strukturu za pochodu. Samotný výběr konkrétního nástroje je otázkou operačních systémů designérů a vývojářů, ceny, případně předchozích znalostí.

6.2 Front-endové webové technologie

Front-endové technologie se věnují především přímé interakci s uživatelem pomocí webového prohlížeče. Definují tak vzhled a samotné ovládání aplikace pro koncového uživatele. Tyto technologie pak zpravidla jaksi obalují a zpracovávají funkcionality serverů tak, aby se koncovému uživateli s danými daty pracovalo co možná nejjednodušeji a nebyl nucen pracovat se surovými daty. V drtivé většině pak neposkytují uživatelům jen data samotná, ale hlavně informace z nich zpracované.

Tyto technologie by se daly ještě pomyslně rozdělit na standardizované a ostatní. Standardizované tvoří základní stavební kameny všech webových stránek a aplikací, kterým rozumí webové prohlížeče. Ostatní pak představují jakési nadstavby nad těmi standardizovanými a snaží se je nějakým způsobem rozšířit či zjednodušit. Tyto už standardizované nijak nejsou a mohou se proto často zásadním způsobem měnit od verze k verzi. Je jich mnohem více a každým dnem vznikají nové, vývojářům značně ulehčují práci, a proto jsou tolik oblíbené.

6.2.1 Standardizované front-endové technologie

Jsou definovány standardy společností W3C, což umožňuje vyvíjet v podstatě univerzální stránky a aplikace běžící na koncovým uživatelem zvoleném prohlížeči. Zástupci těchto technologií jsou HTML (HyperText Markup Language), CSS (Cascading Style Sheets), JS a nově i WebAssembly, a představují tedy základní nástroje pro tvorbu jakýchkoliv stránek a aplikací jež je možné provozovat na internetu. [2]

Nejdůležitější je značkovací jazyk HTML, bez kterého se nelze obejít. Tento jazyk definuje základní strukturu a obsah pomocí tzv. značek, které se do sebe zanořují a tvoří tak stromovou strukturu bloků dokumentu. Struktura vychází z jazyka XML (Extensible Markup Language), který je obohacen hlavně o vlastní značky s přidaným významem pro orientaci prohlížečů při vykreslování stránek. [3]

Další velmi důležitou technologií je stylovací jazyk CSS definující vzhled struktury dokumentu pro koncové uživatele. Pomocí tohoto jazyka je možné upravovat písma, barvy, pozice prvků dokumentu a dokonce i animace. Vzhled stránek a aplikací je čím dál tím důležitější jak pro snazší orientaci, tak pro odlišení od konkurence. [4]

V dnešní době už téměř stejně důležitý stavební prvek jako HTML nebo CSS je skriptovací jazyk JS umožňující na front-endu dynamicky, v průběhu interakce uživatele se stránkou, modifikovat strukturu a vzhled původní stránky či aplikace. To otevírá obrovské možnosti pro tvorbu komplexních animací, dynamického donačítání obsahu ze serveru, her, přehrávačů videí a mnoho dalšího. [5]

Alternativou, případně doplňkem, k JS je nový standard WebAssembly. Je to nízko úrovňový jazyk podobný Assembleru sloužící jako univerzální jazyk, do kterého je možné překládat kód z mnoha již existujících jazyků, jako např. C++ nebo Rust. Oproti JS má ohromnou výkonnostní výhodu, což otevírá pro výpočetně náročné aplikace možnost běžet

v klasickém prohlížeči. WebAssembly se ale dá použít společně s JS a lze tak využít výhody obou jazyků. [6]

6.2.2 CSS preprocesory, frameworky a knihovny

Používání základního jazyka CSS může být zejména u větších projektů mnohdy těžkopádné a špatně udržovatelné. Proto existují preprocesory, frameworky a knihovny ulehčující vývojářům zásadním způsobem práci.

Preprocesory jsou jazyky představující jakousi nadstavbu základního CSS a rozšiřují ho o vlastní funkcionality. Problém je v tom, že prohlížeče rozumí pouze standardizovanému CSS, proto musí být tyto jazyky překládány zpět do CSS.

Frameworky a knihovny pak seskupují předpřipravené kusy CSS kódu připravené pro rychlé použití při tvorbě stránek. Díky nim je možné rychle stylovat stránky, protože vývojář nemusí pomocí CSS konfigurovat vše, co se týče vzhledu, ručně. Místo toho může použít výše zmíněné kusy kódů nebo již celé hotové bloky.

Nejrozšířenějším preprocesorem je jednoznačně Sass (Syntactically Awesome Style Sheets). Oproti základnímu CSS totiž nabízí širokou škálu funkcionalit od vnořování, funkcí pro automatizaci až třeba po dědičnost. Prohlížeče ale tomuto jazyku nerozumí, proto je nutné ho překládat do CSS, pomocí předpřipravených nástrojů. Tento jazyk mimo jiné hlavně usnadňuje organizaci stylovacího kódu, a je proto jednodušší ho v budoucnu rozšiřovat a modifikovat. [7]

Méně používanou alternativou k Sass je Less (Leaner Style Sheets), který stejně jako Sass rozšiřuje základní CSS o další funkcionality jako vnořování nebo dědičnost, a je též nutné ho překládat do standardizovaného CSS. [8] Less ale oproti Sass ve spoustě věcech pokulhává, a proto není mezi vývojáři tolik oblíbený. Sass má např. prokročilejší možnosti scriptování zahrnující cykly a podmínky blížící se programovacím jazykům a umožňuje tak jakousi automatizaci generovaného CSS kódu. Naproti tomu Less poskytuje jen základní cykly a proměnné, což může být pro některé vývojáře dosti omezující. Sass má také mnohem pokročilejší a efektivnější dědičnost, nastavení neduplikuje ale spíše nahrazuje. [9].

Dlouholetým favoritem mezi frameworky je Bootstrap. Ten obsahuje velké množství základních předpřipravených šablon, nastavení typografie, formulářů, tlačítek atd. Kromě toho lze využít komunitních šablon celých stránek a vytvořit s jeho pomocí finální stránky během velmi krátké doby. Bootstrap je ale paměťově náročný a mnohdy těžkopádný, a proto se ho, pokud využívají jen část jeho funkcionalit, někteří vývojáři zbavují a nahrazují jednoduššími alternativami či přímo nově vznikajícími standardy v samotném CSS. [10]

Poměrně novou oblíbenou alternativou k Bootstrapu je TailwindCSS poskytující obecné CSS třídy pro pozicování a stylování HTML struktury. To umožňuje, stejně jako u Bootstrapu, rychlou tvorbu webových stránek a aplikací bez nutnosti hlubších znalostí samotného jazyka CSS. [11] Oproti Bootstrapu je ale mnohem univerzálnější a díky univerzálním třídám, stránky nevypadají podobně jako v případě Bootstrapu, který poskytuje spíše předpřipravené bloky či rovnou celé stránky. TailwindCSS je také méně paměťově náročný, a proto nepředstavuje tak velkou zátěž jako Bootstrap. [12]

6.2.3 Javascriptové frameworky a knihovny

Frameworky a knihovny představují v JS (oproti čistému Javascriptu) sadu předpřipravených nástrojů pro snazší a rychlejší vývoj. Dříve vývojáři hojně využívali pro chybějící základní funkcionality knihoven, protože nebyl v některých ohledech tak pokročilý. Dnes je tomu naopak a není problém používat pouze čistý JS, přesto se dnes spíše používají fra-

meworky a knihovny pro snadnou tvorbu reaktivních znovupoužitelných komponent. Tytp komponenty představují samostatné bloky, jako jsou např. formuláře nebo tlačítka, udržující si vlastní data a při změně se automaticky překreslují.

Momentálně nejznámější knihovnou pro tvorbu takových komponent je React, jenž byl vytvořen především pro tvorbu interaktivních GUI (grafické uživatelské rozhraní) webových stránek a aplikací. Zaměřuje se hlavně na zobrazovací vrstvu, což zahrnuje reaktivnost a překreslování komponent a skládání výsledných stránek z takových komponent. Ostatní funkcionality pak spíše přenechává již existujícím specializovaným knihovnám, které jsou často kvůli delšímu vývoji a velké komunitě lepší. Stejně jako další podobné frameworky je postaven na systému reaktivity, v němž má každá komponenta vlastní data, ta když se změní, část stránky se automaticky překreslí s novými daty bez nutnosti explicitního překreslení vývojářem. Nicméně React může být složitější na naučení, kvůli většímu množství možných vývojových přístupů. Má ale velkou komunitu a s tím i spojené velké množství materiálů. Navíc je vyvíjen samotným Facebookem, což zaručuje, že se bude ještě dlouhou dobu rozvíjet a jen tak nezmizí. [13]

Další čím dál více oblíbenou alternativou pro snadnou tvorbu interaktivních GUI je Vue. Stejně jako React se zaměřuje pouze na zobrazovací vrstvu s využitím reaktivních komponent a ostatní funkce přenechává již existujícím knihovnám. To umožňuje vývojářům vybrat správné technologie pro daný projekt a nemuset se omezovat vybraným hlavním frameworkem. Jeho velkou výhodou je flexibilita míry integrovanosti do projektu. Lze ho totiž použít buď jako pomocníka při tvorbě jednoduchých komponent ve stávajícím projektu, nebo jako celou platformu pro tvorbu sofistikovaných webových aplikací. [14] Vue je oproti Reactu mnohem jednodušší na naučení, jak pro zkušené vývojáře, tak pro začátečníky, a společně s ne tak často měnícím se kódem je čím dál více používán nejen jednotlivci, ale i velkými firmami. Nemá sice zatím tak velkou komunitu, ale je dostatečně velká na to, aby se o něm dalo uvažovat jako o solidní možnosti pro nový projekt. [15]

Svelte je nejnovějším přírůstkem do skupiny nejpoužívanějších frameworků pro tvorbu GUI, avšak přináší odlišný způsob tvorby interaktivních stránek a aplikací. Stejně jako Vue je možné ho použít jen okrajově jako pomocníka nebo pomocí něj vytvořit celou aplikaci. Od konkurence (React a Vue) se Swelte zásadně liší ve formě, v jaké běží v prohlížeči. Zatím co React a Vue používají svoji logiku za chodu aplikace k sestavování stránek, Svelte překládá kód do čistého kompaktního JS kódu. To má vliv především na rychlost celé stránky či aplikace, zejména pak u složitých aplikací, ovšem za cenu vyšší rychlosti je zde omezení v podobě vlastního scriptovacího jazyka. [16] Ten se podobá klasickému JS, ale nelze říct, že je s ním zaměnitelný. V případě potíží při vývoji tak může být pro některé vývojáře obtížné danou situaci vyřešit. [17]

Mezi tyto frameworky by se dal zařadit ještě Angular/Angular 2, ten už ale ztrácí na popularitě a je používán spíše v korporátní sféře. Angular je spolehlivý a mocný nástroj a poskytuje obdobné funkcionality jako konkurence, nicméně nemá moc dobrou dokumentaci a je obtížný na naučení. [18]

6.3 Back-endové webové technologie

Back-endové technologie běží na serverech, a proto se na rozdíl od front-endových vůbec nedostanou do styku s koncovými uživateli. Místo toho se zaměřují hlavně na práci s daty a informacemi. Tím může být poskytování relevantních dat front-endovým technologiím nebo provádění výpočetně náročných operací nad uživatelskými daty.

Pro komunikaci mezi těmito dvěma světy se používá standardizovaný aplikační protokol HTTP (Hypertext Transfer Protocol). Ten definuje strukturu přenášených dat, tak aby jakýkoliv server s jakoukoliv technologií mohl bez problému komunikovat standardizovaným způsobem s jakýmkoliv prohlížečem. [19]

Díky tomuto protokolu servery nejsou omezeny základními standardizovanými technologiemi jako tomu je v případě HTML, CSS a JS, a vývojáři tak mohou používat teoreticky jakýkoliv programovací jazyk a ekosystém s ním spojený.

6.3.1 Java

Java je velmi univerzální objektově orientovaný programovací jazyk, jenž může být provozován téměř na jakémkoliv hardwaru od serveru až po mikrořadiče. Umožňuje vyvíjet ohromnou škálu aplikací počínaje desktopovými aplikacemi, přes hry až po třeba ty serverové aplikace. [20] Ačkoliv je psaní kódu v Javě oproti jiným jazykům mnohdy zbytečně zdlouhavé, Java nabízí velmi stabilní a zpětně kompatibilní prostředí pro vývoj. Není se proto třeba obávat, že by se v příští verzi udály velké zpětně nekompatibilní změny nebo dokonce by přestala být zcela podporována. Java se osvědčila jako vhodná i pro výpočetně náročné serverové aplikace potřebující provádět několik paralelních operací najednou.

V dnešní době jsou serverové aplikace čím dál tím složitější a je v podstatě nutné pro udržení přehledného a snadno rozšiřitelného kódu používat nějaký framework či knihovnu poskytující nástroje pro práci s protokolem HTTP a všeho s tím spojené.

Jedním z nejpoužívanějších frameworků pro vývoj webových aplikací v Javě je Spring. Věnuje se velké škále různých nástrojů pro vývoj a mezi ty hlavní patří systém Dependency Injection (DI) usnadňující práci s mnoha objekty, které jsou mezi sebou propojené; spravovat tyto vazby svépomocí by bylo obtížné. Neméně důležitou funkcionalitou je systém událostí poskytující vývojářům možnost v rámci programu jednoduše vyvolávat události, na které se lze kdekoliv v kódu napojit a spouštět potřebné procesy. [21] Pro vývoj běžných webových stránek pak poskytuje nástroje pro implementaci MVC (Model-View-Controller) architektury, REST (Representational State Transfer) API (Application programming interface) architektury atp. [22] Nicméně Spring poskytuje velké množství dalších nástrojů a funkcionalit, a proto může být pro začínající vývojáře jeho rozsáhlost odrazující. Avšak znalost tohoto frameworku se značně vyplatí, hlavně pak při tvorbě velkých projektů; příkladem mohou být třeba e-shopy.

Hojně využívanou alternativou (především v korporátní sféře) k Springu je Java EE, resp. v současné době Jakarta EE (pouze jiný název). Ta definuje oficiální standardy pro různé způsoby zpracování HTTP požadavků, od nejzákladnějšího obecného zpracování až po např. formátování odpovědi podle daných požadavků. Zároveň definuje standardy např. pro mapování objektů na databázové tabulky, DI a atp. Tím, že se jedná ve své podstatě pouze o standardy, je zapotřebí vybrat pro vývoj knihovny implementující právě tyto standardy. [23]

Obě technologie jsou velmi populární a velmi mocné. Každá z nich má své klady a zápory a nelze proto jednoznačně určit lepší z nich, nicméně Spring je obecně jednodušší při vývoji, a navíc je zdarma. Java EE naproti tomu přichází s Oracle licencí, a proto se více hodí pro použití ve velkých korporátních společnostech. [24] Existuje spousta dalších menších frameworků, tyto jsou však nejpoužívanější.

6.3.2 Javascript

V dnešní době je možné scriptovací jazyk JS používat nejen k vývoji interaktivního GUI, ale také k vývoji serverových aplikací zpracovávajících HTTP požadavky za pomoci běhového prostředí Node.js, které využívá jádro V8 stejně jako prohlížeče. [25] Ačkoliv JS běží pouze jednovláknově, je poměrně výkonný pro obsluhu velkého množství menších dotazů, a to díky systému událostí, v němž se zpracovává vždy jen ten nejdůležitější požadavek. Není ale úplně vhodný pro výpočetně náročné operace, protože pak může blokovat ostatní požadavky, právě kvůli neschopnosti rozdělit požadavky mezi více vláken. [26] Pro některé vývojáře může být použití výhodou, protože v případě použití JS též na front-endu, nemusí řešit rozdílné jazyky a knihovny. Jednoduše použijí jeden jazyk a podobné knihovny. Stejně jako v případě Javy existuje mnoho frameworků a knihoven poskytujících předpřipravené nástroje pro ulehčení tvorby webových stránek.

Momentálně nejpoužívanějším frameworkem pro back-endový JS běžící na Node.js je Express.js. [27] Ten umožňuje především přijímat samotné HTTP požadavky a odesílat HTTP odpovědi, dále vývojářům poskytuje prostředky pro zpracovávání těchto požadavků pomocí architektury MVC, REST API atd. Mimo jiné je pro spoustu ostatních frameworků základním stavebním kamenem, ke kterému pak přidávají své funkcionality. [25]

Next.js je rovněž frameworkem běžícím na Node.js, ale cílí na front-endovou knihovnu React. Jeho hlavním úkolem je zjednodušení jeho provozu v produkčním prostředí. Může být totiž obtížné nastavit prostředí serveru pro správné poskytování Reactu pro front-end. Kromě výše zmíněného jej rozšiřuje např. o schopnost vykreslení částí nebo celků stránek již na serveru a prohlížeči posílá připravenou stránku či aplikaci. [28]

Obdobou Next.js pro front-endový Vue je Nuxt.js. Ten poskytuje dost podobné funkcionality jako usnadnění provozu Vue v produkčním prostředí, sestavování částí nebo celků stránek nebo třeba jednodušší podporu pro navigaci mezi jednotlivými stránkami. [29]

Při výběru mezi Nuxt.js a Next.js tak záleží především na volbě front-endového frameworku či knihovny nikoliv obráceně, protože nelze použít Nuxt.js společně s Reactem a naopak.

6.3.3 PHP

Dalším, i přes jeho stáří, stále velmi oblíbeným jazykem je PHP (PHP: Hypertext preprocessor) zaměřující se především na tvorbu webových stránek a aplikací. [30]

Tento jazyk je již sám o sobě mocný a dokáže spoustu věcí. Avšak poskytuje spíše základní nástroje a vývojáři musí tak psát pokročilejší logiku pokaždé od nuly. Proto existují frameworky poskytující jistou úroveň abstrakce, čímž nezbytnost tvorby základní logiky ze strany vývojáře, jako třeba navigaci mezi stránkami, obsluhu požadavků atp., která je mnohdy pro většinu stránek a aplikací podobná, ne-li stejná. [31]

V České republice je velmi oblíbený český framework Nette díky jeho přehlednosti a univerzálnosti. Poskytuje širokou škálu nástrojů a funkcionalit podobně jako Spring pro Javu. Též umožňuje vývojářům využít hotového systému DI, implementovat MVC architekturu nebo třeba používat vlastní šablonovací systém pro tvorbu zobrazovací vrstvy v rámci MVC. [32]

Na poli globálně populárních frameworků dominuje především Laravel. Mimo základní funkce přináší schopnost stavět komplexní webové aplikace s velkou rychlostí a bezpečností v kombinaci s jednoduchostí vývoje, kterou přinášejí předpřipravené nástroje starající se o běžné repetitivní úkoly. Kvůli jeho velké popularitě přichází s velkým ekosystémem v

podobě např. dostupnosti hostujících serverů s instalací na jedno kliknutí nebo velkého množství návodů. [31]

Největší konkurencí Laravelu je Symphony, který je starší a má podobně velkou komunitu i ekosystém. Narozdíl od Laravelu je rozdělen do komponent podle zaměření funkcionalit, a je tak možné je používat samostatně. [31] Symfony oproti Laravelu poskytuje nástroje pro velké projekty jako např. škálování. Nicméně se může jevit jako složitější na pochopení pro začínající vývojáře a kvůli externím knihovnám může být výrazně pomalejší než Laravel. Naproti tomu Laravel se často mění a může být proto obtížné stávající projekty aktualizovat. [33]

6.3.4 C#

Podobně jako Java je C# univerzální objektově orientovaný programovací jazyk pro tvorbu nejen desktopových aplikací a her, ale právě i webových stránek a aplikací. K tvorbě webových stránek existuje oficiální framework ASP.NET, který rozšiřuje základní schopnosti komunikace pomocí HTTP protokolu o pomocné nástroje pro snadný vývoj webových aplikací, REST API nebo třeba komunikace mezi prohlížečem a serverem v reálném čase. [34]

Zajímavou a nadějnou novinkou v rámci frameworku ASP.NET je Blazor. Ten umožňuje vývojářům vyvíjet i front-endovou interakci s koncovým uživatelem pomocí pouze C#, HTML a CSS bez jakékoliv nutnosti znát JS, přičemž Blazor nabízí dvě možnosti, jak tohoto docílit; a to pomocí WebAssembly, nebo pomocí již zmíněné komunikace mezi prohlížečem a serverem v reálném čase. V případě WebAssembly se kód napsaný v C# překládá právě do WebAssembly a běží tak čistě v prohlížeči jako front-endová technologie. Pokud si ale vývojář vybere druhou možnost, v prohlížeči běží jen malý komunikační nástroj delegující operace spojené s interaktivním front-endem na server, kde je kód vykonáván přímo v C#. Obě možnosti pak dovolují používat již existující knihovny pro .NET ekosystém, což může být pro některé vývojáře obrovskou výhodou stejně jako fakt, že nemusí programovat v JS. Nicméně vzhledem k tomu, že je tato technologie poměrně nová, nemusí být pro všechny projekty vhodné spoléhat se na nezaběhnutou a neověřenou technologii, protože není jasné jestli budu ještě dlouhé roky podporována, a navíc nemá z daleka tak velkou komunitu jako např. React nebo Vue. [35]

6.4 Databázový systém

Databázový systém je stěžejní technologie pro uchování a agregaci uživatelských dat. Bez komplexních databázových systémů by nebylo možné efektivně uchovávat strukturovaná data, nad kterými lze vyhledávat potřebné informace a data agregovat pro různá shrnutí, přehledy a grafy. Výběr správného systému je proto jedním z nejdůležitějších kroků před implementací samotného řešení, protože přímo zásadním způsobem ovlivňuje, jak se poté bude s daty ve výsledné aplikaci nakládat, a jaká omezení a výhody vybraný systém přináší.

Existuje spousta databázových systémů s velmi odlišnými zaměřeními a přístupy ke struktuře dat. Různé systémy se tak často kombinují, každý poskytuje jiná data pro jiné účely. Nejčastějším takovým rozdělením u klasických webových aplikací je použití jednoho databázového systému pro uchování originálních dat sloužící jako hlavní zdroj dat, a jako doplněk se pak často vybírá systém pro např.: fulltextové vyhledávání agregující pouze nejnovější data z hlavní databáze pro vyhledávání. Existují samozřejmě i další specializované systémy s různými zaměřeními, jako je mezipamět, vyhledávání podle pokročilých kritérií, uchovávání speciálních dat atd. Vyvíjená webové aplikace však bude potřebovat pouze

uchovávat a vyhledávat data, postačí proto pouze jeden databázový systém jako zdroj dat a jeden pro vyhledávání.

Existují dvě kategorie databázových systémů podle způsobu jakým pracují s daty: SQL (Structured Query Language) a NoSQL (Not only SQL). SQL je standardizovaný jazyk postavený na principu relačních databází umožňující pracovat s daty pomocí tabulek. Jedná se o velice univerzální nástroj pro ukládání, vyhledávání a agregaci dat dělající z tohoto jazyka nástroj použitelný pro většinu projektů. Nicméně i přesto, že je tento jazyk standardizovaný, každý databázový systém postavený na SQL má vlastní implementaci a často obsahuje rozšíření nebo změny v syntaxi. [36] Odlišnosti od standardu ale nejsou příliš velké, a je proto možné celkem jednoduše přecházet mezi jednotlivými systémy, bez nutnosti učení se kompletně nového jazyka. NoSQL systémy již nijak standardizované nejsou a každý takový systém obvykle přichází s vlastním vyhledávacích jazykem a strukturou dat. Naštěstí se během své existence NoSQL systémy kategorizovaly čtyřmi široce používanými datovými strukturami. Nejrozšířenější jsou databáze pracující s JSON (JavaScript Object Notation) či XML dokumenty a databáze využívající strukturu klíč-hodnota. V krajních případech se používají i databáze, podobné relačním, ukládající data v tabulkách nebo databáze pracující s grafy a uzly. [37]

SQL databáze, jak již bylo zmíněno, ukládají data v tabulkách neboli relacích, v nichž každý řádek reprezentuje jednu entitu/jedince a sloupce reprezentují jednotlivé atributy dané entity. Jednotlivé relace lze navíc mezi sebou propojovat k vytvoření složitějších struktur pomocí vztahů. Díky tomu je možné např.: k entitě logicky navázat kolekci jiných a úplně odlišných entit, které spolu souvisí nebo tvořit hierarchie entit stejného typu. Možnosti návrhu struktury dat jsou tak celkem rozsáhlé, bohužel pro aplikace využívající OOP (Objektově orientované programování) je celkem obtížné mapovat tabulky na jednotlivé objekty. Při prvním pohledu se může zdát, že objekty a tabulky se dají snadno mapovat, praxe ukazuje mnoho skrytých komplikací.

NoSQL databáze pracující s JSON dokumenty zjednodušující právě zmiňované komplikace při mapování na objekty, protože JSON struktura vychází právě z objektového přístupu. To zároveň umožňuje jednodušší změnu struktury při změně objektů v aplikaci, ale taky velice ztěžuje agregaci dat a vyhledávání s takovými daty. Nejjednodušším databázovým systémem je systém využívající strukturu klíč-hodnota, v níž každá entita má svůj unikátní klíč a jednu hodnotu, připomínající tak tabulku o dvou sloupcích. Mezi příklady patří: mezipaměť pro rychlé získání konkrétních entit, nákupní košíky nebo třeba preference uživatele. Pro analytická data jsou vhodné databáze pracující se sloupci podobně jako relační databáze. Narozdíl od relačních databází jsou ale data uložená ne po řádcích, ale po sloupcích, což umožňuje jednodušší agregaci dat, jako je výpočet sumy. Tato struktura má ale nevýhodu ve velmi pomalém ukládání, protože vyžaduje několika násobné zapisování dat na disk. Posledním rozšířeným typem NoSQL databází je grafová struktura zaměřující se na vztahy mezi entitami. Podle vztahů lze pak jednoduše vyhledávat. Tyto databáze se používají většinou spíše jako sekundární databáze pro analýzu dat a zkoumaných problémů. [38]

6.4.1 Databázový systém pro ukládání dat

Jak již bylo zmíněno, databázové systémy pro ukládání dat se používají jako primární zdroj s nejaktuálnějšími i historickými daty. U těchto systémů je pak téměř vždy vyžadované pravidelné zálohování dat a schopnost zotavení se z pádu celého systému, protože v případě jakéhokoli výpadku by mohlo dojít ke ztrátě často nenahraditelných dat, a to může dále

vést ke ztrátě uživatelů nebo dokonce zisků. Stěžejní je pak schopnost zajistit integritu dat při paralelním zápisu a čtení dat, aby nedošlo k situacím, kdy každý uživatel vidí stejná data v jiných verzích, nebo k vzájemnému přepisu. To totiž může v lepším případě vést k mylným informacím zobrazovaným uživatelům a v tom horším, až třeba ke ztrátě peněz. Z těchto důvodů existují v databázích transakce definující sled kroků, které musí být splněny při změně dat. S tím souvisí množina vlastností zvaná ACID (atomicita, konzistence, izolace, trvalost). Atomicita říká, že databázová transakce rozdělená na více kroků musí být splněná celá. Pokud alespoň jedna její část selže, selže celá transakce, čímž je zaručena konzistence změny dat v rámci transakce. Vlastnost konzistence zaručuje správnost dat oproti pravidlům nastaveným v daném datovém modelu tak, aby nebylo možné do relací zapsat nevalidní data. Izolace transakcí zajišťuje nepřepisování stejných dat, pokud dojde k paralelní změně ve více transakcích. Pokud jedna transakce změní nějaká data, ostatní transakce musí počkat, až předchozí transakce skončí, a až tehdy mohou data znovu změnit. Aby bylo možné zaručit, že se data změněná transakcí, nemohou ztratit, existuje vlastnost zvaná trvalost dat. [39]

Nejpoužívanějšími SQL databázemi splňující výše zmiňované požadavky jsou PostgreSQL, MySQL, MariaDB nebo třeba Oracle DB. Všechny kromě Oracle DB jsou open source a může je využívat kdokoliv bez nutnosti placené licence. To neplatí o Oracle DB vyžadující placenou licenci, za kterou uživatel dostane nejen samotnou databázi, ale i podporu ze strany firmy Oracle jenž využívají především firmy velkých rozměrů, které mají v takových databázích vyšší milióny dat, a proto potřebují prvotřídní a přímou podporu ze strany poskytovatele databázového systému. Většina firem a jedinců si však toto nemůže dovolit, a proto volí spíše ostatní databáze, které díky rozsáhle komunitě poskytují mnohdy stejné funkce bez větších omezení . Díky internetovým diskuzím mají také uživatelé těchto databází přístup k diskuzním fórům nahrazující do jisté míry placenou podporu. Liší se tak spíše v krajních případech, jako jsou podporované datové typy, uživatelská rozšiřitelnost systému nebo v podporovaných indexech. Např.: PostgreSQL obsahuje podporu pro pole hodnot, rozsahy hodnot a pokročilou práci s datovým typem JSON. Naproti tomu MySQL a MariaDB mají omezenější výčet podporovaných datových typů a užší schopnosti práce s datovým typem JSON.

Populární alternativou k tradičním SQL databázím se stávají NoSQL databáze, a to díky jejich flexibilnějšímu přístupu. Zdaleka nejpoužívanější takovou databází je MongoDB, která pro ukládání dat využívá JSON dokumenty. Její popularita spočívá především ve snadnosti mapování dat na objekty při zachování ACID vlastností. Pro běžné webové aplikace, nezabývající se např.: složitými statistickými výpočty a agregáty, poskytuje MongoDB jednodušší a rychlejší vývoj kvůli podobnosti s objektovým přístupem. MongoDB navíc umožňuje větší flexibilitu struktury dat, takže při budoucím rozvoji datového modelu aplikace se MongoDB databáze přizpůsobí snadněji než databáze postavená na SQL. Díky tomu bývají NoSQL databáze jednodušeji škálovatelné přes více serverů.[40] To ale také znamená, že vývojář nemusí chyby spojené se strukturou dat objevit už při vývoji, ale až při provozu v produkčním prostředí. JSON dokumenty mohou také často vést ke zvýšené duplicitě dat, pokud datový model zahrnuje spoustu propojení mezi entitami. Kolekce JSON dokumentů spoléhají na to, že většina dat spojená s danou entitou bude obsažena právě v daném JSON dokumentu, a proto podporují jen hodně omezené propojování mezi kolekcemi. MongoDB se tak spíše hodí pro aplikace, v níž jsou data hodně unikátní a lze tak využít výhod JSON dokumentů, jako je rychlost načtení nebo škálování.[41]

6.4.2 Databázový systém pro fulltextové vyhledávání

Fulltextové vyhledávání umožňuje vyhledávat entity nejen podle přesně daných klíčových slov, ale hlavně podle frází nacházejících se v dlouhých textech nebo názvech. Většinou se tak zadaná fráze uživatele vyhledává napříč několika různými texty entit, a nalezené výsledky se pak řadí podle relevantnosti, tj. jak moc daná entita odpovídá hledané frázi.

Databáze používané primárně pro vyhledávání dat, a tedy jako spíše sekundární zdroj, nutně nepotřebují pravidelné zálohování dat, protože vyhledávací data lze kdykoliv obnovit z hlavní databáze. Je spíše kladen důraz na rychlost a dostupnost - toho lze docílit duplikací dat na vícero serverů v různých částech světa, čímž se z globálního hlediska dramaticky zvedne dostupnost a zmenší odezva. Jiná je také struktura dat pro optimální vyhledávání jenž nemusí obsahovat nevyhledávaná data a soustředí se spíše na vhodnou agregaci dat.

Ačkoliv většina databázových systémů použitých jako primární zdroj, ať už je to PostgreSQL, MySQL nebo MongoDB, nabízí podporu fulltextového vyhledávání, vlastnost je to spíše doplňková, a oproti dedikovaným databázím pro vyhledávání dosti omezující a pomalá. Specializované databáze navíc poskytují nástroje pro zjednodušení vyhledávání.

Nejpopulárnější takovou databází je Elasticsearch, který je podobně jako MongoDB NoSQL databází využívající JSON dokumenty. Kromě standardního jazykově specifického fulltextového vyhledávání podporuje například i geolokační vyhledávání. [42] Čím dál více používanou alternativou je hostovaná databáze Algolia poskytující velmi podobné vyhledávací funkce jako Elasticsearch, navíc bez nutnosti složité konfigurace a s jednodušším rozhraním pro vývojáře. Bohužel neposkytuje self-hosting, který by umožňoval vývojářům provozovat systém na vlastních serverech, je tak nutné využívat servery přímo Algolie s pravděpodobností vyšších nákladů na provoz a menší kontrolou nad daty a databází. [43] Algolia je i přesto pro běžné fulltextové vyhledávání vhodnější kvůli jednoduššímu rozhraní. Naproti tomu Elasticsearch poskytuje pokročilejší nástroje pro složité analytické vyhledávání a možnost provozovat aplikaci na vlastních serverech.[44] Relativně novou alternativou přicházející s podobnou jednoduchostí jako Algolia je databáze Typesense. Ta je navíc oproti Algolii open-source a lze ji provozovat na vlastních serverech, nicméně momentálně nepodporuje statistiky vyhledávání a personalizaci pro doporučování podobných entit.[45] Problémem u takto nové databáze je pak zejména malá komunita a je pravděpodobné, že při implementaci vyhledávání může vývojář nad touto databází narazit na problém, na který se mu nemusí dostat řešení a bude odkázán na vlastní výzkum a analýzu problému.

6.5 Souborové úložiště

Pro dnešní webové aplikace je běžné uchovávat kromě textových a číselných dat i soubory, jako jsou obrázky, dokumenty nebo videa. Databázové systémy sice umožňují ukládat i soubory v binární podobě, bohužel je ale takové řešení dosti těžkopádné. Při uchovávání souborů se od aplikace očekává, že kromě samotného poskytování originálních souborů uživatelům, bude umět tvořit varianty obrázků, poskytovat duplicitní úložiště napříč celým světem pro rychlé odezvy nazývané CDN (Content delivery network) a zálohování. Zejména pak zmíněné varianty obrázků jsou mnohdy tím stěžejním požadavkem, protože umožňují zobrazovat uživatelům menší komprimované verze originálních souborů a šetří tak objem dat putující mezi serverem a prohlížečem uživatele, rapidně zrychlující načítání webové aplikace.

Souborové úložiště lze implementovat různými způsoby. Kromě již zmíněného databázového systému, lze úložiště implementovat ručně nad vlastním serverem, kde se většinou provozuje samotná webová aplikace. Tento přístup sice vývojáři přináší naprostou kontrolu

nad soubory, musí však řešit implementaci tvořiče variant nebo pravidelného zálohování svépomocí. Musí řešit i napojení na externího poskytovatele CDN systému. Existují však služby poskytující tyto funkcionality s minimální konfigurací zbavující vývojáře nutnosti implementovat a udržovat vlastní systém. Mezi takové služby patří Cloudinary, Sirv, imgix nebo imagekit. Všechny nabízejí základní funkce zahrnující tvorbu variant obrázků a systém CDN s jednoduchým napojením z webové aplikace. Bohužel za cenu jednoduchosti, přijde vývojář o část kontroly nad uživatelskými daty. Avšak oproti budování vlastního úložiště je to pro spoustu firem vítaná varianta umožňující především rychlý vývoj aplikací.

6.6 Technologie pro transakční e-maily a rozesílky

Webová aplikace pracující se zaregistrovanými uživateli potřebuje nějakým způsobem komunikovat se svými uživateli i mimo samotné prostředí webové aplikace. To hlavně když se ve webové aplikaci objeví nějaká nová událost, na kterou je potřeba uživatele upozornit okamžitě bez čekání. Takovou událostí může být cokoliv, zejména upozornění na nové přihlášení z neznámého zařízení, nová aktivita spojená s daným uživatelem nebo upozornění na nové podmínky používání aplikace. Pro tyto a další upozornění se používá převážně emailová komunikace díky její rozšířenosti a univerzálnosti. Kromě upozorňovacích emailů, webové aplikace čím dál častěji využívají hromadné rozesílky novinek, například nových produktů. Bohužel implementace rozesílek je poměrně složitá kvůli velkému množství problémů, které mohou nastat při doručování jednotlivých emailů. Jako příklad lze uvést následující problémy: cílený email může být dočasně zablokovaný nebo poskytovatel dané schránky může zamítnout daný email z podezření na spam. Vlastníci dané webové aplikace navíc potřebují statistiky o úspěšnosti rozesílek, jako je počet úspěšně doručených emailů, počet prokliků konkrétních odkazů a podobně.

Podobně jako u souborového úložiště lze takový systém implementovat svépomocí, to však přináší značnou zátěž na vývoj. Taková zátěž je pak velice znatelná převážně u menších vývojářských týmů. Vznikly proto služby zabývající se poskytováním již hotových služeb s minimální konfigurací. Služeb poskytující tyto funkcionality a mnohé další je mnoho. Některé poskytují pouze základní infrastrukturu pro odesílání emailů, jiné mají nástroje dokonce pro emailový marketing. Mezi nejoblíbenější platformy patří SendinBlue, SendGrid, Mailgun, Mailchimp nebo třeba Sparkpost. Platformy se liší především svými limity v prodávaných balíčcích [46] [47]. Většina z nich zdarma poskytuje omezené balíčky vhodné pro malé webové aplikace, které nepotřebují vysoké limity odesílaných emailů, nebo pro vyzkoušení dané platformy. Při výběru tak hlavně záleží, jaké limity jsou pro projekt stěžejní, případně jestli projekt potřebuje nějaké speciální marketingové nástroje.

6.7 Provoz v produkčním prostředí

Nedílnou součástí vývoje webové aplikace je naplánování strategie provozu v produkčním prostředí. Oproti lokálnímu vývojovému prostředí, vyžaduje produkční prostředí mimo jiné určité požadavky ohledně bezpečnostni a dostupnosti. Nejdůležitější je zabezpečit produkční server proti neoprávněnému přístupu tak, aby na server mohli přistupovat pouze administrátoři. Dalším bezpečnostním prvkem sloužícím zejména pro bezpečí koncových uživatelů je HTTPS (Hypertext Transfer Protocol Secure) protokol zajišťující bezpečnou a šifrovanou komunikaci mezi prohlížečem a serverem. V dnešní době je tento protokol již standardem a prohlížeče na jeho absenci uživatele upozorňují. Chybějící podpora tohoto protokolu může vést k odcizení hesel nebo bankovních údajů, a útočník se tak může vydávat za úplně někoho jiného. Šifrování je zajištěno certifikátem SSL (Secure Sockets Layer) podepsaným

věrohodnou autoritou. Mimo bezpečnost je stěžejní i nepřetržitá dostupnost aplikace. Pokud je webová aplikace často nedostupná pro koncové uživatele, ať už z důvodu zahlcení aplikace požadavky nebo chybě v infrastruktuře, znamená to, že uživatelé mohou přestat aplikaci využívat a přejít ke konkurenci, i v případě, že konkurence nemusí nabízet stejné funkcionality. To má obrovský dopad hlavně na zisky firmy stojící za aplikací, zejména pak pro internetové obchody.

6.7.1 Technologie pro provoz produkčního prostředí

Existuje několik odlišných variant, jakými řešit provoz produkčního prostředí, každé se svými jasnými výhodami i nevýhodami. Nejstarší variantou je pronajmutí serveru. Ten lze pronajmou celý nebo jen jeho část ve formě VPS (Virtual private server). Pronájem celého dedikovaného serveru je mnohdy zbytečné, pokud aplikace neumí využít celý jeho potenciál. Právě díky tomu vznikly zmiňované VPS poskytující jen takový výkon, který aplikace skutečně využije za zlomek ceny. Díky tomu lze navíc levněji poskytovat aplikaci z různých koutů světa bez nutnosti vlastnit několik, ne zcela využitých, serverů. Nicméně takové řešení je velice náročné na konfiguraci a správu. Z tohoto důvodu vyžaduje provoz vlastního serveru mnohdy dedikovaného správce, v případě velkých projektu ne-li celý tým. To si však malé start-upy nemohou většinou dovolit, a i pro velké organizace je to podstatná finanční zátěž.

Postupem času a zkušeností s provozem různých webových řešení vznikly nástroje pro částečnou automatizaci celého procesu, protože podstatná část je ve většině případů shodná. Populárními řešeními pro jednodušší aplikace se stávají předpřipravená prostředí. Ty umožňují při zachování bezpečnosti a dostupnosti provozovat webové aplikace s minimální konfigurací ze strany administrátora.

Pro náročnější webové aplikace vyžadující vlastní konfiguraci existují nástroje, které kombinují prvky obou předchozích řešení. Administrátor se pak stará pouze o infrastrukturu dané aplikace a už neřeší infrastrukturu serverů a úložišť. Mezi takové nejpoužívanější nástroje patří systémy založené na konceptu kontejnerizace aplikací. Kontejner v tomto kontextu představuje spustitelný balíček spojující kód aplikace a všechny její potřebné závislosti, jako jsou knihovny, běhová prostředí, konfigurace prostředí [48]. Tento balíček je možné spustit na jakémkoliv stroji podporující běh kontejnerů bez omezení [48]. Příkladem je orchestrační systém Kubernetes zajišťující nejen běh několika aplikací najednou a jejich vzájemné síťové propojení, ale taky schopnost jednoduše duplicitně provozovat webovou aplikaci na více serverech. Alternativou, také postavenou na kontejnerizaci, je Docker, který tento přístup zpopularizoval. Velmi oblíbený je zejména pro běh aplikací na vývojářských strojích. Avšak za cenu složitější konfigurace ho lze využít i pro provoz produkčního prostředí. Docker je spíše nástroj pro samotnou správu a běh kontejnerů než komplexní správce pro produkční prostředí. Síla také spočívá v univerzálnosti kontejnerů. Ty mohou obsahovat téměř jakoukoli aplikaci, přes webové aplikace, desktopové nástroje až po databázové systémy.

Díky těmto systémům je pak možné poměrně jednoduše spustit webovou aplikaci, API server, hlavní databázi, a navíc ještě třeba databázi pro vyhledávání pomocí pár příkazů v uzavřené lokální síti s pevně definovanými pravidly síťového provozu.

I přes správně nakonfigurované prostředí však může dojít k chybě v samotné aplikaci, a ta se může stát nedostupnou nehledě na to, že samotná infrastruktura funguje v pořádku. Pro tyto případy je dobré mít nějaký monitorovací systém sledující dostupnost aplikace. K tomu je většinou využíván strojově ovládaný prohlížeč simulující základní uživatelské kroky

při vstupu na stránku aplikace. Kromě jednoduchého monitorování dostupnosti, umožňují některé nástroje i monitoring rychlosti načtení stránky pro průběžnou kontrolu, že se v aplikaci neděje něco neobvyklého.

6.7.2 Poskytovatelé prostředí pro produkční provoz

Z důvodu velké náročnosti provozu vlastního fyzického serveru ve vlastní serverovně využívají jednotlivci i menší firmy serverů ostatních společností zabývajících se čistě provozem serverů k pronájmu. Takové společnosti se nazývají poskytovatelé. Koncový administrátor se pak už nemusí starat o infrastrukturu spojenou s propojením serverů k internetu.

Mezi ty nejznámější patří: DigitalOcean, Microsoft Azure, Google Cloud Platform, Amazon Web Services, Linode nebo třeba Vultr. Samozřejmostí je poskytování základních služeb v podobě různých variant VPS na různě výkonných serverech. Lze si ale pronajmout již předkonfigurovaný server přímo pro konkrétní typ aplikace, kdy stačí navést poskytovateli ke zdrojovému kódu, vše ostatní připraví on. Tato varianta je nejjednodušší a ideální pro aplikace nemající specifika oproti standardizovanému postupu pro vybraný programovací jazyk. Není tak ani potřeba mít v týmu administrátora starajícího se o provoz produkčního prostředí. Pro složitější aplikace s proprietární konfigurací nebo více dílčími aplikacemi si lze často pronajmout předkonfigurovanou strukturu pro běh systému Kubernetes. Ani v tomto případě většinou není potřeba speciálního administrátora, protože je Kubernetes primárně navržen pro programátory. Jednotliví poskytovatelé se pak hlavně liší v cenách za pronájem, kvalitou podpory, dostupným hardwarem, předkonfigurovaných prostředích a v garantované dostupnosti. Záleží tedy na požadavcích a rozpočtu projektu.

Poskytovatelé samotného prostředí sice většinou umožňují monitorovat aplikaci a samotný server zevnitř serveru, pro monitoring aplikace zvenčí však existují externí služby zabývající se pouze monitoringem, díky tomu nabízí pokročilejší nástroje. Takových služeb je nepřeberné množství a většina z nich poskytuje téměř stejné funkce. Pravděpodobně nejznámější službou je Pingdom umožňující monitorovat webové stránky z desítek lokalit po celém světě [49]. Přichází nicméně pouze s placenými plány, které jsou pro malé a neprofitující aplikace poměrně drahé. Naštěstí existují bezplatné omezenější služby poskytující alespoň základní monitoring dostatečný pro pravidelné ověřování běhu aplikace, i s ověřováním správnosti odpovědí serverů za cenu méně častého provolávání. Některé služby dokonce zdarma poskytují i pokročilejší funkce, tím je většinou monitoring platností SSL certifikátů. Jako příklad lze uvést Better Uptime, UptimeRobot, HetrixTools nebo Checkly. Samozřejmostí jsou pak i placené plány s více funkcemi, takže lze podle potřeby funkce dodatečně rozšířit.

7 Implementace

Pro samotnou implementaci vlastního navrhovaného řešení je potřeba vybrat správné technologie podle definovaných požadavků popsaného řešení. Je dobré se zamyslet nad výběrem technologií z každé kategorie hned na začátku a promyslet, zda jsou všechny vybrané technologie mezi sebou kompatibilní, případně zdali je daná kombinace optimální pro implementovaný projekt. Při výběru jen části technologií by se mohlo v průběhu vývoje stát, že nějaká nově vybraná technologie nemusí být snadno zakomponovatelná do již existujícího kódu. To by mohlo vést k nutnosti předělat část aplikace. Po výběru technologií je vhodné navrhované řešení, pro ustanovení finální podoby a rozvržení struktury aplikace, nejdříve načrtnout v nějakém grafickém editoru. Následně je již možné začít se samotnou implementací s pomocí vybraných technologií a prototypu UI.

7.1 Pojmy

Je dobré si stanovit a vysvětlit určité běžně používané technické pojmy, které nemusí být úplně jednoznačné na první pohled:

Doména Doména představuje objekty v systému, které zastupují reálný řešený problém.

Business logika Tento pojem označuje hlavní část zdrojového kódu zabývající se naplněním stanovených funkčních požadavků aplikace. Běžně taková logika řeší manipulaci dat z nějakého úložiště. Může ale vykonávat i složité výpočetní operace. Do této části se nevztahuje to, jak jsou data dotažena z databáze nebo v ní uložena, neřeší komunikaci s okolím, nebo sestavení stránek z poskytnutých dat a podobně.

7.2 Návrh implementace

Způsobů, jak implementovat webovou aplikaci, je mnoho, proto může být o to těžší pro nezkušeného programátora vybrat ten vhodný. Není totiž vždy úplně jednoznačné, jakou cestou se vydat, protože pro většinu běžných webových aplikací je více vhodných způsobů. Je tak na zvážení implementátora, jaký způsob je vhodnější pro daný projekt. Nejpoužívanějšími jsou následující dva přístupy: pomocí serverové aplikace a kombinací klientské a serverové aplikace.

Serverová aplikace poskytující sestavené stránky prohlížeči je starším a do nedávna nejvíce obecně používaným způsobem. Taková aplikace sestavuje stránky z programátorem připravených šablon jednotlivých stránek, do kterých se při sestavení vloží na určená místa aktuální a požadované informace. Těmi může být téměř cokoli: od jednoduchého údaje, jako je jméno přihlášeného uživatele, přes celý seznam článků k přečtení až po kompletně dynamické sestavení dané stránky, třeba na základě dat z databáze. Pro tvorbu takových šablon existuje velké množství specializovaných scriptovacích jazyků. Sestavená stránka poslaná prohlížeči je nicméně víceméně statická, a i když lze takové stránky pro dodatečnou dynamičnost v uživatelově prohlížeči obohatit například jazykem JS, s rostoucí dynamičností roste komplexnost a nepřehlednost. Kvůli této nevýhodě tento způsob společně s nutností tvořit pro každou stránku novou šablonu upadá díky lepším a novějším alternativám. Nástroje pro tvorbu šablon sice do jisté míry umožňují přepoužívání částí stránek bez nutnosti duplicity, nicméně neumožňují velkou míru dynamičnosti po sestavení výsledné stránky. Výhodou je ale přímé propojení s logikou aplikace a díky tomu i snazší přístup k

samotným datům. Pro jednodušší, a hlavně méně dynamické stránky v podobě například blogů, může být tento způsob díky jeho relativní jednoduchosti stále vyhovující. Pro vetší webové stránky, a především webové aplikace, je v dnešní době tento způsob již zbytečně nepřehledný a složitý, a postrádá dynamičnost nativní aplikace.

Novějším a stále více používaným a oblíbeným způsobem je sestavování stránek pomocí dynamických a znovu použitelných komponent prvků s pomocí jazyka JS. Každá komponenta reprezentuje jeden prvek (např.: tlačítko nebo kontejner), který má svoji šablonu pro vykreslení a svůj stav zajišťující dynamičnost prvku v čase nebo při interakci uživatele. Šablonovací nástroje pro tvorbu komponent jsou velmi podobné těm z předchozího způsobu, nicméně hluboce integrují jazyk JS pro propojení vzhledu a stavu komponent. Stav takové komponenty může obsahovat jednoduchý příznak, kdy bylo tlačítko stisknuto, nebo komplexnější data, jako je uživatelem vložený text a jeho formátování. Díky zapouzdření stavu a šablony je možné celkem jednoduše takové komponenty používat na více místech bez nutnosti přidávat do rodičovské stránky logiku pro ovládání daného prvku. Pro zajištění všech těchto vlastností se používá jazyk JS běžící v prohlížeči, kde se běžně děje veškeré sestavování. Nicméně prohlížeč v tomto případě nemá přístup k datům, a je proto nutné zajistit ještě nějaký zdroj dat a komunikaci mezi tímto zdrojem a aplikací v prohlížeči. Tím je většinou serverová aplikace poskytující API zpřístupňující data pomocí standardizované strojové komunikace. API je obecný název pro rozhraní poskytující více aplikacím možnost strojově komunikovat mezi sebou pomocí strukturovaných dat. Aplikace vystavující API definuje jaká data a funkce umí zpracovat a ostatní aplikace při dodržení těchto pravidel mohou získávat z této služby data nebo je naopak odesílat ke zpracování. Momentálně nejpoužívanějšími přístupy jsou REST API a GraphQL API, každý má své výhody, nevýhody a myšlenku, jak má takové API vypadat. Nicméně oba přístupy slouží ke stejnému účelu: poskytovat data frond-endovým aplikacím ze serveru. I tak je oproti předchozímu způsobu, v němž mají šablony přímí přístup k aktuálním datům, složitější správně synchronizovat informace mezi prohlížečem a serverem, a to právě z důvodu separace, protože front-endová aplikace musí data hlídat a sama aktualizovat. Především je tato nevýhoda viditelná při implementaci systémů uživatelů a přihlášení. Jak už se může na první pohled zdát, tento způsob tvorby aplikace je složitější, protože vyžaduje znalosti více technologií najednou a schopnosti všechny tyto komponenty správně propojit do fungujícího celku. Další nevýhodou jsou zvýšené nároky na výkon cílového zařízení uživatele kvůli zmíněné dynamičnosti probíhající přímo v uživatelově prohlížeči. Z pohledu vyhledávačů je zde nevýhoda omezených možností nastavení SEO (Search engine optimization) vlastností z důvodu delegovaného sestavování stránek na prohlížeč. S tím má kvůli snížené podpoře JS většina botů prohledávající weby problém. Naštěstí zvýšenou náročnost převyšují výhody v podobě velké dynamičnosti, přehlednosti a znovupoužitelnost. Navíc v současnosti existují i nástroje sestavující prvotní podobu stránek již na serveru, a díky tomu lze poskytnout sestavenou stránku uživateli dříve a se správným nastavením pro SEO.

I když by se tedy daly použít oba způsoby, bylo vybráno řešení využívající dynamické komponenty, zejména z důvodu požadované velké dynamičnosti aplikace. Tento způsob navíc zjednodušuje budoucí transformaci webové aplikace na téměř nativní mobilní aplikaci.

Původní prototyp aplikace byl sice implementován pomocí prvního zmíněného způsobu za pomoci šablon stránek, jak se ale ukázalo krátce po začátku vývoje, by byl tento způsob při snaze vyvinout moderní dynamickou aplikaci spíše limitující. Bylo proto rozhodnuto pojmout výzkum dostupných a vhodných technologií sofistikovaněji a začít víceméně od znovu. Znamenalo by to také vyvinout vlastní ekosystém zbytečně oddělený od samotných stránek, což dále vede k nepřehlednosti a horšímu rozšiřování. Kromě jasných výhod pro

tento konkrétní projekt, je zde ještě vedlejší benefit v podobě zkušeností získaných z návrhu komplexnějšího systému, v němž jednotlivé části mezi sebou musejí korektně komunikovat.

Pro vybraný způsob je zásadní vybrat i způsob komunikace. Pomineme-li kompletně vlastní řešení, které by bylo pracné na vyvinutí a pro tento projekt by nepřineslo žádné benefity, je na výběr mezi vzory SOAP (Simple Object Access Protocol), REST nebo GraphQL API. Vzor SOAP je již poměrně starý, a i když nabízí rozsáhlé možnosti, jeho struktura založená na značkovacím jazyku XML je zbytečně složitá a příliš upovídaná, což může zbytečně zatěžovat a zpomalovat síťovou komunikaci. Z tohoto důvodu již nějakou dobu výměně strukturovaných dat dominuje jazyk JSON, který je značně stručnější, čitelnější a hlavně jednodušší na zpracování. Na tomto jazyku staví právě vzory REST a GraphQL. REST staví na principu poskytování jednotlivých ucelených dat v předem specifikované formě a objemu v podobě jednotlivých zdrojů [50]. Ty lze buď získat nebo zapsat, a to vše bez návaznosti mezi jednotlivými požadavky [50]. GraphQL je novější a oproti REST postaven na myšlence získání jen těch dat, které aplikace opravdu potřebuje [51]. K tomu mu slouží speciální dotazovací jazyk [51]. Stejně jako REST má pevně danou strukturu dat, ta ale spíše udává jaká všechna data lze získat a jak budou vypadat [51]. GraphQL pomocí toho řeší problém se získáváním příliš mnoho dat nebo nedostatek dat, což je nevýhoda REST API. Nelze ale říci, že GraphQL je za všech podmínek lepší než REST, protože každé API musí z pohledu serveru řešit stejné problémy, jako jsou serializace dat, zpracování chyb, cachování a mnoho dalšího. GraphQL tak sice poskytuje lepší kontrolu nad vracenými daty, nemusí však být vhodné pro některé projekty (např.: obecné mikroslužby poskytující API pro kohokoliv). Navíc klient konzumující takové GraphQL API potřebuje komplexnější knihovnu pro zpracování API. Kromě toho REST za svou delší existenci vyspěl a existuje nepřeberné množství dostupných materiálů k tomu, jak navrhnout správné API, a nástrojů pro samotnou tvorbu.

Právě kvůli vyspělosti REST API a rozšířenosti padlo rozhodnutí právě na tento přístup. Vzor REST je navíc stále velmi populární a používaný, a tak hlubší znalosti návrhu API pomocí tohoto vzoru nejsou k zahození. Obecně jsou však pro tento projekt vhodné oba přístupy a ani jeden by nekomplikoval vývoj.

7.3 Výběr technologií

Po rozhodnutí, jakým způsobem bude projekt fungovat, je možné vybrat konkrétní technologie, pomocí kterých bude aplikace vyvíjena. Je patrné, že vybrat správnou technologii pro daný projekt není snadný úkol, především pokud vývojář nemá ani okrajové zkušenosti s většinou z těchto technologií a nemůže tedy vybírat podle vlastních poznatků. Pokud jsou ale vybrány celosvětově používané a podporované technologie, nelze většinou udělat chybu výběrem ani jedné z nich, protože každá je většinou dostatečně univerzální. Krajní výjimkou můžou být specifické projekty s náročnými požadavky, to ovšem není případ tohoto projektu. Pro běžnější projekty spočívá hlavní rozdíl ve stylu, jakým se v dané technologii projekt vyvíjí, a v tom, jestli daný vývojář má alespoň nějaké znalosti používaného programovacího či skriptovacího jazyka. Čas strávený studováním nových technologií může být v porovnání s již nastudovanými technologiemi poměrně velkou částí samotného vývoje.

7.3.1 Kritéria pro výběr technologií

Jak již bylo zmíněno, vyvíjený projekt bude webovou aplikací potencionálně dostupnou ve všech moderních prohlížečích. Stěžejním kritériem je vybraný přístup s využitím dynamických komponent, a tudíž bude aplikace rozdělena na front-endovou aplikaci a back-endový

API server. Důležité je omezení pouze na moderní prohlížeče, protože pokud by byl požadavek na podporu i několik let starých, již nepodporovaných prohlížečů používaných jen malým procentem uživatelů, možnosti výběru technologií by to značně omezilo. Nebylo by možné použít novější technologie značně usnadňující vývoj a bylo by nutné dělat mnohem více kompromisů, především při samotném vývoji.

Dalším velmi důležitým aspektem pro aplikaci zveřejňující spoustu vyhledatelných informací je možnost nastavit kvalitní SEO, aby se daná aplikace dostala do předních žebříčků při vyhledávání skrze internetové vyhledávače. SEO je technika pro optimalizaci webových stránek, tak aby byly vyhledávače schopné co nejlépe pochopit a kategorizovat informace poskytnuté danou stránkou. [52]

7.3.2 Vybrané technologie

Návrh UI a UX Prvním nástrojem při návrhu aplikace by měl být grafický editor pro náčrt struktury a vzhledu. Vzhledem k tomu, že všechny dostupné nástroje obsahují podobné základní funkce, byl vybrán editor Figma díky schopnosti běhu v prohlížeči, a tím odpadající nutnosti vlastnit zařízení s operačním systémem Windows nebo macOS. Program je navíc bezplatný i pro základní kolaboraci mezi členy v týmu.

Technologie pro vývoj uživatelského prostředí Vzhledem k tomu, že cílem je vytvořit webovou aplikaci, která poběží v standardních webových prohlížečích, nelze se úplně vyhnout základním front-endovým technologiím HTML, CSS a JS. Ty lze ale do jisté míry nahradit jejich nadstavbami.

V případě CSS byl vybrán preprocesor Sass, pro jeho širokou komunitu a funkce. Tento preprocesor umožní přehlednější kód a tím rychlejší vývoj. Kvůli výše zmíněné potřebě zobrazovat různé typy odkazů na kartách, Sass umožní do jisté míry automatizovat repetitivní a nepřehledné CSS třídy pro velké množství barevných kombinací (každá sociální síť používá jiné barevné kombinace ve svém designu).

V případě tvorby interaktivního GUI je výběr složitější. Je možno využít jak základního JS, tak frameworků či knihoven. Alternativou může být jazyk C# s technologií Blazor, díky které je možné se úplně od ekosystému JS odstínit. Ze studijních účelů byl vybrán jazyk JS, především pro hlubší pochopení tohoto jazyka, neboť je používán v současné době téměř každou webovou stránkou či aplikací. Jako nadstavba jazyka pro zrychlení vývoje byl vybrán framework Vue díky jeho velmi dobré dokumentaci, snazšího pochopení a možnosti programovat přímo v jazyku JS. Vue (a stejně tak ostatní takové nadstavby) sám o sobě ale nedisponuje zrovna dobrými prostředky pro SEO. Tuto techniku poskytují serverové frameworky Nuxt.js nebo Next.js. Protože byl ale vybrán Vue, lze vybrat pouze Nuxt.js. Kromě lepších nástrojů pro SEO, sestavení stránky na serveru zrychlí prvotní vykreslení stránky pro koncové uživatele.

Technologie pro vývoj API serveru Co se týče technologií pro vývoj serverové aplikace fungující jako API poskytující data, je výběr spíše otázka preferencí a předchozích znalostí vývojáře. Všechny zmiňované totiž nabízí alespoň základní nástroje pro tvorbu REST API. Z důvodu že byl vybrán serverový framework Nuxt.js pro podporu SEO, by bylo možné jednoduše použít Node.js, na kterém Nuxt.js běží v kombinaci s frameworkem Express.js, a vystavit tak REST API přímo pomocí něj. Ekvivalentní alternativou je použití jazyka Java, C# nebo PHP. Ze studijních důvodů a vývojářovy předchozí znalosti byl

vybrán právě jazyk Java společně s frameworkem Spring. Kromě podpory pro REST API, ekosystém Spring přichází například s vlastní knihovnou pro zabezpečení dat.

Databázový systém jako zdroj dat Pro uchovávání uživatelských dat je nutné vybrat vhodný databázový systém. Pro účely tohoto projektu by se daly použít jak SQL databáze tak i NoSQL databáze. Projekt nevyžaduje specifické náročné analytické agregace dat, dalo by se tak uvažovat o NoSQL databázi MongoDB pro ulehčení mapovaní objektů na JSON dokumenty. Nicméně navrhované řešení počítá s určitou hierarchií karet, což už by v případě MongoDB mohl být značný problém [41]. Projekt sice bude zprvu vyžadovat pouze jednoúrovňovou hierarchii, i tak by ale docházelo ke zbytečné duplicitě dat. Z těchto důvodů padla volba na některou z SQL databází. Z bezplatných databází by se dala vybrat například PostgreSQL, MySQL nebo MariaDB, a všechny by byly schopny poskytnou potřebné funkce pro vývoj navrhovaného řešení. Proto finální výběr byl tak spíše preferencí databázového systému PostgreSQL.

Databázový systém pro fulltextové vyhledávání U systému pro vyhledávání dat je rozhodování celkem jednoduché, avšak jedná se o preferenci, protože požadavky tohoto projektu splňují všechny zmíněné databáze. Z výše zmíněných systémů byl vybrán Elasticsearch kvůli rozsáhlé komunitě a možnosti provozovat jej svépomocí.

Souborové úložiště Dalším důležitým úložištěm dat je úložiště uživatelských souborů, v tomto případě se jedná především o obrázky. Implementovat vlastní řešení by bylo poměrně pracné a i přes velký investovaný čas, by nemuselo poskytovat všechny funkce co ostatní řešení. Vhodné řešení by mělo poskytovat kromě CDN hlavně tvořič variant. Takových služeb je vícero a mnohdy jsou svými funkcemi ekvivalentní, výběr je tedy opět spíše preferencí. Vybrána byla bezplatná služba Cloudinary především díky její rozšířenosti mezi ostatními uživateli.

Transakční emaily a rozesílky Protože aplikace bude pracovat se zaregistrovanými uživateli, bude potřeba uživatelům odesílat transakční emaily o důležitých událostech spojených se změnami v účtech, případně o jiných důležitých událostech, jako je třeba změna podmínek aplikace.

Opět jako v případě souborového úložiště lze připravit vlastní implementaci zpracovávání šablon emailů a jejich odesíláním pomocí knihovny JavaMail, která se stará pouze o odeslání emailů. Avšak u emailů je nutnost řešit i šablony (podobně jako u šablon stránek) a doručitelnost. To už by zabralo nemalou část vývojového času.

Většina nástrojů zmíněných v analýze poskytuje poměrně jednoduché API pro odesílání emailů s již připravenou infrastrukturou pro tvorbu šablon a jednoduchou možnost nastavit odesílání emailů, tak, aby byly správně doručeny. Na druhou stranu bezplatné plány jsou omezené na počet možných odeslaných emailů za nějakou dobu, a za větší limity se musí platit. Avšak tato řešení přináší záruku určité spolehlivosti doručitelnosti emailů díky serverům s dobrou pověstí u emailových serverů, které jen tak daný email nezahodí do nevyžádané pošty. V případě této aplikace je kladen důraz především na jednoduchost tvorby šablon a odesílání transakčních emailů. Z toho důvodu byla vybrána služba SendinBlue poskytující intuitivní editor šablon emailů a jednoduché API pro odesílání emailů s parametry.

Provoz aplikace Poslední částí skládanky je samotný provoz v několika prostředích.

Jak již bylo v analýze nástrojů a poskytovatelů zmíněno, možností jak webovou aplikaci provozovat je mnoho: od manuálního nastavování a spouštění až kompletní automatizaci. Protože se jedná o nový projekt nemá moc smysl se vydávat starou metodou manuálního nastavování všeho. Místo toho se pro tento projekt vyplatí jít cestou kontejnerizace, protože umožňuje jednoduché nahození a propojení všech dílčích aplikací, na kterých bude aplikace stát. Bonusem je zjednodušení celého procesu sestavovaní aplikace ze zdrojového kódu a přenos na cílové prostředí. Nevýhodou může být nutnost znalosti dalšího nástroje pro sestavování takových kontejnerů a jejich následné spuštění, u většiny projektů je ale tato nevýhoda mnohonásobně překročena zrychlením a zjednodušením vývoje.

Výběrem systému Kubernetes pro provoz se výběr možných poskytovatelů zužuje. Sice v dnešní době již většina poskytovatelů nabízí servery Kubernetes, nicméně často se liší cenou daného řešení. Mezi dlouhodobě nejlevnější patří poskytovatelé DigitalOcean a Linode. Oba poskytují široké možnosti konfigurace, přívětivou cenu, rozsáhlou dokumentaci, základní monitoring hardwaru a ochotnou podporu. Z vlastních zkušeností byla dána přednost DigitalOceanu díky jednodušší konfiguraci celého Kubernetes systému, zejména pak při nastavování bezpečnostních funkcí. DigitalOcean je navíc jeden z nejpopulárnějších poskytovatelů, a tak kromě oficiální dokumentaci a podpory lze najít spoustu komunitních materiálů.

Finálním nástrojem je externí monitorovací systém. Pro tento konkrétní projekt byla vybrána služba Checkly díky poměrně široké nabídce funkcí, zejména pak možnost monitorovat dotazy na webové stránky, dotazy na API, expiraci SSL certifikátu a možnost napojení na komunikační aplikace (např.: Discord).

7.4 Prototyp

Prvotní verze aplikace byla postavena na frameworku Spring společně ORM (Object-relational mapping) frameworkem Hibernate pro přístup k databázovému systému. Tato verze stavěla na starším způsobu tvorby webových aplikací se sestavováním jednotlivých stránek již na serveru pomocí šablon. Neexistovalo tak rozdělení na front-endovou aplikaci a API server. Aplikace však zahrnovala střípky REST API se snahou udělat GUI více dynamické. Aplikace tak měla dvě části: hlavní, v níž byly šablony a business logika hluboce propojeny, a vedlejší poskytující část dat pomocí API. Nicméně i přes to, že se jednalo o rannou fázi aplikace s minimem funkcí, začalo být postupně znát, že vytvořit plně dynamickou aplikaci chovající se podobně jako nativní aplikace, by bylo při nejmenším kostrbaté. Pro rozšíření API by se musela business logika přizpůsobit a být více modulární. Šablony stránek byly tvořeny bez jakéhokoliv vzhledu a UX návrhu, a obsahovaly pouze nejnutnější prvky pro interakci. To však pro aplikaci vyžadující intuitivní UI není vhodné a mohlo by se stát, že při implementaci finálního vzhledu by se podstatná část stránek mohla zásadně změnit, což by vyžadovalo velký zásah do již existujícího kódu.

Právě kvůli těmto obavám a možnostem moderních technologií byl kompletně změněn přístup k vývoji, a bylo potřeba začít od nuly s analýzou technologií a možností. Část původního kódu se sice přepoužila, nicméně původní kód byl po analýze natolik přepracován, že se dá uvažovat spíše o nové aplikaci.

7.5 Uživatelské prostředí

Zejména pro explicitní definování funkčních požadavků a způsobu interakce uživatele s aplikací, byl před samotnou implementací věnován čas návrhu UI a UX. Byl tedy připraven kompletní grafický návrh prvků a stránek webové aplikace. Díky tomuto návrhu bylo možné

postupně vypilovat grafické prvky, styly a strukturu. Bylo možné lépe nahlédnout na designový jazyk použitý v různých scénářích a upravit ho tak, aby byl dostatečně univerzální.

Designový jazyk je soubor pravidel, schémat a stylů grafického designu, kterým se řídí vývoj veškerých prvků UI. Tento jazyk má zajistit jednotnost designu napříč celou aplikací pro poskytnutí intuitivnějšího rozhraní a zamezení frustrace uživatelů. Z tohoto důvodu je nutné věnovat návrhu designového jazyka nějaký čas a zamýšlet se nad všemi možnými scénáři.

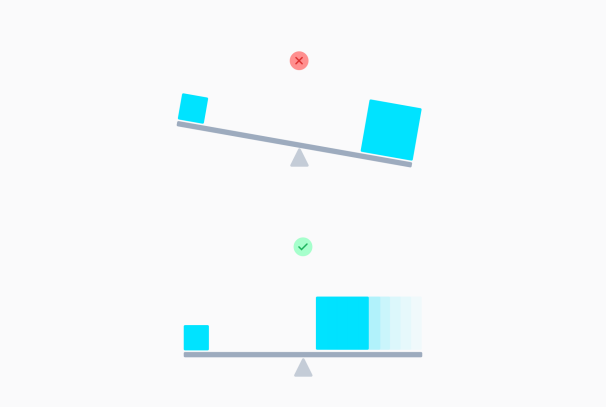
Design jazyk si bere velkou inspiraci z Material Design 2, jenž definuje jakýsi soubor základních rozvržení, velikostí prvků, barev a podobně. Navržený design jazyk se však v některých prvcích rozchází, a není proto pravidlem, že Material Design 2 striktně kopíruje. I tvůrci Material Designu berou tyto materiály jako výchozí bod, a počítají s tím, že daný projekt si ho přizpůsobí, tak, aby co nejlépe vystihoval danou aplikaci. Z Material Designu si bere hlavně velikosti mezer a prvků a základní pravidla, jako je pozicování tlačítek, rozvržení seznamů, modální okna a podobně.

7.5.1 Rozvržení prvků a grafická pravidla

Rozvržení prvků by mělo být maximálně přehledné a intuitivní. Mezi hlavní pravidla patří nezatěžování stránek příliš mnoha informacemi, které mnohdy nejsou potřebné. Pokud je nutnost zobrazit velké množství informací, je lepší rozdělit jednu stránku na více přehlednějších.

Dalším podstatným pravidlem je zakomponování tzv. bílého prostoru kolem prvků. Bílý prostor je prostor kolem prvků jenž odděluje jednotlivé prvky prázdnotou. Prvky tak nejsou na sobě tak nalepené a uživatel se dokáže lépe orientovat. Neméně důležité je vizuálně navést uživatele, co na dané stránce může vše provádět. Díky tomu je pravděpodobnější, že uživatel dosáhne svého cíle bez frustrace, opuštění aplikace nebo vyhledání pomoci. Pro spokojenost uživatele je dále nutné udržovat konzistenci grafických pravidel a prvků napříč celou aplikací, aby uživatele jen tak něco nepřekvapilo. [53]

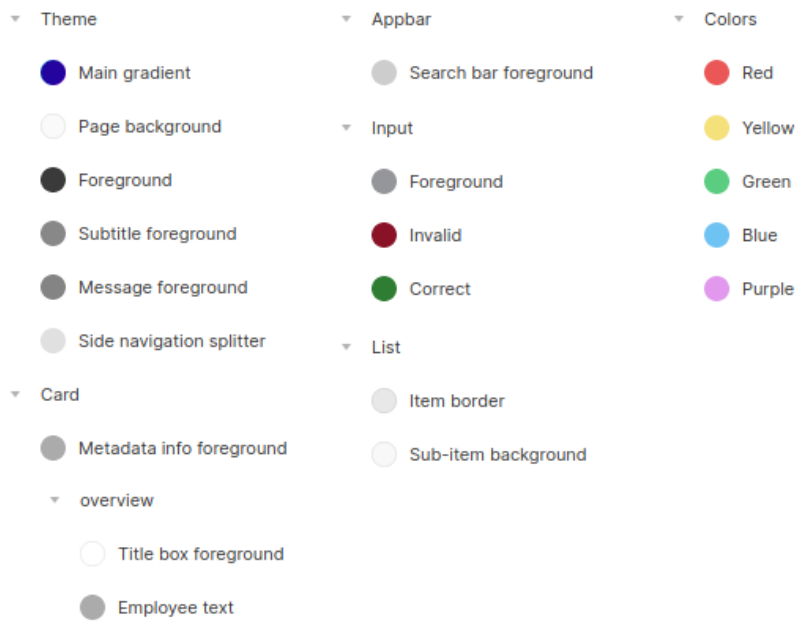
Užitečnou pomůckou pro návrh rozvržení prvků je princip vizuální rovnováhy. Vizuální rovnováha staví na pomyslné vizuální váze prvků, v níž určité vlastnosti, jako je kontrast nebo složitost, váhu navyšují. Následně dva prvky porovnává na páce, která musí zůstat v rovině. V případě, že jsou prvky jinak těžké se využívá odsazení prvků od sebe (bílý prostor). [54]



Obrázek 1: Páka rovnováhy vzhledu. Zdroj: [54]

Rozvržení prvků ale není vše, a je potřeba myslet i na samotný design.

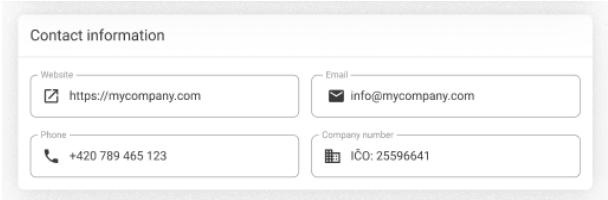
Paleta barev Důležitou částí designu je správně připravená paleta barev, jenž bude použita napříč aplikací. Ta definuje jak obecné barvy prvků, jako jsou texty, bloky, tlačítka, tak i ty méně obecné.



Obrázek 2: Navržená paleta barev. Zdroj: [autor]

Stíny a linky Dalším velmi důležitým designovým prvkem jsou stíny pomáhající vizualizovat pomyslnou vzdálenost jednotlivých prvků od uživatelova zraku. Díky stínům je možné uživateli ilustrovat, které prvky jsou více v popředí než ostatní. Způsobů používání stínů je mnoho. Pro tento projekt byl vybrán způsob, kdy se stín snaží simulovat reální stín generovaný světelným zdrojem. Na takové stíny jsou totiž uživatelé zvyklí z reálného světa. Velmi používané je i využití linek kolem jednotlivých prvků pro oddělení částí bloků.

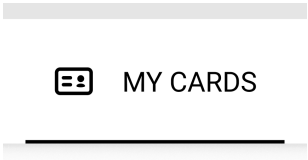
Tento projekt využívá kombinace linek a stínů. Stíny jsou použity pro bloky ucelených informací a prvky, které mají vyšší prioritu než ostatní, a linky jsou použity pro prvky uvnitř bloků. Kombinace byla vybrána z důvodu zjednodušení vzhledu, protože použití pouze stínů by mohlo být pro některé uživatele příliš rozptylující.



Obrázek 3: Ukázka kombinace stínů a linek v jednom bloku. Zdroj: [autor]

Okraje Kromě linek a stínů je dobré myslet i na samotné okraje prvků, tedy jestli budou mít zaoblené rohy či nikoliv. Po vzoru Material Designu byly zvoleny zaoblené rohy o průměru 8px pro větší lehkost prvků. Toto rozhodnutí je však celkem subjektivní a někteří uživatelé mohou preferovat ostré rohy.

Signalizace výběru Posledním pravidlem je zobrazování signalizace výběru. Tato část musí být pro koncové uživatele obzvlášť konzistentní a intuitivní napříč aplikací. Uživatel musí být schopen jednoznačně určit jaký prvek ze všech možných je vybraný. Velice používaným způsobem jak tuto signalizaci řešit je jednoduchá linka zobrazená u vybraného prvku. V rámci této aplikace byly navrženy dvě varianty: varianta pro pás prvků a varianta pro osamocené prvky. Varianta pro pás prvků se použije v případě pásu prvků, u kterých nemůže dojít k přetečení na více řádků. V takovém případě se použije jednoduchá linka pod či nad prvkem. V případě osamocených prvků nebo víceřádkových prvků je použita linka kolem celého okraje prvku.



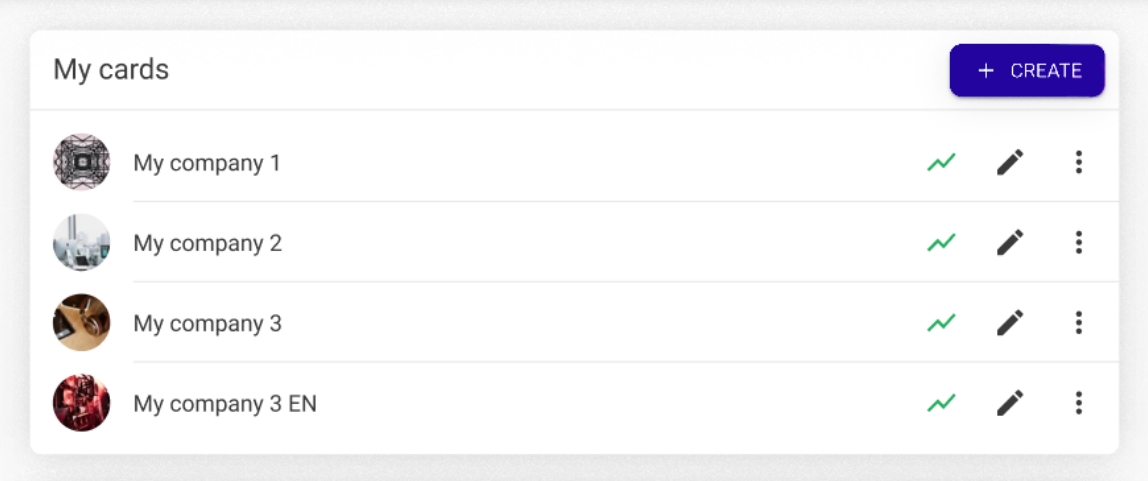
Obrázek 4: Ukázka signalizace výběru v navigaci. Zdroj: [autor]

Obrázek 5: Ukázka signalizace výběru konkrétního prvku. Zdroj: [autor]

7.5.2 Strukturální prvky

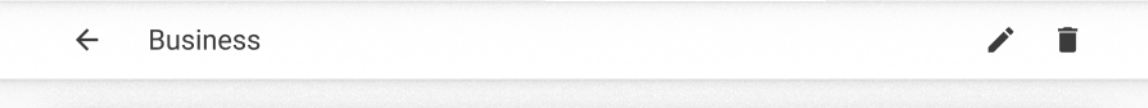
Strukturálními prvky jsou myšlené takové prvky, které slouží jako základní stavební kameny pro jednotné rozvržení stránek.

Blok obsahu Stránky jsou rozvrženy hlavně pomocí tzv. bloků obsahu označující jeden logický celek informací na stránce. Blok obsahu vždy obaluje nějaký konkrétní obsah: dlouhý text, seznam položek, editor nebo cokoliv dalšího. Blok obsahuje hlavičku s nadpisem a lištou tlačítek, která dávají smysl pro celý logický celek. Samotný obsah těla už je pak záležitostí konkrétního využití bloku, avšak pravidlem je jednotné odsazení 24px od okrajů. Krajním případem je využití bloku bez hlavičky na speciálních stránkách s vlastním nadpisem, u nichž blok představuje logický celek informací platných pro celou stránku.



Obrázek 6: Ukázka využití bloku obsahu. Zdroj: [autor]

Lišta akcí Kromě hlavní navigační lišty aplikace bylo nutné zavést ještě jeden typ lišty: lištu akcí. Díky ní je možné na určitých stránkách zobrazit sekundární lištu s nadpisem a dodatečnými akcemi, jenž se stahují k obsahu celé stránky. Takovými stránkami jsou většinou stránky zobrazující detail určitého objektu, se kterým lze manipulovat.

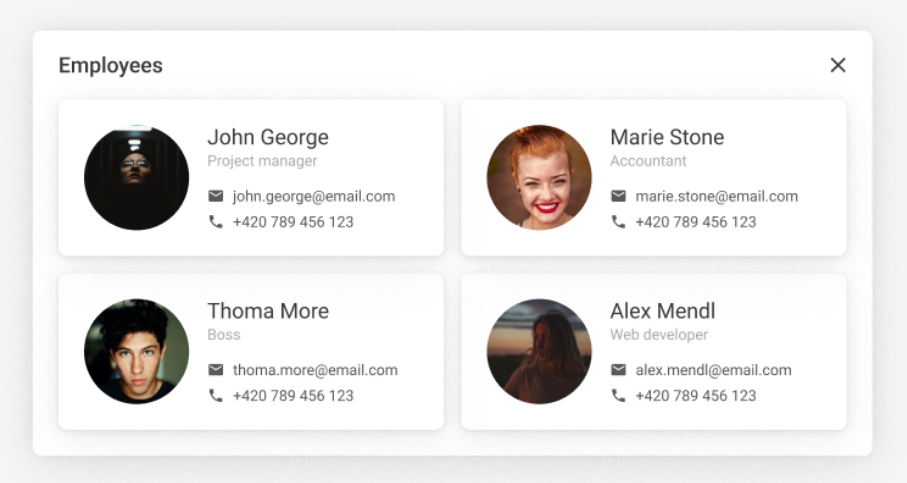


Obrázek 7: Ukázka lišty akcí. Zdroj: [autor]

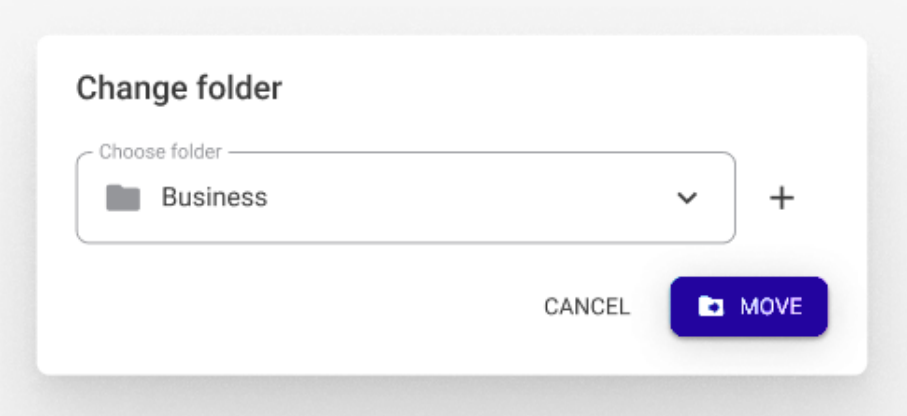
Modální okno Modální okna představují bloky obsahu, které pro zdůraznění důležitosti při zobrazení překrývají všechen ostatní obsah. Jedná se především o potvrzení určitých akcí nebo zobrazení detailu nějakého objektu. Každé okno má hlavičku, a případně i patičku, avšak struktura se mění podle typu okna. Tato aplikace rozlišuje dva primární typy oken: informační a akční okno. Informační okno slouží k zobrazení již existujících dat a nevyžaduje od uživatele žádnou akci. Lze ho zavřít buď křížkem v hlavičce nebo kliknutím mimo okno, a neobsahuje patičku. Akční okno akci vyžaduje a uživatel musí vždy nějakou akci zvolit,

nelze okno zavřít pouhým kliknutím mimo okno. Takové okno má v hlavičce pouze nadpis a obsahuje patičku se všemi dostupnými akcemi. Příkladem takových akcí může být potvrzení vytvoření objektu nebo zrušení operace. Tlačítko nejdůležitější akce je patřičně odlišeno od běžných tlačítek, aby bylo zřejmé, co se po uživateli chce.

Okna mají podobná pravidla odsazení jednotlivých vnitřních prvků jako bloky obsahu. Obsah by měl být odsazen 24px od okrajů okna (až na výjimky s vlastním odsazením) a akční tlačítka jsou mezi sebou odsazena 8px. Celá patička je navíc pro jasné rozdělení částí rovněž odsazena 24px.

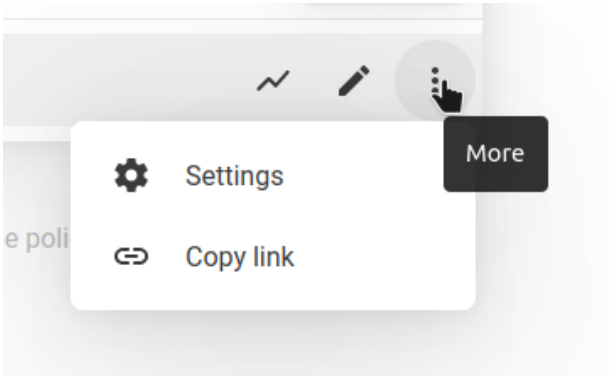


Obrázek 8: Ukázka informačního modální okna. Zdroj: [autor]



Obrázek 9: Ukázka akčního modálního okna. Zdroj: [autor]

Kontextuální nabídka Jedná se o malé menu zobrazující rozšířené možnosti nějaké akce. Většinou se používá, pokud není v UI dostatek místa pro všechny dostupné akce, nebo ve formuláři pro výběr konkrétní položky z výběrového pole.



Obrázek 10: Ukázka kontextuální nabídky akcí položky v seznamu. Zdroj: [autor]

7.5.3 Prvky pro interakci

Po prvcích strukturující obsah je možné navrhnout prvky, se kterými bude uživatel přímo či nepřímo interagovat.

Tlačítko Nejhlavnějším takovým prvkem je tlačítko. To umožňuje uživatelům vyvolávat akce, potvrzovat akce nebo se nechat přesměrovat na jiné stránky. Tlačítek je v aplikaci spousta a každé dělá trochu něco jiného. Lze je ale zařadit do pár kategorií, a navrhnout tak několik obecných typů. Všechny tlačítka byla rozdělena na tlačítka s nízkou prioritou, vysokou prioritou a vysokou vážností akce.

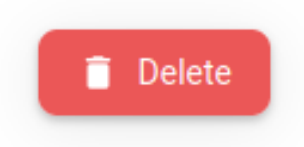
Tlačítko s nízkou prioritou reprezentuje vedlejší akci, kterou sice uživatel může provést, ale není jeho hlavním cílem. Takové tlačítko má neutrální barvu a ve většině případech neobsahuje ani ikonu.

Obrázek 11: Ukázka tlačítka s malou prioritou. Zdroj: [autor]

Tlačítko s vysokou prioritou představuje primární akci, kterou by měl uživatel provést. Taková tlačítka představují potvrzení akcí, vytvoření nových objektů, přidání existujících objektů a podobně. Měla by být proto patřičně zvýrazněna a uživatele podvědomě navádět. Z tohoto důvodu mají tlačítka výraznou barvu a vystihující ikonu.

Obrázek 12: Ukázka tlačítka s vysokou prioritou. Zdroj: [autor]

Pro případy, kdy uživatel provádí nějakou akci s vysokou vážností, jako je například nevratné smazání nějakého objektu, vzniklo rozšíření tlačítka s vysokou prioritou. Tlačítko se liší pouze v červené barvě evokující dojem závažné akce.



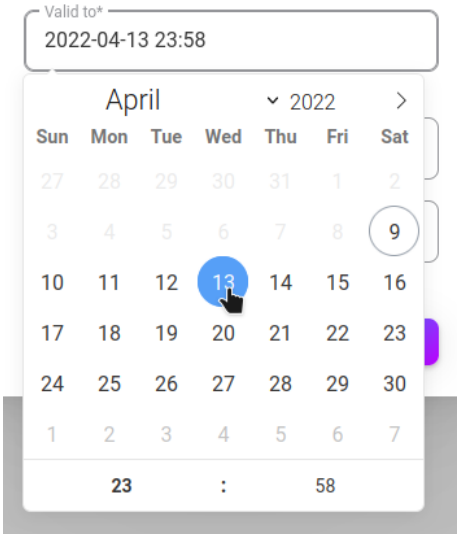
Obrázek 13: Ukázka tlačítka provádějící nevratnou změnu. Zdroj: [autor]

Vstupní pole Vstupní pole různých druhů jsou společně s tlačítky nejdůležitějšími prvky jakékoliv aplikace. Vstupní pole umožňují uživatelům předat data aplikaci v různých formátech. Vzniklo proto několik základních typů vstupních polí, které lze uvnitř v formulářů jednoduše kombinovat. Mezi ně patří: textová pole, výběrová pole, pole pro výběr času a data, pole pro nahrání obrázků a pole pro výběr geografických lokací. Vstupní pole využívají linek namísto stínů, protože reprezentují pouze část celého bloku. Pole mají popisek pro identifikaci účelu, ikonu a mohou mít i akční tlačítko. Pod pole je navíc možné vepsat nápovědu.

Textová pole slouží pro zadávání krátkých i dlouhých textů, mají vždy popisek a mohou mít ikonu a akční tlačítko.

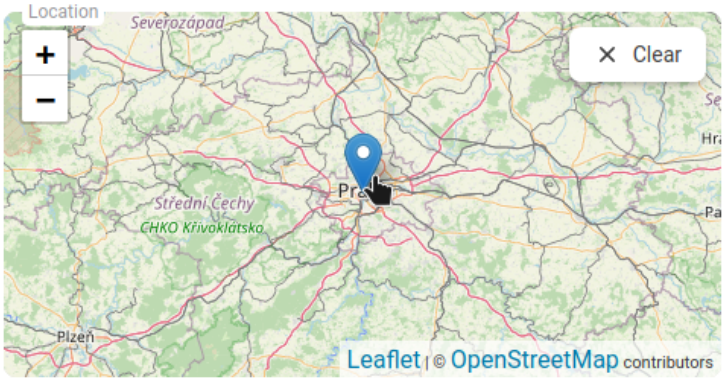
Obrázek 14: Ukázka textového pole s ikonou a nápovědou. Zdroj: [autor]

Extenzí textového pole jsou pole pro zadání času a data. Ty rozšiřují manuální zadávání o okénka pro pohodlný výběr bez nutnosti klávesnice.



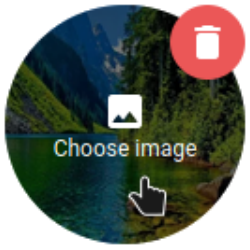
Obrázek 15: Ukázka pole s výběrem data a času. Zdroj: [autor]

Vzhledem k požadavku specifikovat geografické lokace karet, bylo potřeba zavést i specifická pole pro výběr bodů v mapě. Pole zobrazuje mapu světa, se kterou je možné hýbat, přibližovat ji a vybírat konkrétní body.



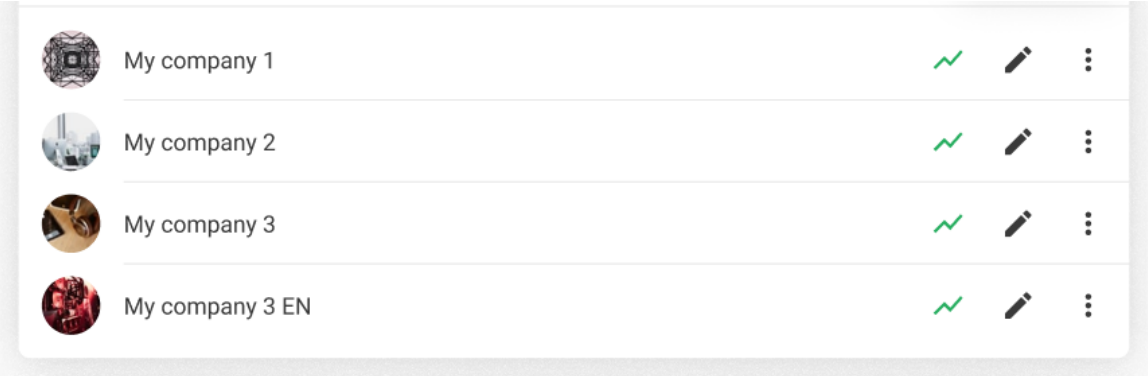
Obrázek 16: Ukázka pole pro výběr geografického bodu v mapě. Zdroj: [autor]

Posledním méně používaným polem v této aplikaci je pole pro výběr obrázku. Pole umožní výběr souboru ze souborového systému uživatele, který je následně nahrát a zobrazen jeho náhled přímo v poli. Vybraný obrázek je pak možné jednoduše změnit nebo úplně odstranit.



Obrázek 17: Ukázka pole s nahraným obrázkem. Zdroj: [autor]

Seznam položek Velmi využívaným prvkem je seznam položek. Každá položka může obsahovat ikonu nebo obrázek, text, seznam akcí, nebo dokonce i vnořený další seznam. Akce položek se kromě běžného zobrazení umí v případě malého místa seskupit a schovat do kontextuální nabídky, aby uživatel využívající malé zařízení nepřišel o možnost využívat všechny akce.



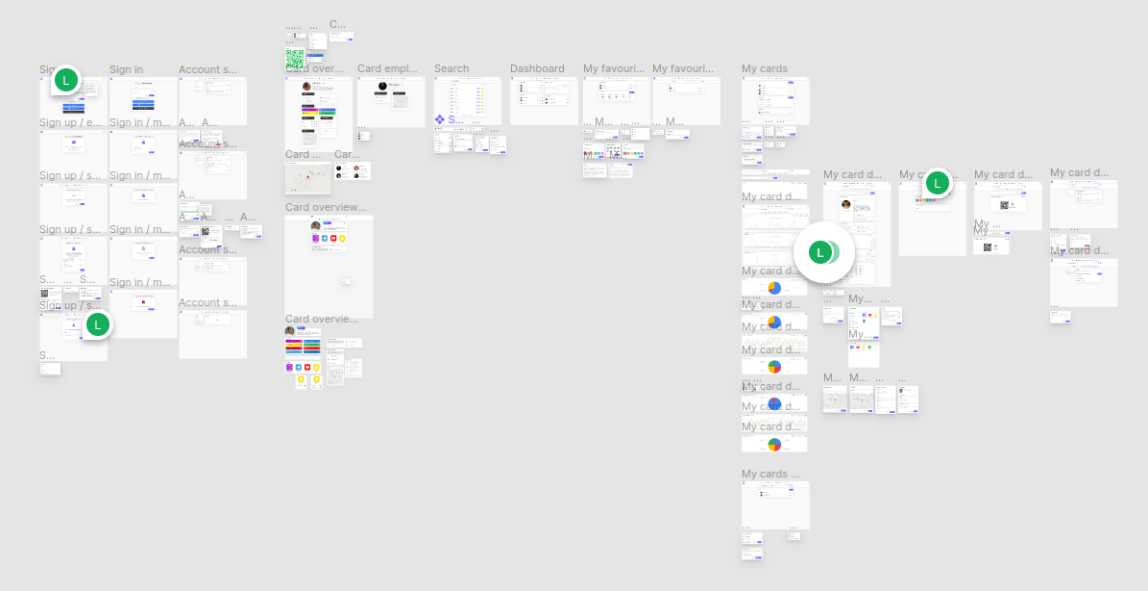
Obrázek 18: Ukázka pokročilého seznamu prvků. Zdroj: [autor]

Stránkování Při velkém množství položek v jednom seznamu se využívá systém stránkování. Prvek stránkování umožňuje přepínat mezi konkrétními stránkami, nebo se přepínat mezi předchozí a následující stránkou.

Obrázek 19: Ukázka stránkování. Zdroj: [autor]

7.5.4 Finální návrh

S pomocí výše navržených prvků a jejich variacemi, bylo navrženo finální UI všech stěžejních stránek.



Obrázek 20: Finální grafický návrh aplikačních stránek a prvků. Zdroj: [autor]

Tento postup se ukázal jako velmi efektivní oproti původnímu prototypu, protože pomohl snadněji implementovat samotné GUI a navrhnout API díky zřetelným funkčním požadavkům.

7.6 Implementace API serveru

7.6.1 Datový model

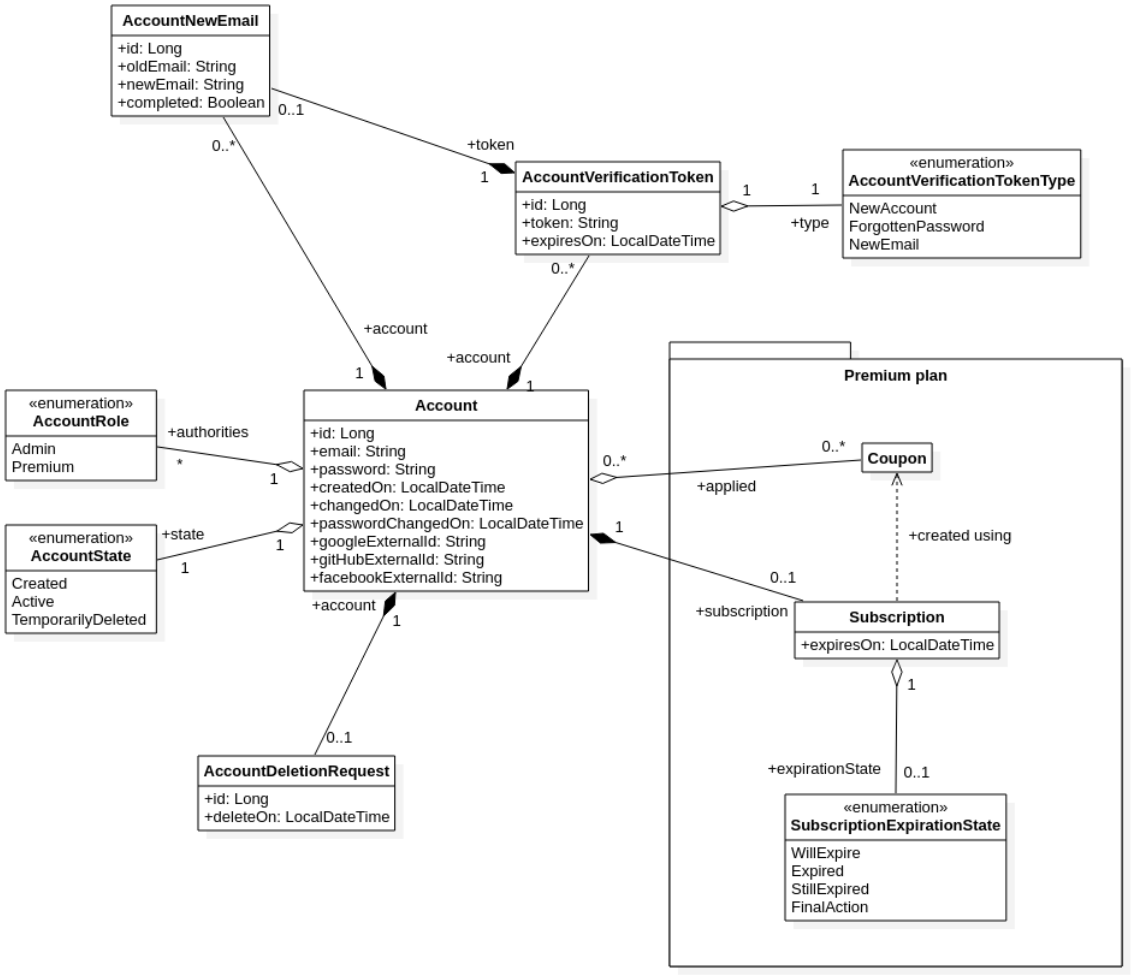
Úkol API serveru je poskytovat a zpracovávat data, ať už textová v podobě JSON dokumentů, nebo binární v podobě obrázků. Tyto funkce pro samotné API bude zprostředkovávat implementovaná business logika starající se o to, aby všechna data byla správně zpracována a uložena. Je ale stěžejní se na počátku zamyslet, vybrat styl a navrhnout model, jakým bude business logika implementována. Stěžejní je to z důvodu budoucí čitelnosti kódu a možnostech rozšíření o nové funkce.

Vzhledem k tomu, že byl vybrán jazyk Java, který je primárně objektově orientovaným programovací jazykem, business logika bude využívat právě principů OOP. OOP je velice populární a většinově používané programovací paradigma využívané spoustou programovacích jazyků. Toto paradigma využívá objekty pro sestavení systému. Každý objekt má nějaké své vlastnosti a operace, které může buď využívat vnitřně nebo poskytnout ostatním objektům. Hlavní myšlenka OOP je popisovat reálné objekty (člověk, pes, budova, událost...) těmi virtuálními ve zjednodušené formě, v níž popisujeme jen tu část reálného objektu, která nás zajímá. Tyto objekty pak v systému mezi sebou komunikují pomocí zpráv a předávají si data tak, aby dosáhly výsledku. Každý objekt v systému má svou předlohu zvanou třída jenž definuje podporované vlastnosti a operace. Kromě tříd ještě existují rozhraní a abstraktní třídy, které můžou mezi sebou dědit vlastnosti a operace. Rozhraní

a abstraktní třídy definují obecnou předlohu operací, tzv. kontrakt, které dědící třída musí splňovat, což následně umožňuje využít techniku polymorfismus, tedy provádět operace na základě obecného rozhraní bez nutnosti znát konkrétní třídu. [55]

Právě díky OOP bylo možné poměrně věrně reprezentovat objekty využívané jednotlivými stránkami v logické, přehledné a jasně definované struktuře. Model byl pro přehlednost interně rozdělen na několik modulů podle zaměření, nicméně všechny moduly mezi sebou i tak komunikují. Momentálně se datový model skládá z modulu pro správu uživatelských účtů, modulu pro správu karet a modulu pro správu kupónů.

Model uživatelských účtů Modul uživatelských účtů je zodpovědný za vytváření, uchování, úpravu a mazání jednotlivých účtů. S tím je spojena i správa oblíbených karet a poznámek, které jsou vázány na konkrétního uživatele. Modul dál řeší i prémiový plán a předplatné.



Obrázek 21: Datový model popisující uživatelské účty. Zdroj: [autor]

Jak je možno vidět z diagramu, účet má kromě základních přihlašovacích informací i spoustu doplňujících dat.

V první řadě, každý účet má přiřazený vždy nějaký stav. Při vytvoření má účet stav `Created` vyjadřující nový a zatím neaktivovaný účet. Tento účet nelze využívat. Po aktivaci

se změní stav účtu na `Active`, který říká, že účet je již aktivovaný, a tedy plně použitelný. Poslední stav `TemporarilyDeleted` označuje dočasně smazaný účet, což znamená, že je naplánováno jeho permanentní smazání v budoucnu. To umožňuje uživatelům v omezené lhůtě svůj účet obnovit.

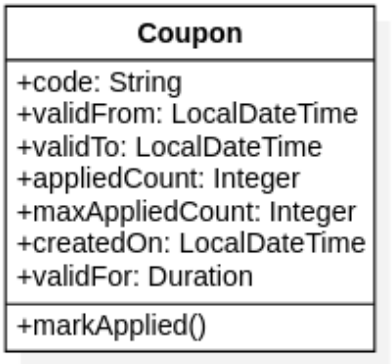
Další velmi důležitou komponentou je systém ověřovacích tokenů sloužících k potvrzování akcí, kde je potřeba ověřit, že akci provádí skutečný uživatel, nikoliv podvodník. Tokeny jsou pseudo náhodně generované unikátní textové řetězce přiřazené vždy k jednomu určitému uživateli. Typ tokenu pak říká, co daný token ověřuje. Ověřitelnou akcí může být aktivace nového účtu, změna emailové adresy nebo zapomenuté heslo. V případě tokenu pro změnu emailové adresy se nová emailová adresa ukládá přímo k danému tokenu, tedy mimo účet. Díky tomu je možné emailovou adresu změnit, až ve chvíli, kdy byl token ověřen, a zároveň je možné v případě odcizení účtu zobrazit historii všech změn.

Systém dále pracuje s rolemi udávajícími obecná práva uživatele. Každé obecné roli jsou přiřazena nějaká práva v systému (čtení, modifikace...), a pokud uživatel má takovou roli přiřazenou, automatický dědí její práva. Toto je poměrně jednoduchý a efektivní systém práv. Momentálně systém pracuje s administrátorskou rolí a prémiovou rolí. Administrátorská role uděluje uživateli práva pro přístup do interní administrační části aplikace, pomocí které je možné spravovat provoz aplikace. Druhou rolí je role prémiová, která je určena pro běžné uživatele, a odemyká jim prémiové funkce aplikace. Tato role vzniká automaticky při vytvoření předplatného a zaniká s jeho expirací.

Pro plánované mazání účtů se uchovává požadavek s informací o tom, jaký účet bude smazán, a kdy dojde k permanentnímu smazání společně s jeho daty.

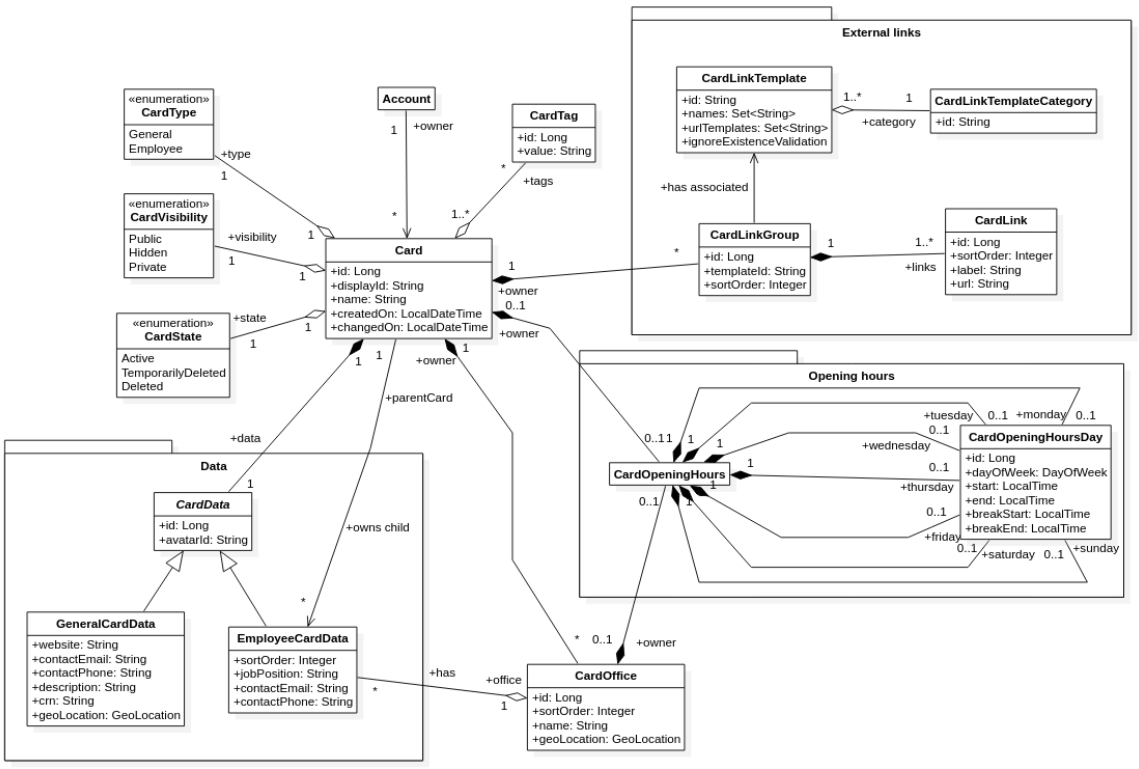
Poslední částí uživatelských účtů jsou komponenty pro prémiové funkce. Jedná se především o popisný objekt předplatného. Ten vzniká aktivací prémiovýho plánu a říká, že účet má práva na prémiové funkce. V jistých případech může mít přiřazený i datum expirace, se kterým automaticky zanikne. S tím se váže i stav expirace, která pomáhá upozorňovat uživatele na současný stav předplatného.

Model kupónů Předplatné může vzniknout na základě zadání platného kupónu, avšak systém kupónů je navržen obecně, aby bylo možné ho v případě budoucích rozšíření jednoduše přepoužít i na aktivaci jiných funkcionalit, ne jen pro prémiové plány. Každý kupón musí mít unikátní kód, buď vygenerovaný nebo zadaný správcem. Mimo to má definovanou platnost, maximální počet možných aplikování a dobu po jakou bude vytvořené předplatné platné.



Obrázek 22: Datový model kupónového systému. Zdroj: [autor]

Model karet Nejhlavnějším prvkem v této aplikaci jsou samotné karty a jejich data.



Obrázek 23: Datový model karet a navázaných dat. Zdroj: [autor]

Každá karta má v prvé řadě vždy nějakého vlastníka, který má jako jediný práva na úpravu. Samotná karta je rozpadnuta do několika částí pro podporu více typů karet. Srdcem je jednoduchá třída popisující pouze obecná metadata, která jsou potřeba pro všechny typy karet. V této třídě je tedy zahrnut vlastník, typ, viditelnost, stav, veřejné jméno, display ID a další metadata. Display ID je unikátní textové ID karty určené primárně pro koncové uživatele. Toto ID je totiž zobrazováno v URL adrese konkrétní karty a má za cíl umožnit jednoduše sdílet karty bez nutnosti znát kryptické náhodně generované číselné identifikátory. Toto ID může být definováno ručně uživatelem nebo generováno aplikací (vhodné pro skryté karty). Typ karty definuje jak s danou kartou lze pracovat a jaká data poskytuje. Momentálními typy jsou obecná karta a karta zaměstnance, avšak tento systém dovoluje poměrně jednoduché rozšiřování o nové typy karet. Obecné karty je možné tvořit uživateli přímo a podporují všechny typy údajů. Původně byl tento typ rozdělen ještě na osobní a firemní karty, nicméně jediný rozdíl spočíval v omezení podporovaných dat v osobních kartách. Tento přístup však přinášel mnoho duplicit, a zároveň znemožňoval uživatelům osobních karet využívat funkce firemních karet, jenž za určitých okolností dávaly smysl i u osobních karet (např. pro OSVČ). Právě kvůli těmto nevýhodám a žádným jasným výhodám, byly tyto dva typy sjednoceny, především pro pohodlí koncových uživatelů. Karty zaměstnanců se pak tvoří nepřímo jako hierarchičtí potomci obecných karet v editaci nadřazené karty, a obsahují specifická data pouze pro zaměstnance. Posledním důležitým metadatem je viditelnost specifikující, kdo může danou kartu vidět. Veřejné karty může navštívit a vyhledat kdokoliv. Skryté může navštívit pouze uživatel disponující konkrétní

URL adresou odkazující na danou kartu. Privátní karta pak už slouží pouze pro autora karty.

Každé kartě lze přiřadit libovolné množství štítků pro kategorizaci. Štítky jsou uživatelsky definované, a můžou tak obsahovat cokoliv. Navíc jsou znovupoužitelné, takže více karet může být označeno stejným štítkem.

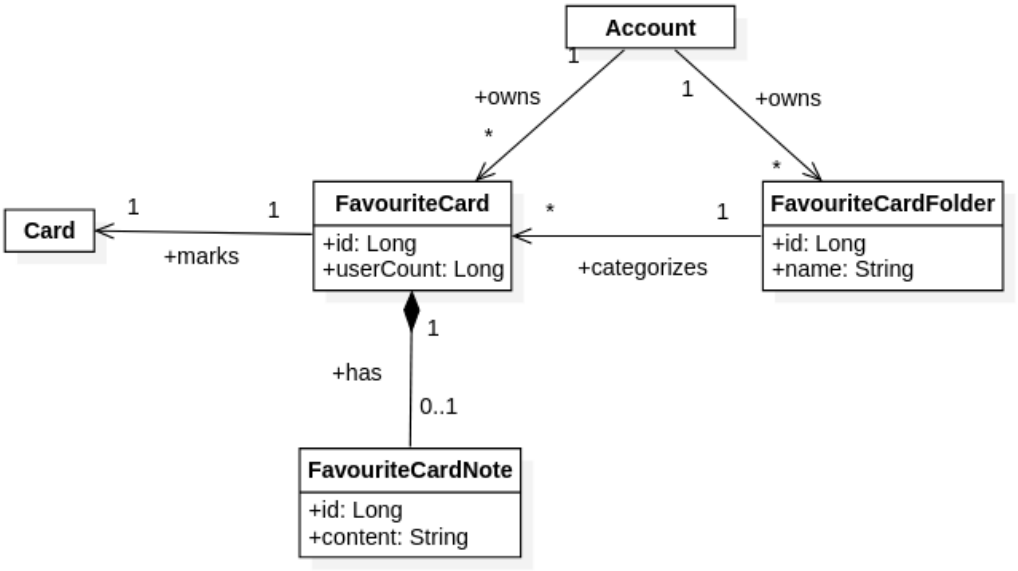
Pro definování informací o kartě existuje koncept napojených dat. Tyto data reprezentují soubor údajů, který daný typ karty podporuje. Každý typ karty má svoji implementaci těchto dat. Je pak na business logice aplikace, aby dokázala s daty správně pracovat. Tato obecnost umožňuje jednoduše vytvářet nové typy karet s úplně odlišnými daty, aniž by se data musela míchat a být nepřehledná.

Pro většinu karet pak pravděpodobně nejdůležitějšími daty budou URL odkazy na externí webové stránky, především na sociální sítě. Ty z karet dělají jakési rozcestníky pro objevení profilů subjektů na sociálních sítích. Každý jeden odkaz má originální cílovou URL adresu, popisek a pořadí mezi ostatními odkazy na stejné úrovni. Popisek slouží především pro odlišení více odkazů vedoucí na stejnou webovou stránku. URL adresa je hlavní informace a z důvodu bezpečnosti musí být validována oproti zvolené šabloně sítě. Díky tomu není možné vytvořit odkaz pod jednou sítí, ale odkazovat na podvodnou. Jednotlivé odkazy jsou pak seskupovány do skupin odkazů vycházející ze stejné šablony. I přes to, že skupiny zastřešují vždy nějaký soubor sítí stejného typu, může existovat více skupin vycházející ze stejné šablony. To umožňuje poměrně velkou flexibilitu struktury odkazů a neomezuje uživatele na jeden odkaz pro jednu síť (což je nevýhoda některých konkurenčních řešení). Jednotlivé šablony jsou reprezentovány separátními konfiguračními třídami. Z návrhu vyplývá potřeba definovat názvy, podle kterých lze šablony vyhledat, a platné šablony URL odkazů oproti kterým se validují konkrétní cílové URL adresy. Stejně jako názvů, může být i více šablon URL adres, protože většina webových stránek podporuje více domén třetího řádu, a je tak nutné definovat všechny, protože není jasné jakou URL adresu uživatel zadá. Každá taková šablona je pak zařazena do jedné nebo více kategorií sloužící čistě pro jednodušší vyhledávání.

Pro firmy poměrně užitečnou informací může být jejich hierarchie poboček a zaměstnanců, zejména u větších společností. Proto každá obecná karta může mít libovolné množství přiřazených poboček/kanceláří/prodejen, kde každá má své jméno a geografickou lokaci. Ke každé takové pobočce je možné přiřadit zaměstnanecké karty. Kromě zaměstnanců může mít pobočka i vlastní otevírací dobu.

Velmi důležitou informací pro potencionální zákazníky je aktuální otevírací doba. Byl proto navržen obecný model otevírací doby, který lze navázat na jakoukoliv jinou entitu. Model je zastřešen jednou třídou seskupující otevírací dobu jednotlivých dnů v týdnu. Každý den má pak definovaný interval, kdy má daná entita otevřeno, a kdy má přestávku.

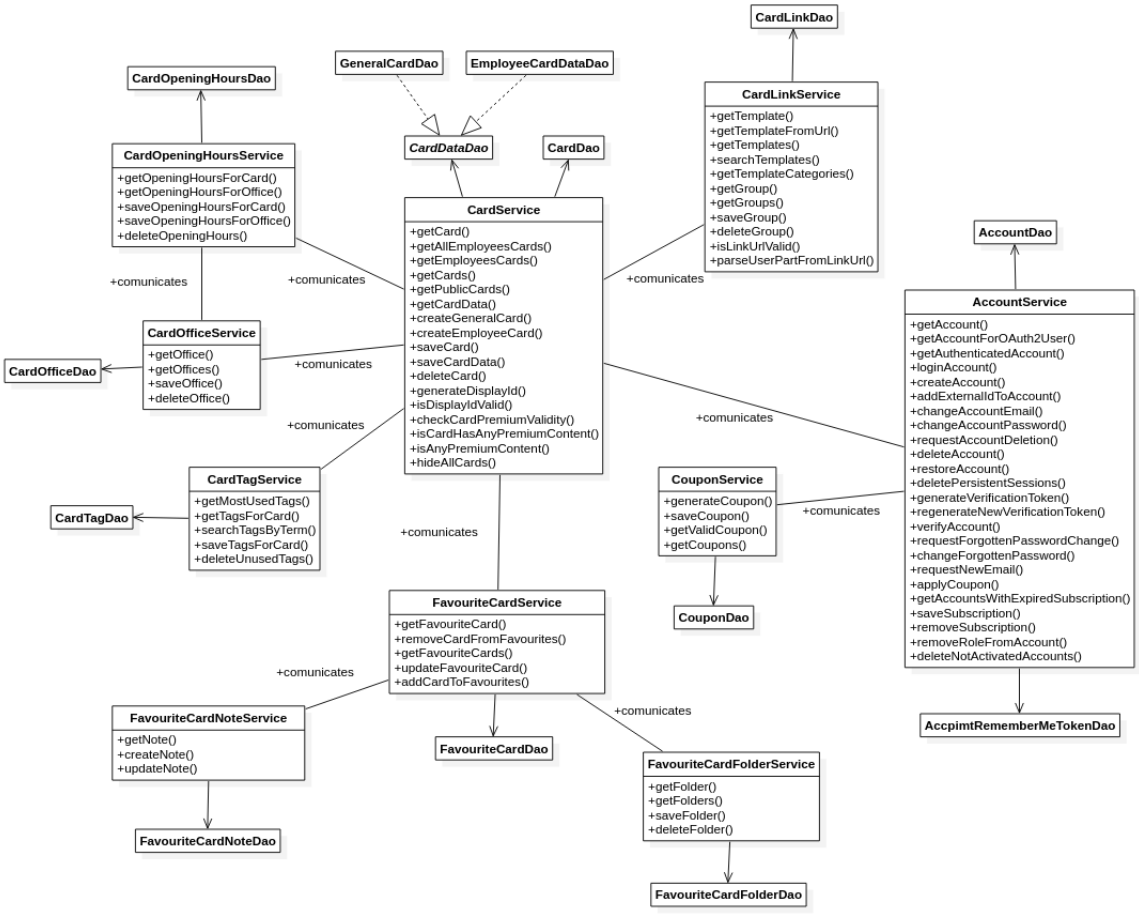
Model oblíbených karet Oblíbené karty umožňují k uživatelskému účtu přiřazovat cizí karty jako oblíbené. Každý účet může jako oblíbené označit libovolný počet karet, což danému uživateli umožní rychlý přístup k požadovaných informacím. Každá oblíbená karta navíc může mít jednu textovou poznámku s formátováním. Oblíbené karty je pak možné seskupovat do složek definovaných uživatelem pro lepší orientaci. Složky, poznámky ani oblíbené karty však nevidí nikdo jiný než autor.



Obrázek 24: Datový model oblíbených karet uživatelských účtů. Zdroj: [autor]

7.6.2 Práce s datovým modelem

Práci s datovým modelem zastřešuje právě již několikrát zmíněná business logika. Struktura business logiky si bere inspiraci z návrhového vzoru DDD (Domain-driven design), který se zabývá navrhováním přehledné, znovu použitelné a hlavně jednoduše rozšiřitelné struktury, řešící zadané businessové požadavky projektu. Tento vzor pohlíží na problematiku jako na doménu, a staví hlavně na konceptu služeb, což jsou speciální třídy, které neudržují vnitřní stav, ale poskytují služby manipulující s datovými objekty [56]. Jednotlivé služby by se měli vždy zabývat pouze jedním typem dat kvůli přehlednosti, a ostatní data delegovat na ostatní služby [56]. Druhým podstatným stavebním blokem jsou repozitáře zajišťující předávání dat mezi aplikací a databázovým systémem [56]. DDD obsahuje spoustu dalších konceptů, nicméně ty byly při implementaci využity jen vzdáleně. Implementovaná business logika má tak několik službových tříd a repozitářů. Pro každou separátní část modelu existuje manipulující služební třída a jeden repozitář starající se především o načítání a ukládání dat do databáze. Konzumující logika si pak může vybrat přesně s jakými daty chce pracovat. Zatímco repozitáře se starají pouze o komunikaci s databází pomocí SQL jazyka, služební třídy tyto informace předávají dál, upravují nebo validují.



Obrázek 25: Zjednodušený model služeb a repozitářů. Zdroj: [autor]

Momentální implementace služeb komunikuje mezi sebou převážně pomocí primárních klíčů jednotlivých entit, aby bylo zaručeno, že entita, se kterou služba manipuluje je vždy aktuální verzí reflektující stav databáze. Nicméně tento přístup značně zesložiťuje závislost tříd na jiných třídách, což vede k složitějšímu instanciování a testování tříd.

Bohužel, při postupném rozšiřování aplikačních požadavků se ukázalo, že rozpad logických celků entit není úplně vhodný, a i když funguje bez problémů, jeho testování a nastavování je složité. Tento problém však řeší právě návrhový vzor DDD svými entitami a hodnotovými objekty. Podle tohoto vzoru by se mělo pracovat s logickými celky jako s jednou entitou, aby se předešlo výše zmíněné složitosti rozpadlosti služebních tříd. Další podstatnou výhodou této provázanosti je zaručení konzistentnosti dílčích entit. Implementovaný přístup tento problém také řeší, a to pomocí validací, nicméně kvůli nutnosti se stále dotazovat na stav nadřazených entit složitost rapidně stoupá. [56] Pro momentální stav aplikace je implementovaný model dostačující, nicméně pro budoucí rozšíření by bylo nutné model značně optimalizovat pomocí DDD vzorů.

7.6.3 Komunikace s databázových systémem

Prvním problémem k řešení při implementaci již samotné business logiky bylo mapování objektů do databázové struktury.

V případě, že by aplikace využívala nějakou z NoSQL databází s podporou JSON dokumentů, by bylo toto mapování poměrně jednoduché. Jedinou složitost by pravděpodobně představovalo mapování hierarchických karet.

V případě vybrané SQL databáze je ale problém složitější, protože tyto databáze pracují s tabulkami místo objekty. Na první pohled se může zdát, že by to neměl být velký problém, protože sloupce tabulek jsou podobné atributům objektů. Složitost je hlavně v načítání a ukládání komplexních entity složených z více objektů a kolekcí, hlavně pak kolekcí 1:N (případně M:N). Jazyk SQL totiž neumožňuje jednoduše vrátit více řádků z různých tabulek, a tak není možné najednou vrátit objekty různých tříd tvořící jeden celek. Stejný problém je i v případě ukládání dat, při němž SQL umožňuje ukládat pouze řádky do jedné tabulky. Ještě horší je to v případě ukládání již existujících dat, protože SQL neumožňuje hromadné úpravy.

Naštěstí existuje několik knihoven a nástrojů, které tyto problémy řeší za programátora. Je však možné vše implementovat ručně, to je ale zdlouhavé a zbytečné. Mezi nejpoužívanější knihovny patří Hibernate a MyBatis. Poměrně novou zajímavou alternativou je Spring Data JDBC, bohužel tato knihovna v době vývoje nebyla příliš rozšířená a neosvědčená na to, aby se dostala do výběru.

Hibernate a MyBatis mají dost rozdílné přístupy k řešení výše zmíněných problémů. Zatím co Hibernate se snaží být plným ORM frameworkem a odstínit programátora od jakéhokoli SQL jazyka [57], MyBatis je lehčí alternativa, která poskytuje pouze nástroje pro mapování a dotazování, ale nezatěžuje programátora všemi nízkoúrovňovými problémy [58]. Další velký rozdíl je v práci s entitami na pozadí. Hibernate využívá mezičlánku mezi repozitářem a databází, kde jsou dočasně entity uloženy [57]. Díky tomu je Hibernate schopen poskytovat mezipamět pro entity při vícenásobném dotahování, a při ukládání se data nepromítají hned, ale až na konci transakce [57]. Kvůli tomu se tak repozitář chová spíše jako paměťové úložiště, v němž má programátor okamžitý přístup k objektům. Bohužel to má i svá úskalí, a to nejen v celkové komplexitě knihovny. Úskalí se skrývá, hlavně pro tento projekt, ve sdílení instancí jedné entity při vícenásobném dotažení entity z databáze. Tento přístup zamezuje porovnávání originální verze entity a nové upravené (většinou přijaté skrze API), protože interně je Hibernate považuje za jednu instanci, a upravují se tak obě dvě. Hibernate má sice dobré nástroje pro validaci entit, ty však validují pouze aktuální hodnoty v objektech, ale není možné porovnat hodnoty s původními verzemi entit. To je v případě, kdy má entita vlastnosti, které se nesmí změnit, poměrně limitující. Instance entity sice lze vyjmout z Hibernatu, to je však poměrně složité. MyBatis tento neduh nemá a vrací vždy novou instanci entity. MyBatis se nesnaží řešit vše, a díky tomu je jeho abstrakce mnohem jednodušší. To ovšem znamená, že programátor musí odvést více práce při mapování a psát SQL dotazy. Musí také ručně řešit ukládání složených entit, což Hibernate řeší sám.

I když by se mohlo zdát, že Hibernate je ultimátním nástrojem pro 99% projektů, existuje mnoho materiálů zkušených programátorů využívajících Hibernate spoustu let, kteří si nemyslí, že Hibernate je ideálním nástrojem pro všechny projekty. Jeho komplexnost totiž může přinést spoustu problémů, které na první pohled nemusí někoho napadnout. Mezi největší problémy patří nejednoznačné zobrazování chyb, nutnost přizpůsobit databázový model Hibernatu, náhodné chyby s odsunutým dotahováním dat nebo nejednoznačné operace v API knihovny. Na druhou stranu spousta uživatelů si míru abstrakce pochvaluje. [59]

Při výběru tedy hodně záleží na konkrétním projektu a zkušenostech implementátora. Pro tuto aplikaci byl z počátku zvolen Hibernate, právě díky jeho abstrakci mapování. Bohužel se posléze začal projevovat již zmiňovaný problém s validací entit. Po dodatečné analýze

padlo rozhodnutí nahradit Hibernate MyBatisem. Za cenu prvotní transformace, MyBatis zjednodušil práci s entitami a hledání chyb bylo přímočařejší. MyBatis navíc podporuje konfiguraci pomocí XML souborů, nebo pomocí Java anotací přímo v Java rozhraních. Pro zjednodušení a držení logiky na jednom místě byl vybrán přístup s Java anotacemi. Ten se ukázal jako dostačující, avšak našli se výjimky, kdy složitost použitého SQL kódu byla na hraně přehlednosti. Momentálně tedy existuje pro každý typ entity třída reprezentující repozitář, která obsahuje metody obalené SQL dotazy. Zde je možné vidět reprezentativní ukázku části repozitáře pro práci s uživatelskými účty.

```
@Mapper
public interface AccountDao {
  @Select({
    "select *",
    "from " + Account.TABLE_NAME,
    "where email = #{email}"
  })
  Optional<Account> findByEmail(String email);
}
```

Ukázka kódu 1: Ukázka získání uživatelského účtu z databáze MyBatisem. Zdroj: [autor]

7.6.4 Validace entit

Validace entit je velice důležitá, pokud uživatelé mohou data upravovat zvenčí. Díky ní se do databáze dostanou jen validní data. Tato aplikace využívá dvou typů validací, každou pro validování něčeho jiného. První a hlavní validací je validace hodnot entit. Tato validace je ta jednodušší, protože je jasně předem daný formát hodnot, a navíc pro tuto problematiku existují knihovny. Druhým typem využívané validace je validace hodnoty oproti původní verzi. Takovými hodnotami jsou většinou stavy entit nebo různé datumy. Příkladem může být stav karty, kde smazaná karta již nesmí přejít do stavu aktivní karty, protože nemá potřebná data. Dalším příkladem je datum vytvoření entity, který se nesmí změnit jinak by postrádal svůj účel.

Pro tento projekt byla využita knihovna Hibernate Validator zabývající se především prvním typem validací. K definici pravidel využívá Java anotace. Pravidla se můžou stahovat buď na celou třídu nebo jen na konkrétní atributy. Knihovna sama o sobě poskytuje rozsáhlou sadu běžných pravidel, avšak umožňuje jednoduché rozšíření o vlastní pravidla. Takto může vypadat třída s definovanými pravidly.

Díky rozšiřitelnosti této knihovny je možné jednoduše implementovat i druhý typ validací. Tyto validace pak mají vlastní Java anotace a validátory. Pokud se nějaký takový atribut o proti původní verzi změnil, validace selže a neumožní uživateli uložit novou verzi entity.

Po nastavení pravidel entitám je již možné provádět samotnou validaci. K tomu v knihovně Hibernate Validator slouží jedna centrální třída validátoru. Ta přijímá validovanou entitu a vrací kolekci chyb, které jsou následně přeloženy na chyby API.

```
@CardLinkUrl
public class CardLinkGroup {
  @CardLinkTemplateId
  private String templateId;

  @NotNull
  @Size(min = 1)
  private List<@Valid CardLink> links;
}
```

Ukázka kódu 2: Ukázka třídy s validačními pravidly pomocí Java anotací. Zdroj: [autor]

7.6.5 Úložiště souborů

Jak již bylo nastíněno, aplikace nebude implementovat vlastní souborové úložiště. Místo toho bude využívat externí službu Cloudinary. Aplikace k ukládání a získávání souborů využívá API vystavené službou Cloudinary. Získávání variant obrázků do GUI zajišťuje front-endová aplikace přímo ze serverů Cloudinary.

Cloudinary poskytuje jeden velký adresář pro všechny soubory, což je při více typů souborů nevhodné. Naštěstí podporuje vnořené adresáře. Z tohoto důvodu vznikl koncept typů uložišť, kde každý typ definuje cestu relativního kořenového adresáře, typ ukládaných souborů a interní ID. Momentálně existuje pouze jedno úložiště, a to pro profilové obrázky karet.

Aby bylo možné zpětně jednoduše identifikovat autora a typ úložiště uloženého souboru, je potřeba ke každému souboru definovat dodatečná metadata. Každý uložený soubor je popsán následující třídou obsahující všechny potřebné informace o souboru pro zobrazení a validaci:

```
public class StorageFile {
  private final StorageType storageType;
  private final long ownerId;
  private final String publicId;
  private final String url;
}
```

Ukázka kódu 3: Třída popisující uložený soubor v Cloudinary. Zdroj: [autor]

Autor a typ úložiště slouží momentálně hlavně k jednoduché validaci zvoleného obrázku při ukládání karty, kdy se validuje jestli majitel karty je zároveň majitelem obrázku a zdali se jedná skutečně o profilový obrázek. Tyto validace existují, aby nebylo možné někým záměrně využít obrázek někoho jiného nebo soubor, který k tomu není určený.

Finální uložení se skládá pouze z načtení binárních dat, sestavení metadat a zavolání API služby. Získání popisného souboru z Cloudinary je podobně jednoduché jako uložení, protože API umožňuje získat data o jednotlivých souborech pomocí interního ID.

Popisující soubor se obecně používá na místech, kde nejsou potřeba konkrétní binární data souboru, pouze informace o něm. Binární data jsou potřeba pouze v případě zobrazení souborů v GUI. Konkrétní binární data souborů pro zobrazení například v GUI aplikace je možné získat přímo ze serverů Cloudinary pomocí speciální URL adresy obsahující iden-

tifikaci úložiště, identifikaci konkrétního souboru a požadovanou variantu. Díky přímému přístupu je možné využít CDN pro menší odezvy při stahování. Konkrétní URL adresa pro stažení varianty profilového obrázku karty o rozměrech 256x256 pixelů může vypadat následovně:

```
https://res.cloudinary.com/id_uloziste/image/upload/t_256x256/id_souboru
```

Ukázka kódu 4: Ukázka URL adresy souboru v Cloudinary úložišti. Zdroj: [autor]

V tomto případě je varianta nakonfigurována přímo v administraci Cloudinary a v URL adrese je pouze její jméno, je však možné formát varianty nadefinovat přímo v URL adrese.

7.6.6 Transakční emaily

Cílem bylo vytvořit systém umožňující odesílat předem definované zprávy na základě aplikačních událostí. Tím že aplikace bude využívat externí službu SendinBlue, v aplikaci bylo nutné připravit pouze propojení API s aplikačními událostmi.

Pro odesílaní emailů se nejdříve připravily šablony zpráv, aby bylo co odesílat. K tomu byl využit integrovaný klikací editor, protože podpora zobrazování ostylovaného HTML obsahu v emailových klientech je poměrně omezená a velice se liší mezi jednotlivými klienty. Kvůli tomu by bylo časově náročné připravit šablonu fungující na všech populárních klientech, a to navíc ve dvou verzích: desktopové a mobilní.

Každá šablona má v systému unikátní ID, takže byla tvořena jednoduchá mapovací enumerace. Díky této enumeraci je pak v aplikaci jasné jaká šablona bude použita. Dále může šablona využívat parametry místo konkrétního textu, které jsou při zpracování šablony nahrazeny parametry požadavku. Těmi může být cokoliv, například emailová adresa adresáta nebo expirační datum nějaké entity.

Odeslání konkrétního emailu už je pak jednoduché a vyžaduje pouze specifikování adresáta, šablony emailu a parametrů, a následné odeslání požadavku do API služby.

Jak bylo nastíněno, systém bude odesílat emaily automaticky, jako reakci na vzniklé události v aplikaci. Události jsou v tomto případě reprezentování aplikačními událostmi z frameworku Spring. Spring umožňuje do systému publikovat aplikační události a definovat pozorovatele, kteří těmto událostem naslouchají a mohou na ně reagovat. Samotná aplikační událost je pouze jednoduchý objekt nesoucí kontextová data reprezentovaná událostí.

```
public class CardUpdatedEvent extends ApplicationEvent {
  private final Card originalCard;
  private final Card updatedCard;
  /* ... */
}
```

Ukázka kódu 5: Ukázka aplikační události. Zdroj: [autor]

Příkladem může být událost vyvolaná jako následek uložení upravené karty. Událost reprezentuje úpravu karty a nese sebou upravenou kartu, kterou pozorovatel může využít pro provedení svých operací.

Právě tento přístup využívá modul odesílání emailů. Modul má do systému vystavené vlastní posluchače na konkrétní aplikační události (změna emailu uživatele, změna hesla, vypršení platnosti předplatného...), kteří při vzniku údalostí odesílají pomocí API emaily.

```
public class CardUpdatedEventListener {
  @EventListener
  public void processCardUpdatedEvent(@NonNull CardUpdatedEvent event) {
    /* ... */
  }
}
```

Ukázka kódu 6: Ukázka pozorovatele aplikačních událostí. Zdroj: [autor]

Tento přístup je vhodný, protože ostatní moduly vůbec nevědí o modulu emailů, a nevzniká tak fyzické propojení znepřehledňující kód. Navíc je jednoduché přidávat nové události a jejich posluchače.

7.6.7 Vyhledávání karet a míst

Fulltextové vyhledávání je komplexní operace, která vyžaduje analýzu textů a přípravu vyhledatelných dat.

I přes míru abstrakce je potřeba znát základní koncepty fulltextového vyhledávání. Fulltextové vyhledávání pracuje s tzv. analyzátory, kteří analyzují text a generují z něj index relevantních slov. Tento proces je aplikován na vyhledatelné dokumenty a uživatelské vyhledávací fráze. Vygenerované indexy se při vyhledávání porovnávají, a podle určitých kritérií databáze vyhodnotí relevantnost dokumentů pro vyhledávanou frázi. [60]

Každý analyzér se skládá z filtrů znaků, tokenizérů a filtrů tokenů. Filtry znaků v analyzovaném textu upravují jednotlivé znaky. Znaky mohou přidávat, odebírat nebo nahrazovat jinými (např.: odstranění diakritiky). Poté je takto upravený text předán tokenizérům, kteří text rozdělí na kolekci tokenů. Token může představovat cokoliv co dává v kontextu aplikace smysl, nicméně většinou přestavuje jedno slovo. Tyto tokeny můžou být ještě upraveny pomocí filtrů tokenů. Ty umožňují přidávat, upravovat a mazat tokeny. Konkrétními filtry mohou být: filtr pro odmazání generických slov bez přidaného významu (např.: spojky), filtr pro odstranění diakritiky nebo filtr pro převedení tokenů na malá písmena. Výsledné tokeny pak tvoří index, nad kterým již lze vyhledávat. [61] Pro tuto aplikaci byl sestaven analyzér skládající se primárně z tokenizéru rozdělující věty na slova a filtru přidávající dodatečné části původních slov pro širší podporu hledaných frází.

Elasticsearch pracuje s tzv. indexy, což jsou kolekce dokumentů entit stejného typu. Tyto dokumenty pak lze vyhledávat pomocí dotazovacího jazyka. Každý index má kromě samotných dat i konfiguraci, jak s daty pracovat. Kromě obecných konfiguračních hodnot obsahuje hlavně strukturu dokumentů, aby databáze byla schopna efektivně vyhledávat. Tato definice zahrnuje v první řadě určení datových typů jednotlivých atributů entit. Od datového typu se pak odvíjí další konfigurační hodnoty. Nejdůležitějším takovým nastavením pro fulltextové vyhledávání je specifikování analyzérů pro textové hodnoty. Specifikované analyzéry by měli odpovídat analyzéru použitému při vyhledávání.

Elasticsearch pro práci s těmito indexy a dokumenty poskytuje Java knihovnu reflektující jeho REST API, prostřednictvím které lze jednoduše nahrávat entity do indexů.

Hlavní řešené problémy spočívaly v přípravě entit a správě indexů. Tuto problematiku totiž Elasticsearch knihovna řeší ve formě dílčích operací, nikoliv však jako celek. Bylo potřeba vyřešit indexaci a vyhledávání karet a geografických míst. Karty musí být uživatel schopen vyhledat fulltextově skrze vícero atributů (název, popisek, odkazy...) a geografická místa podle lokace v mapě.

```
final SearchableCard cardToIndex = searchableCardFactory.create(card);
final IndexResponse cardResponse = esClient.index(i -> i
    .index(SearchableCard.INDEX_ALIAS)
    .id(String.valueOf(card.getId()))
    .document(cardToIndex));
```

Ukázka kódu 7: Indexace jedné karty do Elasticsearch databáze. Zdroj: [autor]

První jednodušší problematikou je příprava entit. Mohlo by se zdát, že tento problém nastane spíše v nestandardním projektu, avšak data využívaná k vyhledávání jsou trochu jiná než ta hlavní. V případě karet je nebytné komplexní strukturu dat značně zjednodušit, protože by jinak Elasticsearch nebyl schopen zanořená data správně analyzovat. Některé atributy je zase potřeba agregovat do jiné formy. V případě geografických míst je naproti tomu nutné zagregovat více různých dat, a paradoxně strukturu míst lehce zesložitit, aby obsahovala vše co uživatel potřebuje.

```
public class SearchableCard {
  String displayId;
  String avatarId;
  Set<CardLinkDto> links;
  CardOpeningHoursDto openingHours;

  String name;
  String website;
  String contactEmail;
  String contactPhone;
  String description;
  String crn;
  String jobPosition;
  Set<String> linkProfiles;
  Set<String> tags;
  Set<String> officeNames;
  /* ... */
}
```

Ukázka kódu 8: Indexovatelná karta. Obsahuje metadata a vyhledatelná data. Zdroj: [autor]

Na ukázce karty připravené pro indexaci je možné vidět právě zmiňované zjednodušení a agregaci dat. Třída obsahuje část nepostradatelných originálních dat a část agregovaných dat vnořených entit (z poboček jsou zde pouze jejich názvy). Ukázkovým příkladem agregace je pak kolekce profilových ID odkazů. Originální karta má totiž přístup pouze k cílovým URL adresám, a je tak na aplikaci, aby z těchto dat vytáhla ID profilů uživatelů. K tomu využívá šablony URL adres a standardizovaného formátu URL adres. Pomocí URL šablony je možné získat část URL adresy, která je specifická pro uživatele a obsahuje profilové ID. Tento řetězec je následně očištěn částmi URL adres, jako jsou: část cesty nebo dotazovací parametry.

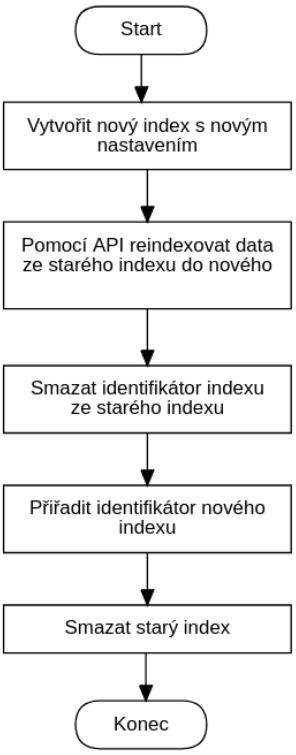
Tyto transformovací operace zajišťují speciální třídy zvané továrny, které vezmou originální entity a vytvoří z nich indexovatelné verze.

```
// sablona URL adresy
https://example.com/*/{userPart}
// URL adresa
https://example.com/profile/jan_novak?detail
// ID
jan_novak
```

Ukázka kódu 9: Ukázka transformace URL adresy na profilové ID. Zdroj: [autor]

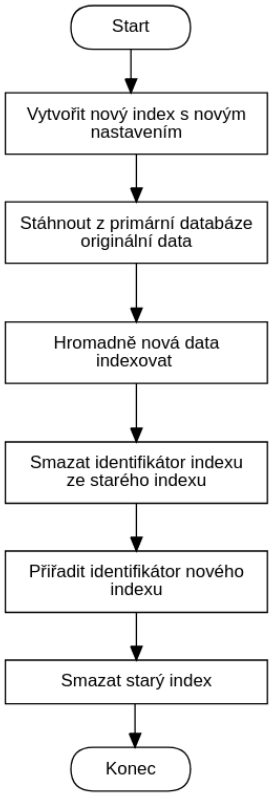
Dalším, již složitějším problémem, je změna struktury dat a změna konfigurace již existujících indexů. Je předpokládatelné, že entity se v průběhu vývoje budou měnit a rozšiřovat. Bohužel tím že Elasticsearch je spíše vyhledávacím strojem než klasickou databází, neposkytuje nástroje pro editaci dílčích dat, jako tomu je v případě SQL. Stejně tak má omezené nástroje pro úpravu konfigurace indexu, a není proto možné jednoduše změnit například použitý analyzátor. Naštěstí Elasticsearch nabízí alespoň možnost vzít dokumenty z jednoho indexu a nahrát je do jiného pomocí nové konfigurace. To ovšem stále neřeší stav, kdy se změní struktura entit nebo formát jednotlivých atributů. Bylo proto nezbytné vytvořit systém, který by zajistil reindexaci všech entit oběma způsoby. Oba způsoby mají podobnou základní strukturu, ale liší se v přípravě dat. Proces nejdřív vytvoří nový prázdný index s novým nastavením. Následně se některým ze způsobů indexují data do nového indexu. Po dokončené indexaci se přesune unifikovaný identifikátor indexu na nový index, a starý index se permanentně smaže.

Prvním jednodušším a rychlejším způsobem je reindexace na úrovni Elasticsearch. Pro tento případ má Elasticsearch připravené jednoduché API, v němž programátor pouze specifikuje starý a nový index, a vše ostatní už si zařídí sám Elasticsearch. Tento způsob však nenabízí možnost nahrání upravených dat entit, pouze změnu konfigurace indexu.



Obrázek 26: Proces hromadné reindexace dat na úrovni Elasticsearch. Zdroj: [autor]

Druhým složitějším a časově náročnějším způsobem, který umožňuje indexovat kompletně nová data, je reindexace dat z primární databáze. Tento způsob obsahuje hromadné stažení entit z primární databáze, jejich transformaci do indexovatelných entit a hromadnou indexaci pomocí API. Tento proces je zdlouhavý, právě kvůli nutnosti přesunutí dat mezi dvěma kompletně oddělenými systémy.



Obrázek 27: Proces hromadné reindexace nových dat z primární databáze. Zdroj: [autor]

Každý index má svou spravující třídu, která obsahuje napojení na korektní index v databázi a jeho nastavení.

7.6.8 Nastavení aplikace

Nedílnou součástí větší aplikace jsou rozdílná nastavení každého provozního prostředí. Nastavení každé aplikace je vysoce individuální a záleží čistě na požadavcích projektu. U většiny aplikací se nachází alespoň nastavení připojení k primární databázi, která je pro každého prostředí odlišná, aby nedocházelo k poškození dat (např.: mezi vývojovým a produkčním prostředím).

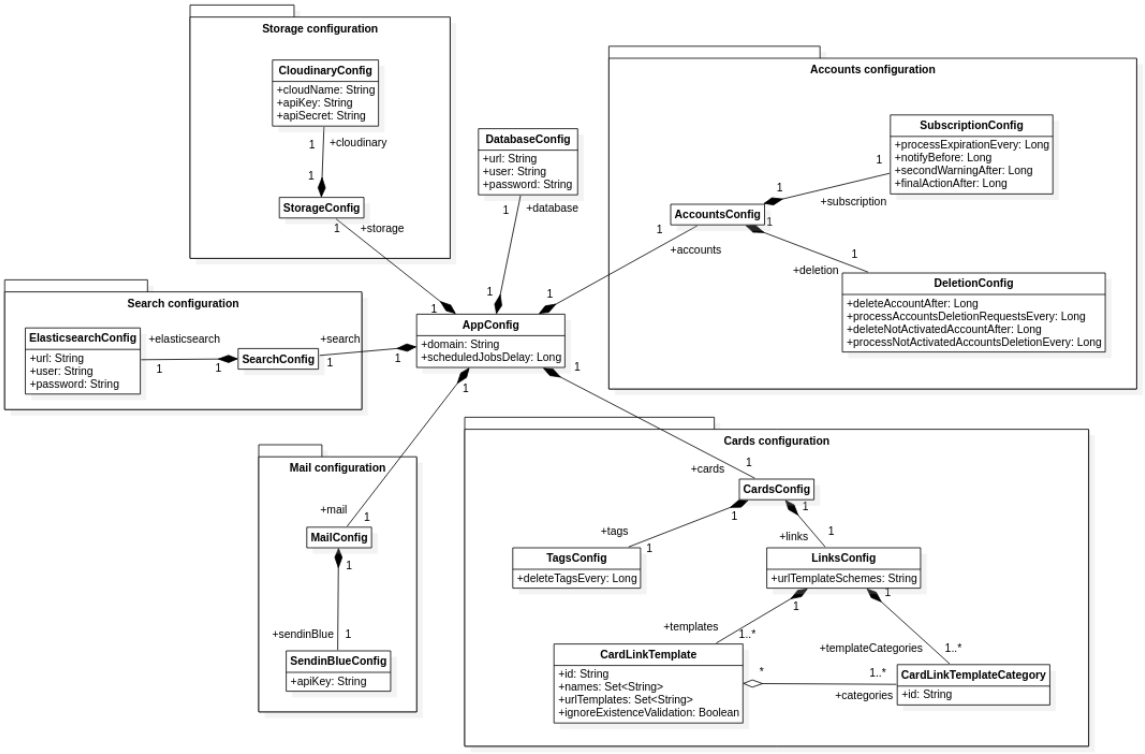
Tato nastavení musí být dostupná v aplikaci, a je tak nutné definovat jak se budou nastavení zapisovat, a jak se zpřístupní kódu. V případě Java aplikací se často využívá "properties"souborů, které obsahují nastavení v podobě klíč-hodnota. V modernějších aplikacích postavených na frameworku Spring se tento formát postupně nahrazuje flexibilnějším YAML (YAML Ain't Markup Language) jazykem, jenž umožňuje lépe definovat celé objekty nastavení.

```
accounts:
  deletion:
    process-accounts-deletion-requests-every: 300
    delete-account-after: 180
  subscription:
    process-expiration-every: 60
```

Ukázka kódu 10: Ukázka části nastavení aplikace v YAML konfiguračním souboru. Zdroj: [autor]

Spring tyto soubory umí zpracovat a poskytnout jednotlivé hodnoty atributů programátorovi. Nicméně používání tohoto způsobu pro získaní nastavení napříč aplikací je náchylné k chybovosti a zesložiťuje úpravu struktury nastavení.

Proto byla vytvořena tenká abstrakce nad Springovými nástroji vystavující jasně definovavé třídy a jejich atributy reflektující nastavení celé aplikace. Implementace zahrnovala připravení datového modelu a načítače hodnot z YAML souborů.



Obrázek 28: Datového model nastavení aplikace. Zdroj: [autor]

7.6.9 Šablony odkazů sítí

Aby bylo možné uživatelům poskytnou komfort v podobě předdefinovaných sociálních sítí společně s validací URL adres, bylo potřeba vytvořit robustní systém. Proto byl vytvořen systém šablon, v němž každá šablona definuje vlastnosti jedné sítě. Každý konkrétní odkaz pak má přiřazenou jednu šablonu, na základě které je možné adresu validovat a zobrazit

uživateli správnou ikonu a název sítě. Definice šablon jsou načítány z YAML souboru v rámci celého nastavení aplikace.

Šablony URL adres Šablony URL adres slouží pro specifikování vzoru povolených URL adres dané sítě. Tím je zaručeno, že odkaz vede skutečně na ověřenou síť. Zároveň to umožňuje získat ID profilů z URL adres.

S pomocí speciálních vyhrazených sekvencí znaků je možné definovat poměrně velké množství kombinací šablon. Šablony jsou při načtení zanalyzovány a jsou z nich vytvořeny Regex výrazy. Regex je výrazový jazyk skládající se ze speciálních znaků definující pravidla porovnávání [62]. Díky tomu je možné jednoduše zjišťovat, jestli textové řetězce vyhovují nějakému výrazu. Systém momentálně podporuje sekvence `*` a `{userPart}`. Sekvence `*` říká, že se na tomto místě může nacházet jakýkoliv znak, vyjma hostové části URL adresy, kde se mohou pod touto sekvencí skrývat pouze písmena a pomlčky. Sekvence `{userPart}` může obsahovat stejné znaky jako `*` sekvence, avšak textové řetězce označené touto sekvencí mají speciální význam. Nachází se v něm totiž právě zmiňované profilové ID. Díky tomuto označení je pak systém schopen určit, kde se profilové ID nachází a vytáhnou jej.

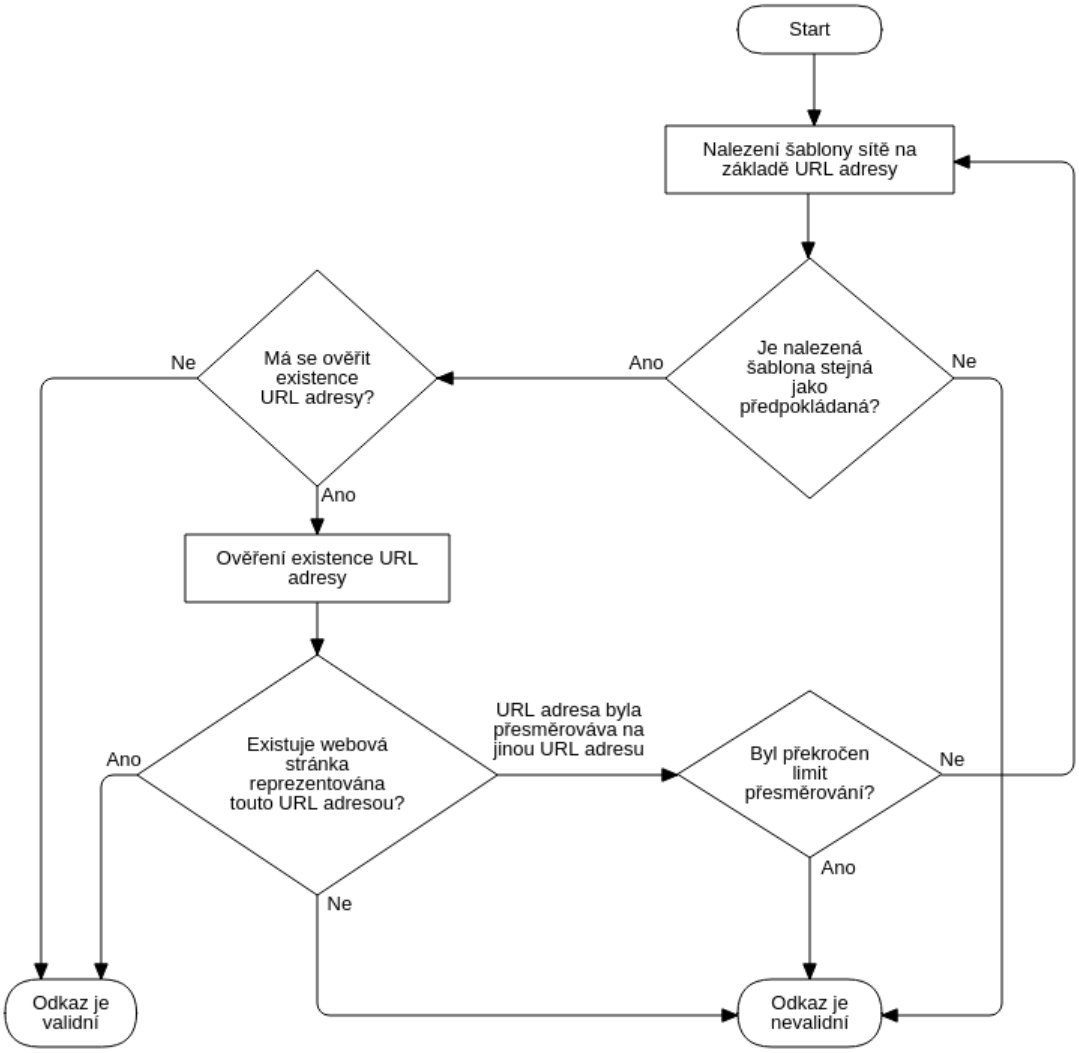
```
https://*.example.com/{userPart}
https://{userPart}.example.com
```

Ukázka kódu 11: Ukázka šablon URL adres. Zdroj: [autor]

Systém obsahuje různé validace aplikované při analyzování šablon, aby se do aplikace nemohli načíst nevalidní šablony. Momentálně definovanými pravidly URL šablon jsou následující:

- právě jedna sekvence `{userPart}`,
- oddělovač mezi jednotlivými sekvencemi,
- šablona je po záměně sekvencí náhodnými slovy validní URL adresa.

Validace odkazů Standardní validace se skládá ze dvou hlavních částí. První část řeší, jestli URL adresa odpovídá předpokládané URL šabloně. Předpokládaná šablona je určena nadřazenou skupinou odkazů. Validátor se pak pokusí pro předanou adresu nalézt existující šablonu. Pokud je tento předpoklad splněn, a zároveň se nemá validovat existence adresy, validace končí úspěchem. V případě vyžadované validace existence validátor přechází do druhé části. Druhá část se zabývá validací existence URL adresy, tedy, zda URL adresa odkazuje na nějakou existující webovou stránku. Toto se ověřuje simulováním HTTP požadavku webového prohlížeče, a z odpovědi se zjistí, že buď stránka existuje, stránka neexistuje nebo URL adresa přesměrovává na jinou stránku. V prvních dvou případech validace končí jasným výsledkem: validní, nevalidní. V tom posledním se proces opakuje s novou vrácenou URL adresou. I u této adresy se kontroluje šablona a existence. Takto se proces opakuje do té doby, než validátor nedostane jasnou odpověď existence, nebo vyprší interně stanovený limit přesměrování. Limit byl zaveden, aby nemohlo dojít k potencionálnímu zacyklení validace, je ale nastaven dostatečně vysoký, aby nedocházelo k předčasným selhání.



Obrázek 29: Proces validace URL adresy odkazu. Zdroj: [autor]

Bohužel ne u všech URL adres lze validovat existenci. Některé stránky mají pokročilou ochranu proti škodlivým strojovým botům, které tento validátor označí za škodlivého bota. Byly provedeny optimalizace v podobě úpravy HTTP hlaviček, aby se validace tvářila jako webový prohlížeč, a u některých stránek to skutečně pomohlo, jiné jsou ale pečlivější. Proto se zavedl příznak přímo do definice šablon, který tuto validaci existence vypíná. Naštěstí se jedná o nižší procenta stránek, a pro bezpečnost uživatelů je stěžejní pouze první část validace.

7.6.10 Prémiový plán a předplatné

Proces aktivace prémiového plánu začíná aktivováním platného kupónu. Na základě této aktivace je uživateli přiřazena speciální role a vytvořen nový popisný objekt předplatného obsahující datum expirace. Datum expirace je vypočítán z doby platnosti kupónu. Každý kupón definuje, po jakou dobu bude předplatné aktivní, a při aktivaci se tato doba přičte k aktuálnímu datu. V tuto chvíli uživatel může využívat všechny prémiové funkce. Pokud

by v tuto chvíli uživatel aplikoval další kupón, datum expirace by se změnil podle nového kupónu.

V případě placeného předplatného by objekt předplatného zprvu datum expirace neměl. Ta by se nastavila automaticky, až při neprovedené následující platbě.

Systém tedy při vytváření a ukládání karet musí vždy validovat zda daná karta neobsahuje prémiový obsah. Pokud ano, musí zkontrolovat, zdali uživatel má příslušnou roli. V případě, že by neměl, je celý požadavek zaříznut a uživateli je vrácena chyba.

Nejkomplikovanější částí celého systému je expirace předplatného. Problém spočívá v existenci prémiového obsahu. Pokud uživatel v době expirace nemá na svém účtu prémiový obsah, je situace jednoduchá a předplatné se zruší okamžitě. Pokud však uživatel v době expirace má na svém účtu prémiový obsah, je potřeba s těmito daty nějak naložit. Uživatel tak dostane tři možnosti. Tou nejjednodušší je upozornění ignorovat, a v takovém případě se po nějaké době všechny karty uživatele skryjí a veřejnost se na ně už nepodívá. Druhou možností je prémiový obsah pomocí speciálního formuláře odstranit, aby bylo možné zrušit předplatné. Poslední možností je prodloužení předplatného. V tomto případě uživateli všechen obsah zůstane a může prémiové funkce nadále využívat.

Celý proces začíná nějakou dobou před expirací. Uživatel je dopředu emailem upozorněn na blížící se expiraci. Pokud uživatel do té doby nesmaže všechen prémiový obsah, v den expirace přijde druhý email s informací o proběhlé expiraci, a možnostmi, které uživatel může provést. Pokud bude uživatel ignorovat toto upozornění, během několika dní mu přijde druhé upozornění. V případě, že uživatel ignoruje i toto poslední varování, všechna data uživatele jsou během několika dní automaticky skryta. Uživateli už jen o skrytí dat přijde informativní email.

Zvolený přístup se jeví jako dostatečně flexibilní a pokrývá všemožné scénáře požadované aplikací.

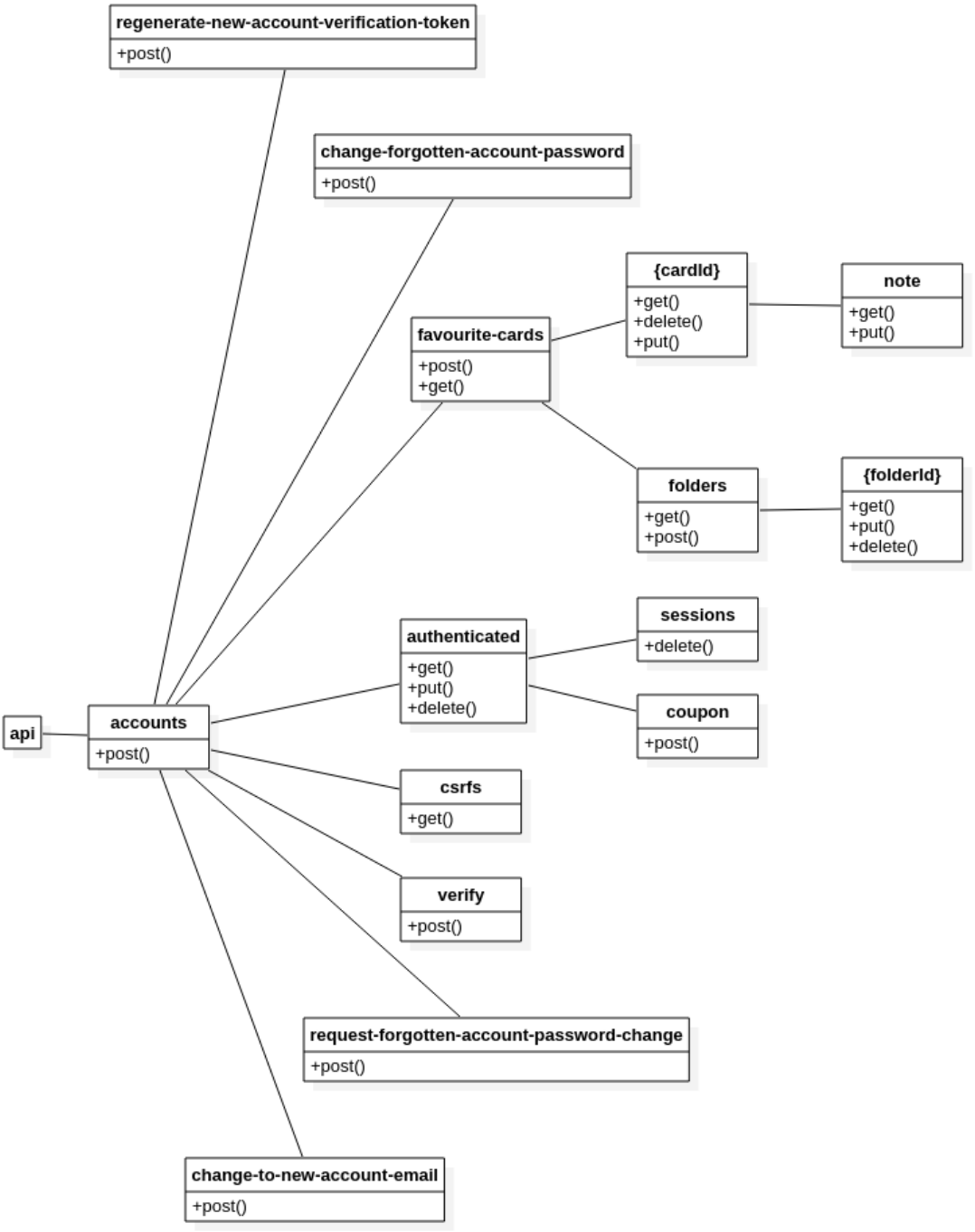
7.6.11 REST API

K zajištění komunikace mezi front-endovou GUI aplikací a aplikací API serveru, server vystavuje právě REST API. Díky němu je možné vytvářet, ukládat a získávat entity, a spouštět různé akce.

REST API obecně pracuje s tzv. zdroji. Každý zdroj představuje jeden typ pojmenovatelné informace. Tím může být entita, obrázek, dokument, nebo třeba kolekce. Jednotlivé zdroje jsou pak vystaveny na přesně daných URL adresách, a pomocí HTTP metod (GET, POST, PUT...) je možné nad těmito zdroji provádět operace. Operace jsou většinou následující: získání konkrétního zdroje nebo jeho kolekce, vytvoření nového zdroje nebo úprava již existujícího. Nejrozšířenějším jazykem pro popis zdrojů je JSON, díky kterému je možné objekty aplikace jednoduše mapovat na JSON dokumenty. [50]

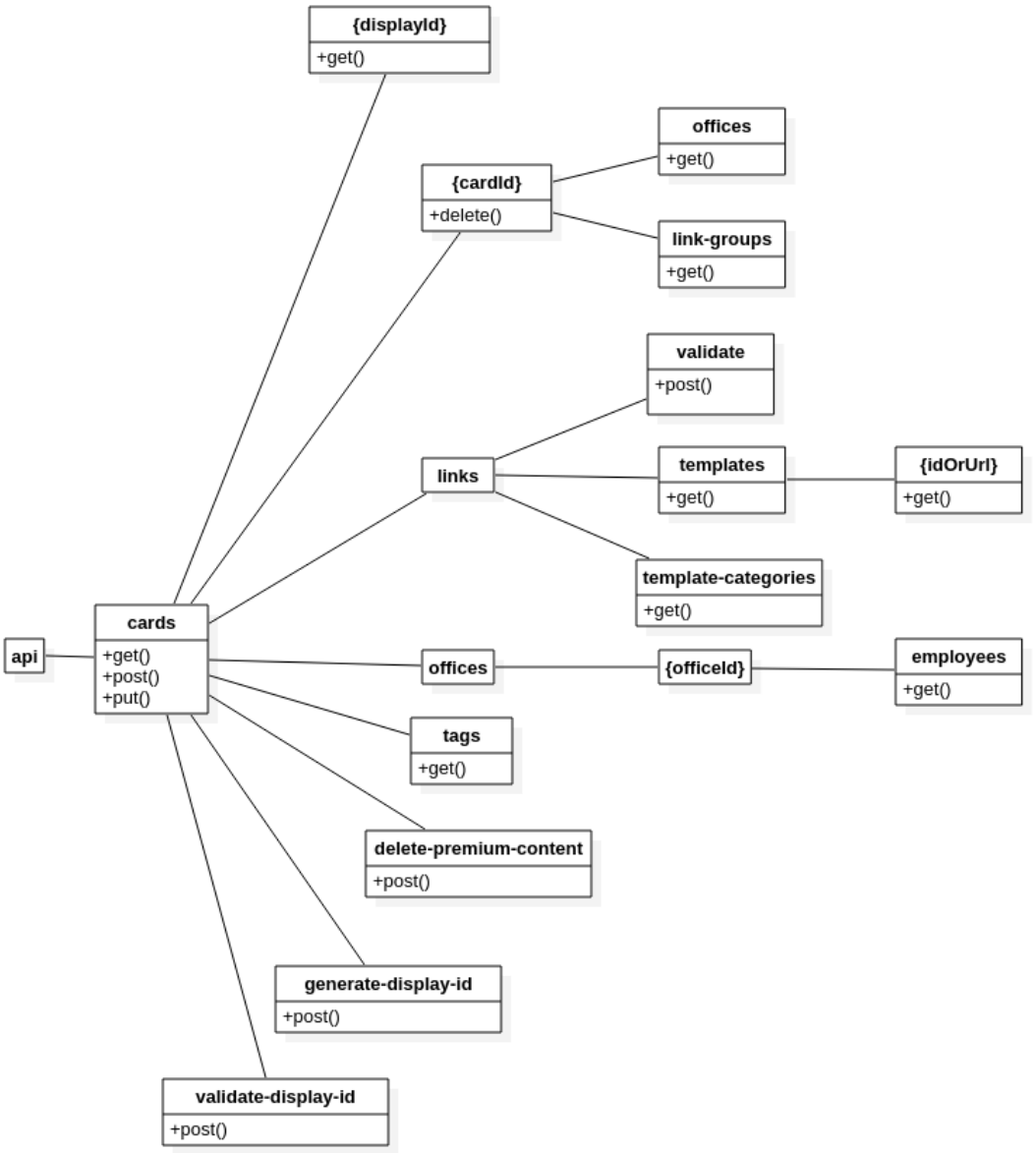
Struktura API Implementované API v běžných scénářích (získání, vytvoření a úprava zdrojů) respektuje doporučovaná pravidla. Nicméně v krajních případech je potřeba provádět i jiné, většinou servisní, operace, které tyto pravidla porušují. Toto je však poměrně běžná praxe, a pokud takto není navržena větší část API, nepředstavuje to velký problém.

Jak je možné vidět níže, API pro uživatelské účty zahrnuje manipulaci s účtem přihlášeného uživatele a manipulaci s oblíbenými kartami.



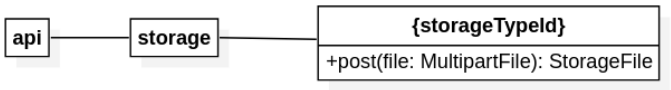
Obrázek 30: Stromový přehled struktury API uživatelských účtů. Zdroj: [autor]

Nejpodstatnější API se zabývá manipulací karet, a zároveň umožňuje získávat i další podřazené entity. Mimo to nabízí i pomocné operace pro manipulaci s display ID.



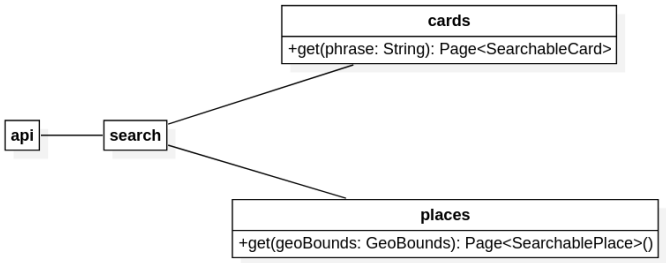
Obrázek 31: Stromový přehled struktury API karet. Zdroj: [autor]

API úložiště momentálně umožňuje pouze soubory nahrávat, protože systém pracuje pouze s veřejnými soubory, které lze získat přímo z Cloudinary serverů.



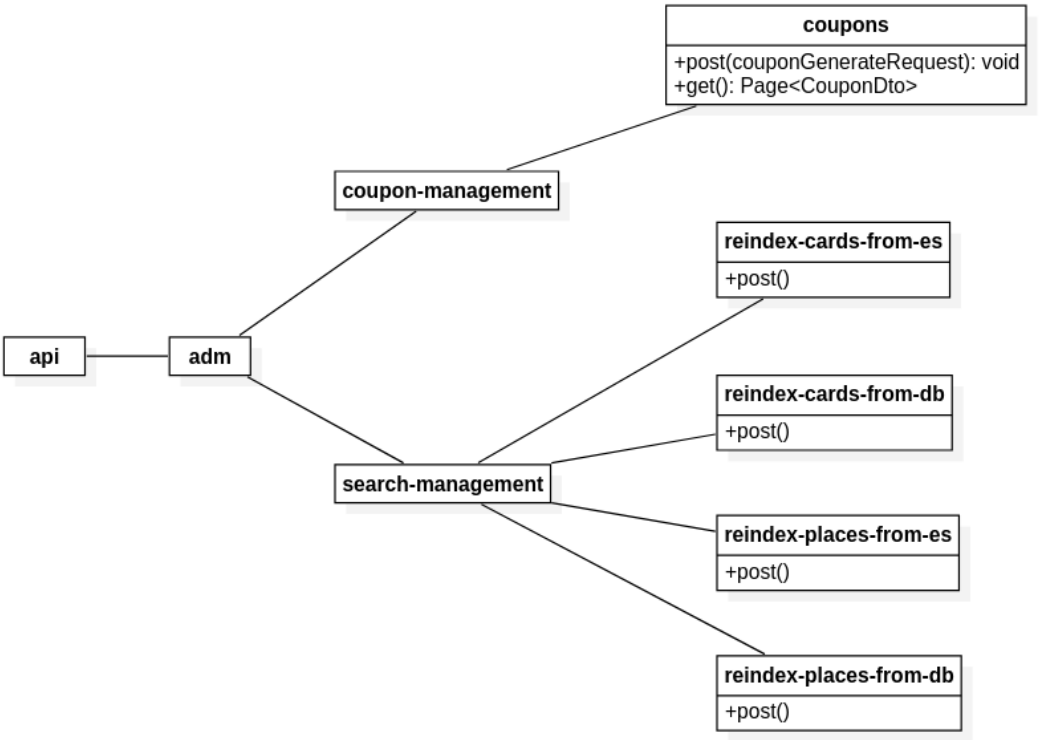
Obrázek 32: Stromový přehled struktury API uložiště. Zdroj: [autor]

Pro přehlednost vyhledávání karet a míst existuje separátní vyhledávací API.



Obrázek 33: Stromový přehled struktury API vyhledávání karet. Zdroj: [autor]

Aplikace také nabízí interní administraci pro správce. Proto vzniklo i API pro spouštění servisních operací a k správě zákulisních zdrojů (např.: kupóny).



Obrázek 34: Stromový přehled struktury API operací administrace. Zdroj: [autor]

Objekty pro přenos dat V rámci komunikace mezi konzumentem API a serverem je nezbytné přenášet objekty reprezentující data aplikace. K tomu by šlo využít přímo doménových tříd. Bohužel některé tyto třídy obsahují data, která by se neměla dostat mimo

server. Nejtypičtějšími takovými daty jsou hesla uživatelů. Někdy je také nezbytné poskytnout agregované atributy, které však nemají své místo v originálních datech. Právě z těchto důvodů API využívá tzv. DTO (Data transfer objekt). Tyto datové objekty slouží obecně pro přenos daty mezi procesy. V tomto případě přenáší data mezi API serverem a front-endovou aplikací. API tak na venek pracuje pouze s DTO objekty, které reflektují doménové objekty. Speciálním případem jsou DTO objekty jenž slouží pouze pro jednorázové předání dat. Většinou se používají pro specifikování požadavků front-endové aplikace na zvolenou operaci, nebo jako objekty nesoucí výsledek nějaké operace.

Pro transformaci mezi doménovými objekty a DTO objekty vznikla jednoduchá podpora pro standardizovanou a centralizovanou transformaci. Slouží k tomu továrnové třídy dědící z obecného rozhraní, v němž se nacházejí metody pro transformaci z doménového objektu na DTO objekt a naopak.

```
public interface DtoFactory<E, D> {
  D createDto(E entity);
  E updateEntity(E originalEntity, D dto);
}
```

Ukázka kódu 12: Obecné rozhraní třídy pro transformaci mezi doménovými a DTO objekty. Zdroj: [autor]

Transformace z DTO objektu na doménový objekt je specifická v tom, že pro transformaci používá zároveň i originální doménový objekt. Díky tomu je možné zajistit, že výsledný objekt bude mít změněné pouze ty atributy, u nichž je to povolené. Rozhraní továrny je generické, a proto pro každá doménová entita má vlastní implementaci.

Zabezpečení Zabezpečení je velmi důležitou součástí aplikace pro znemožnění zneužití dat útočníky. Je nezbytné zajistit, aby se neoprávnění uživatelé nedostali k datům, které jim nepřísluší, a aby je nemohli poškodit. Těmito problémy se zabývá především společnost OWASP poskytující kvalitní materiály o bezpečnosti programátorům. Avšak řešit všechny existující problémy svépomocí by zabralo spoustu času, a bez dlouholetých znalostí v oblasti bezpečnosti, by implementovaný systém nemusel být ani správně zabezpečený.

Naštěstí framework Spring zahrnuje i knihovnu Spring Security. Tato knihovna jednak automaticky zajišťuje doporučené bezpečnostní praktiky, a jednak poskytuje programátorovi nástroje pro tvorbu přihlašování a práv uživatelů. Programátor si navíc tyto nástroje může relativně snadno rozšířit nebo implementovat vlastní.

Specificky pro tuto aplikaci bylo definováno jaké URL adresy API vyžadují přihlášení a specifické role, a výchozí přihlašovací proces byl rozšířen o externí poskytovatele a vlastní správu účtů.

Přihlašování Korektní implementace procesu přihlašování je pro aplikaci využívající uživatelské účty stěžejní. Zabraňuje neoprávněným uživatelům vydávat se za jiné uživatele. Standardním, dlouhá léta využívaným, přístupem je ověření kombinace emailové adresy-/uživatelského jména a hesla. U tohoto přístupu kombinaci ověřuje přímo aplikace. Druhým, stále více používaným, přístupem je přihlašování pomocí třetích stran. Při tomto přístupu aplikace deleguje přihlašování na jiného poskytovatele uživatelských účtů, který následně vrátí aplikaci informace o přihlášeném uživateli. Aplikace se tak nemusí starat o ověřování hesla a podobně. Takovýchto poskytovatelů existuje celá řada, což přispělo

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http
       .mvcMatchers(HttpMethod.GET, "/api/search/cards").permitAll()
       .mvcMatchers(HttpMethod.GET, "/api/search/places").permitAll()
       .mvcMatchers("/api/adm/**").hasRole("ADMIN")
       .mvcMatchers("/api/**").authenticated()
       .anyRequest().denyAll()
       /* ... */
}
```

Ukázka kódu 13: Ukázka části nastavení zabezpečení API pomocí knihovny Spring Security. Zdroj: [autor]

vzniku standardu OAuth2, který standardizuje celý proces komunikace mezi aplikací a poskytovatelem. Díky tomu není potřeba pro každého poskytovatele implementovat odlišný proces.

Aby měli uživatelé volnost při výběru, jakým způsobem se chtějí přihlašovat, byly do této aplikace implementovány oba způsoby. Uživatel se tak může přihlásit a zaregistrovat, buď pomocí běžné kombinace emailové adresy a hesla, nebo pomocí některých z poskytovatelů. Momentálně jsou připraveni nejvyužívanější poskytovatelé Google, GitHub a Facebook, nicméně rozšíření o další poskytovatele nepředstavuje příliš práce. Zaregistrovaný uživatel si pak může ke svému účtu připojit i další způsoby, tedy, pokud se zaregistroval emailovou adresou, může si připojit některého z poskytovatelů a obráceně. Díky tomu si uživatel může při každém přihlášení vybrat, jakým způsobem se přihlásí.

Druhým řešeným problémem je, jak verifikovat přihlášeného uživatele při každém následujícím HTTP požadavku na server, aniž by musel uživatel pokaždé zadávat přihlašovací údaje. Aplikace na serveru může buď udržovat v paměti sezení každého uživatele nebo vygenerovat uživateli token obsahující informace o uživateli.

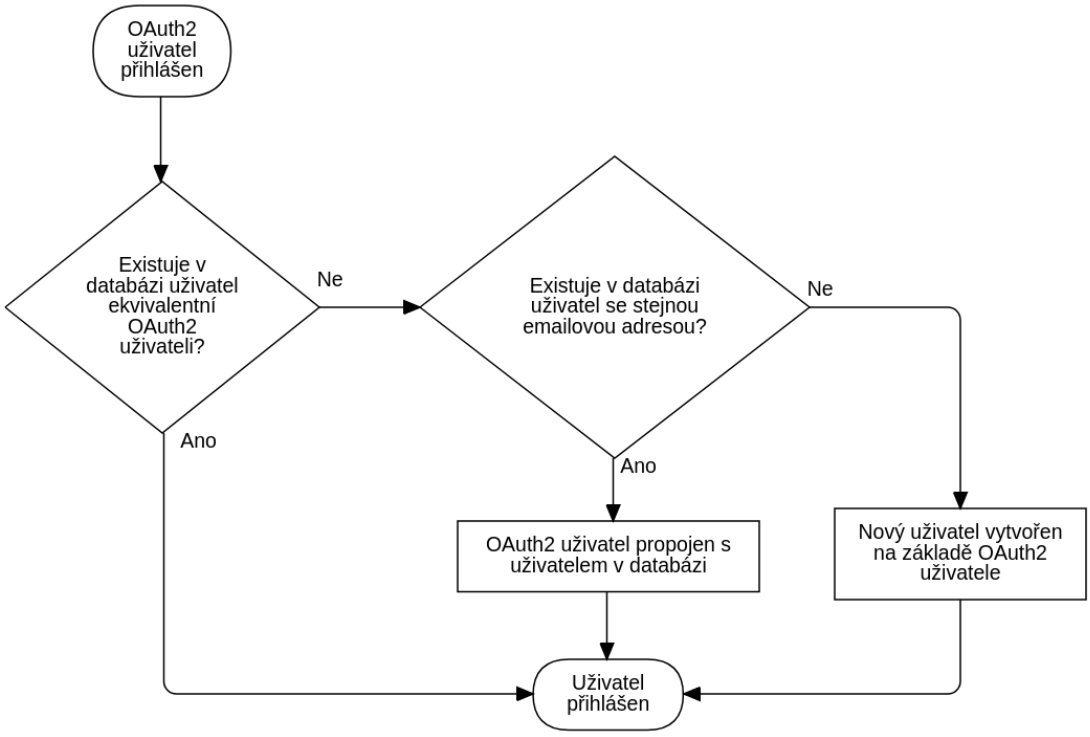
V případě, že aplikace využívá sezení, je na serveru spravován objekt s informacemi o přihlášeném uživateli. Uživatel pomocí webového prohlížeče pak pouze předává ID daného sezení. Server má tak plnou kontrolu nad daným sezením. Bohužel server musí zajistit opatření proti krádeži těchto ID.

Alternativou je vygenerování tokenu obsahující uživatelské informace. Tento token je následně předán klientovi, a ten musí zaručit předání vygenerovaného tokenu severu. Tento způsob je převážně využíván pro komunikaci mezi servery, avšak začíná se používat i v aplikacích, kde je aplikace rozdělena na GUI část a API část. Díky tomuto přístupu server nemusí udržovat sezení přihlášených uživatelů. Tím že server nemá token pod svojí správou, není však snadné takový token označit jako neplatný, třeba v případě odhlášení. To jde řešit například ukládáním tokenů do databáze, nicméně správa tokenů se pak částečně přenáší zpět na server, a výhody nezávislosti na serveru mizí. Řešení se tak přibližuje přístupu se sezeními, avšak za cenu složitější implementace.

V případě této aplikace proběhl experiment s druhým typem ověřování, posléze se ale ukázalo, že řešení je zbytečně složité a ne úplně spolehlivé. Přihlašování bylo tedy přepracováno na řešení se sezeními, které má již knihovna Spring Security v základu vyřešené, což neplatilo o předchozím přístupu. Díky tomu se celý proces zjednodušil a delegoval se na externí knihovnu, na které pracují experti v oboru. Bonusem bylo zjednodušení implemen-

tace přihlašování pomocí poskytovatelů třetích stran, protože pro tyto scénáře má knihovna Spring Security připravenou podporu.

Jediným problémem bylo spojení interních uživatelských účtů a OAuth2 účtů. Spring Security totiž pro každý způsob využívá jiné objekty. Proto byl proces připravený knihovnou rozšířen o vlastní zakončení. Toto zakončení je vyvoláno knihovnou po úspěšném přihlášení poskytovatelem, a dává možnost provést vlastní operace nad přihlášeným uživatelem než se odpověď dostane ke koncovému uživateli. Proces řeší i registraci nového uživatele, protože OAuth2 nerozlišuje mezi novým a existujícím uživatelem. OAuth2 pouze zajišťuje poskytnutí uživatelských dat.



Obrázek 35: Proces zpracování přihlášeného OAuth2 uživatele a transformace na interního uživatele. Zdroj: [autor]

7.6.12 Automatizované testování

Automatizované testování umožňuje kód aplikace otestovat ještě před samotným spuštěním a testovat kód průběžně při úpravách a rozšiřováních. Díky tomuto testování je možné předejít velkému množství logických chyb, již v počátku implementace.

Způsobů testování je více. Tato aplikace využívá unit testování a integračního testování. Unit testování izolovaně ověřuje předpokládané chování komponent systému bez vlivu ostatních komponent [63]. Opakem je integrační testování, jenž ověřuje souhru více komponent najednou [64].

Služební třídy a pomocné třídy provádějící nějaké transformace (DTO továrny, analyzátor šablon odkazů...) jsou testovány izolovaně pomocí unit testů. Pro každou metodu

každé třídy je připraveno několik testů ověřující různé scénáře: běžné chování při různých datech, chování při chybných datech a podobně. Hlavní knihovnou zpřístupňující nástroje pro spouštění testů a ověřování výsledků je JUnit. Aby bylo možné třídy testovat izolovaně, je potřeba simulovat chování ostatních komponent, se kterými testovaná třída komunikuje. K tomu je využita knihovna Mockito, která zároveň umožňuje zkoumat proběhlé interakce, a na základě toho vyhodnocovat testy.

```
@Test
public void getMostUsedTags_ExpectedFoundCorrectOnes() {
  when(cardTagDaoMock.findMostUsed(3)).thenReturn(Set.of(
      new CardTag(null, "dev"),
      new CardTag(null, "nature"),
      new CardTag(null, "tech")
  ));
  assertEquals(
      Set.of("dev", "nature", "tech"),
      cardTagService.getMostUsedTags(3)
  );
}
```

Ukázka kódu 14: Testu ověřující, že služba štítků vrací správné nejvyužívanější štítky z nasimulovaného repozitáře. Zdroj: [autor]

Třídy repozitářů a třídy poskytující REST API jsou spíše integračními testy spojujícími více komponent k zajištění výsledků.

Při testování repozitářů je potřeba testovat SQL dotazy oproti reálné databázi s daty. K spuštění takových testů je tedy potřeba běžící testovací databáze s připravenými testovacími daty. Ta jsou do databáze nahrávána před každým testem a mazána po každém testu. Tím je zaručena izolovanost jednotlivých testů, aby nedocházelo k mylným výsledkům.

Testy REST API jsou kombinací integračních a unit testů. Knihovna Spring poskytuje nástroje pro volání API skrze URL adresy, které simulují reálnou komunikaci s API. Napojení na služební třídy je ale řešeno simulací, aby nebylo nutné zprovozňovat všechny části aplikace.

7.7 Implementace front-endové aplikace

Front-endová aplikace zpřístupňující GUI koncovým uživatelům byla postavena na frameworku Nuxt obalující knihovnu Vue. Díky tomu je zajištěna reaktivnost komponent a prvotní vykreslení stránek na serveru. Zaměření tak padlo na konkrétní problémy řešené domény.

7.7.1 Komunikace s API serverem

Prvním stavebním kamenem je možnost komunikovat s vyvinutým API. Opět existuje spousta ověřených knihoven řešící problémy komunikace. Nuxt přichází se svou vlastní, která má jednoduché API, a proto byla zvolena. Díky integraci s Nuxt frameworkem bylo možné jednoduše centralizovaně přednastavit knihovnu pro všechna volání API. Hlavní takovou konfigurací byla správa chyb z volání API. Ta řeší obecné chyby, jako je nepřihlášený uživatel nebo nedostatečná práva uživatele. Zároveň řeší, aby v každém požadavku byl za-

```
@Test
@WithMockUser
public void validateLink_InvalidLink_ExpectedFalse() {
  /* ... */
  mockMvc.perform(post("/api/cards/links/validate")
          .accept(MediaType.APPLICATION_JSON)
          .contentType(MediaType.APPLICATION_JSON)
          .content(objectMapper.writeValueAsString(requestBody))
          .with(csrf()))
      .andDo(print())
      .andExpect(status().isOk())
      .andExpect(content().contentType(MediaType.APPLICATION_JSON));
}
```

Ukázka kódu 15: Ukázka testu volání API validující odkaz. Zdroj: [autor]

hrnut platný CSRF token. CSRF token zabraňuje útočníkům provádět operace pod účtem jiného uživatele pomocí podvodných stránek [65]. Token je poskytován API serverem na vyžádání. Front-endová aplikace však musí zaručit jeho získání a následné uložení pro další požadavky. Předání tokenu zpět API serveru je zajištěno přidáním tokenu před odesláním HTTP požadavku do speciální hlavičky. Pokud token v aplikaci ještě neexistuje, je odeslán paralelní požadavek na API server pro vygenerování tokenu.

```
$http.onRequest(async (request) => {
  if (!csrfAllowedMethods.includes(request.method)) {
    await addCsrfTokenToRequest(store, $http, request)
  }
})
```

Ukázka kódu 16: Vkládání CSRF tokenu do každého požadavku putujícího na API server. Zdroj: [autor]

7.7.2 Centrální zdroj informací

Dalším důležitým stavebním prvkem je tzv. centrální zdroj informací. Ten představuje centralizované místo pro ukládání a získávání dočasných dat napříč celou aplikací. Je zde možné například uložit informace o přihlášeném uživateli nebo výše zmíněný CSRF token. K tomu slouží knihovna Vuex řešící uchování dat a záznam změn v čase. Díky ní je pak například uživatelský účet jednoduše přístupný ze všech komponent bez nutnosti opakovaného provolávání API.

7.7.3 Lokalizace textů

Lokalizace textů se zabývá zobrazováním textů prvků UI v konkrétním jazyce uživatele. Díky tomu různí uživatelé mluvící odlišnými jazyky mohou bez problému aplikaci využívat v jejich nativním jazyce.

Tuto problematiku řeší například knihovna i18n od tvůrců frameworku Nuxt. Konkrétně zajišťuje nahrazování klíčů konkrétními texty v komponentách a automatické přepínání použitého jazyka podle webové prohlížeče.

Problém ale nastal při tvorbě informačních stránek (nápověda, podmínky použití...). U těchto stránek je obsah zapsán jako jeden velký HTML soubor pro podporu formátování, nikoli jako dílčí texty. Knihovna však očekává všechny texty zapsané v jednom JSON dokumentu. Do těchto dokumentů bylo potřeba nějakým způsobem vložit i ty HTML dokumenty. Z tohoto důvodu vznikl pro každý jazyk JS soubor, který posbírá všechny překladové materiály a poskládá z nich výsledný JSON dokument.

7.7.4 Validace formulářových polí

Validace uživatelsky vložených dat je důležitá především pro zaručení správnosti uložených dat. Ne všichni uživatelé jsou však podvodníci chtějící poškodit data. Většina uživatelů zadává očekávaná data, nicméně je velká pravděpodobnost, že se nějaký uživatel omylem upíše a udělá nechtěnou chybu. Proto je dobré v rámci kvalitního UX poskytnout uživatelům o chybně zadaných datech rychlou a cílenou zpětnou vazbu.

Obrázek 36: Ukázka pole formuláře s textem v nevalidním formátu. Zdroj: [autor]

Pomocí knihovny Vuelidate bylo implementováno zobrazování chyb přímo v konkrétních polích hned, jakmile uživatel vyplní dané pole. Knihovna řeší celý mechanismus validace hodnot polí a poskytování výpisu všech chyb ve formuláři. Zároveň přichází s obecnými validátory, ale programátor může implementovat i vlastní. Toho bylo využito především u validace URL adres odkazů na sociální sítě, u nichž je potřeba se dotázat API, které adresy zvaliduje.

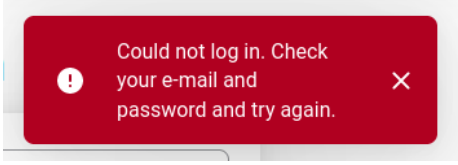
```
return async function (displayId) {
  const requestBody = { cardId, displayId }
  try {
    return this.$http.$post('cards/validate-display-id', requestBody).valid
  } catch (e) {
    this.$notifyError()
  }
}
```

Ukázka kódu 17: Vlastního validátor využívající API k validaci hodnoty. Zdroj: [autor]

S připravenými dílčími validátory je pak možné definovat pravidla pro jednotlivá pole ve formulářích. Tato pravidla jsou následně aplikována na jednotlivá pole, která sledují změny hodnot. Díky reaktivnosti Vue je pak možné chybu zobrazit hned, jakmile vznikne. Při odeslání formuláře je validace provedena ještě jednou, a buď je formulář odeslán nebo je zobrazena chyba. V případě běžných uživatelů by se tedy chybná data na server vůbec neměla dostat.

7.7.5 Upozornění v UI

Pokud v aplikaci vznikne nějaká událost (chyba, úspěšnost operace...), je nutné o tom informovat uživatele. To je v tomto případě vyřešeno vyskakujícími upozorněnými po vzoru mobilních operačních systémů. Tento způsob je výhodný v tom, že uživatel si upozornění všimne ať už je kdekoliv na stránce, i při přechodu mezi modálními okny.



Obrázek 37: Ukázka upozornění na chybu vzniklou špatnými přihlašovacími údaji. Zdroj: [autor]

K implementaci upozornění byla využita knihovna Vue.js notifications poskytující dostatečnou volnost přizpůsobení vzhledu a chování, oproti ostatním podobným knihovnám. Je navíc na míru tvořena pro knihovnou Vue. Knihovna poskytuje jednoduché API pro vystavování upozornění pomocí specifikování typu zprávy (chyba, úspěch, info...) a samotné zprávy. Pro velkou aplikaci, kde je nutné na spoustě místech taková upozornění vystavovat, je nevhodné pokaždé duplikovat celou specifikaci upozornění. Stejně jako komponenty i upozornění využívají překladových textů. Tato kombinace tak značně zvyšuje riziko chybovosti kódu. Vznikla proto jednoduchá abstrakce zpřístupňující pomocné metody, které jsou dostupné celou aplikací. Pro každý používaný typ zprávy (chyba, úspěch, info) existuje právě jedna metoda vyžadující pouze klíč překladového textu. Malou výjimkou jsou chybové zprávy, kde existují dvě metody: jedna s možností vložení vlastního textu, druhá zobrazující generickou chybu.

```
this.$notifySuccess('general.success.added')
this.$notifyInfo('general.info.linkCopied')
this.$notifyError('general.error.invalidData')
this.$notifyError()
```

Ukázka kódu 18: Pomocné metody pro vystavení různých typů upozornění. Zdroj: [autor]

7.8 Příprava pro produkční provoz

Tím že celá aplikace poběží v orchestračním systému Kubernetes, je nutné zprovoznit obě aplikace v Docker kontejnerech. Následně nakonfigurovat Kubernetes, aby sám aplikace spustil a navázal mezi nimi spojení.

7.8.1 Příprava Docker image

Docker image je šablona, podle které se automatizovaně vytvářejí běžící kontejnery s aplikacemi. Pro API server aplikaci a front-endovou aplikaci je nutné tedy připravit separátní Docker image. K tomu poslouží speciální soubor zvaný Dockerfile, sloužící jako předloha pro vytvoření Docker image. Dockerfile obsahuje všechny příkazy potřebné pro spuštění aplikace [66].

Součástí nadstavby Spring frameworku Spring Boot je možné automatizovaně Docker image vytvořit bez manuální tvorby Dockerfilu. Spring Boot tedy zajistí vytvoření optimalizovaného Docker image celé aplikace.

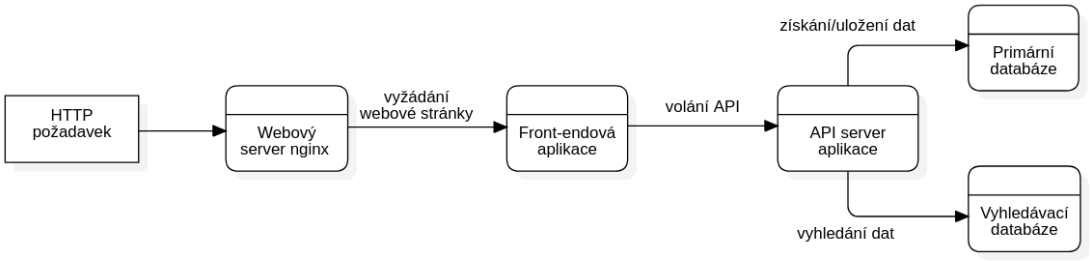
V případě front-endové aplikace tato podpora není a bylo nutné Dockerfile připravit ručně. Nicméně příprava front-endové aplikace pro provoz v Docker kontejneru byla jednodušší, než by byla příprava API serverové aplikace. Dockerfile obsahuje příkazy podobné tomu, jak se aplikace spouští pro lokální vývoj. Dockerfile nejdříve vytvoří cílový adresář pro aplikaci, následně nainstaluje potřebné systémové aplikace pro sestavení a běh aplikace. Nakonec aplikaci sestaví do vytvořeného adresáře a spustí na portu `3000`.

```
FROM node:14.17.6-alpine

RUN mkdir -p /usr/src/quickcards-web-app
WORKDIR /usr/src/quickcards-web-app

RUN apk update && apk upgrade
RUN apk add git

COPY . /usr/src/quickcards-web-app/
RUN npm install yarn
RUN yarn install
RUN yarn build

EXPOSE 3000

ENV NUXT_HOST=0.0.0.0
ENV NUXT_PORT=3000

ENTRYPOINT yarn start --dotenv=.env.$ENV
```

Ukázka kódu 19: Dockerfile spouštějící front-endovou aplikaci v Dockeru. Zdroj: [autor]

7.8.2 Nastavení Kubernetes

Na základě připravených Docker imagí pro obě aplikace byl zprovozněn systém Kubernetes. V systému se kromě implementovaných aplikací nachází i webový server nginx a databázový systém Elasticsearch. Jediný databázový systém PostgreSQL je provozován samostatně poskytovatelem DigitalOcean, není tak nutné se starat o ruční zálohovaní nebo výpadky.



Obrázek 38: Putování HTTP požadavku infrastrukturou. Zdroj: [autor]

Webový server slouží jako vstupní brána do celého systému a poskytuje nástroje pro zabezpečení požadavků. Zároveň zahazuje neplatné požadavky ještě dřív, než by mohli způsobit škody v samotných aplikacích. Následně platné požadavky deleguje na serverovou část front-endové aplikace s požadavkem na poskytnutí webových stránek. V případě, že jsou vyžadována data z API serveru, front-endová aplikace část požadavku deleguje právě na API. API server si navíc data může vyžádat z primární nebo vyhledávací databáze. Zpracovaný požadavek se nakonec vrací uživateli, opět skrze nginx.

Obě aplikace a server nginx jsou nakonfigurované jako Deploymenty, jediný Elasticsearch je nastaven jako StatefulSet. Deployment je entita Kubernetes systému popisující, jak bude aplikace spuštěna [67]. Deploymenty se používají v případě, kdy aplikace neudržuje důležitá data, o která by se v případě vypnutí přišlo [67]. StatefulSety jsou jakousi nadstavbou nad Deploymenty, a zaručují udržení stavů spuštěných aplikací. [68]. Právě z těchto důvodů mohou být implementované aplikace spouštěny jednoduššími Deploymenty, protože všechna důležitá data ukládají do PostgreSQL. Elasticsearch databáze je naproti tomu sama zdrojem dat a není vhodné, aby docházelo k jejímu vypínání.

O finální spuštění takto nakonfigurovaného systému a vytvoření SSL certifikátů se stará poskytovatel; v tomto případě DigitalOcean. V tomto momentě jsou obě aplikace spuštěny a připraveny zpracovávat uživatelské požadavky.

8 Závěr

Tato bakalářské práce se zabývala vývojem webové aplikace pro tvorbu, správu, zobrazení a sdílení vstupních stránek různých subjektů a objektů. Vývoj začal analýzou podobných webových aplikací a zjištěním, co jaká aplikace svým uživatelům poskytuje. Následoval návrh vlastní webové aplikace, která kombinuje vlastní prvky a prvky vícero konkurenčních aplikací do jednoho balíčku. Mezi ty hlavní prvky patří tvorba veřejně dostupných vstupních stránek s kontaktními údaji, odkazy na sociální sítě, firemními hierarchiemi nebo geolokacemi. Dále pak umožňuje fulltextově vyhledávat a ukládat cizí stránky do oblíbených. Nedílnou součástí návrhu bylo i vymyšlení, jak se bude pracovat s uživatelskými účty, a jak by mohl být projekt později financován. Bylo zvoleno, že uživatelské účty budou sloužit pouze pro definování vlastníků karet a ukládání cizích karet do oblíbených. Z možných typů financování (prodej produktů, zobrazování reklam, prémiové plány) byly vybrány právě prémiové plány. Ty oproti reklamám neomezují uživatelské prostředí a není možné je vypnout doplňkem prohlížeče. Oproti prodeji produktů zase nepředstavují tak vysokou zátěž na vývoj.

Následovala analýza a porovnání webových technologií a nástrojů pro implementaci. Výběr obsahoval programovací jazyky a frameworky pro vývoj uživatelského rozhraní a serverové aplikace, nástroje pro návrh rozhraní, a podpůrné systémy, jako jsou databáze, úložiště souborů, systémy pro odesílání emailů a nástroje pro provoz v produkčním prostředí.

Po analýze dostupných technologií, byla v první řadě zvolena architektura aplikace, od které se odvíjel výběr technologií a vývoj. Pro zajištění vysoké interaktivnosti byla zvolena architektura, která rozděluje aplikace na front-endovou JS aplikaci a back-endový API server. Pro komunikaci v rámci této architektury byl zvolen vzor REST pro jeho rozšířenost a vyspělost.

Po zvolení architektury aplikace bylo možné vybrat konkrétní technologie. Pro implementaci samotného vzhledu stránek byl využit framework Sass pro zjednodušení vývoje stylů stránek. Mezi technologiemi pro implementaci interaktivnosti uživatelského rozhraní se objevily frameworky Vue, React, Angular a Swelte. Z těchto byl vybrán framework Vue pro jeho jednoduchost a kvalitní dokumentaci pro začátečníky. U jazyků a frameworků pro vývoj serverové části obsahující veškerou business logiku aplikace, se rozhodovalo mezi Javou, C#, PHP a Javascriptem. Kvůli velmi podobným výhodám napříč programovacími jazyky, byl vybrán jazyk Java společně s frameworkem Spring, spíše podle osobních preferencích a předchozích znalostí.

Další rozhodování se zabývalo výběrem vhodné primární databáze pro uchování uživatelských dat a nástrojem pro poskytnutí fulltextového vyhledávání. Výběr primární databáze zahrnoval populární kandidáty PostgreSQL, MySQL, Oracle DB, MariaDB a MongoDB. Na základě požadavků aplikace se výběr zúžil pouze na SQL databáze, z nichž byla vybrána PostgreSQL, opět podle preferencí, protože žádná z databází neobsahuje významné výhody oproti ostatním. Kromě primární databáze bylo nutné zvolit ještě separátní systém pro fulltextového vyhledávání. V tomto případě byl vybrán Elasticsearch pro jeho rozšířenost, univerzálnost a možnost lokálního provozu.

Souborové úložiště ani systém pro odesílání emailů nebyly implementovány ručně, místo toho byly zvoleny externí služby. V případě souborového úložiště se jedná o Cloudinary (poskytující CDN a tvořič variant obrázků), a v případě emailů byla zvolena služba SendinBlue.

Nakonec, pro provoz aplikace byl zvolen systém Kubernetes a poskytovatel DigitalOcean. Kuberenetes z důvodu volnosti konfigurace více aplikací bez nutnosti spravovat celý server. DigitalOcean pak kvůli nízkým cenám a celkové pověsti poskytovatele.

Následoval návrh uživatelského rozhraní, podle kterého byl navržen i datový model a zvolen návrhový vzor DDD pro strukturování kódu aplikace. Na základě modelu byla nejdříve implementována business logika, jenž byla vzápětí obalena REST API. Business logika zahrnovala mimo jiné implementaci přístupu k databázi pomocí knihovny MyBatis, validaci entit, napojení na souborový systém a propojení aplikačních událostí s emailových systémem. Dále pak zahrnovala i implementaci vyhledávání karet a míst, konfiguraci sociálních sítí nebo implementaci prémiových plánů. Implementace API zase zahrnovala jeho návrh, zabezpečení pomocí knihovny Spring Security a transformaci doménových entity na přenosné datové objekty. S připraveným funkčním API se zaměření přesunulo na tvorbu front-endové části podle návrhu uživatelského rozhraní. Ta řešila problémy v podobě implementace komunikace s API, centrálního zdroje informací, validace vstupních polí nebo upozornění. Nakonec byly pro obě aplikace vytvořeny Docker image a byl nakonfigurován systém Kubernetes.

Všechny definované požadavky na aplikaci byly ve výsledku splněny a aplikace byla otestována na produkčním prostředí. Aplikace je dále otevřena k potencionálním budoucím rozšířením.

Literatura

[1] Design Tools Database, *UX Tools*. [online]. [cit. 2022-03-28]. Dostupné z: https://uxtools.co/tools/design

[2] Web Design and Applications, *W3C*. [online]. [cit. 2021-08-12]. Dostupné z: https://www.w3.org/standards/webdesign/

[3] MDN contributors. HTML: HyperText Markup Language, *mdn*. [online]. [cit. 2021-08-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTML

[4] MDN contributors. CSS: Cascading Style Sheets, *mdn*. [online]. [cit. 2021-08-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/CSS

[5] MDN contributors. What is JavaScript, *mdn*. [online]. [cit. 2021-08-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript

[6] MDN contributors. WebAssembly, *mdn*. [online]. [cit. 2021-08-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly

[7] Sass Basics, *Sass*. [online]. [cit. 2021-08-11]. Dostupné z: https://sass-lang.com/guide

[8] Less Overview, *less*. [online]. [cit. 2021-08-11]. Dostupné z: https://lesscss.org/

[9] Coyier, C. Sass vs. Less, *CSS-TRICKS*, 2012. [online]. [cit. 2021-08-11]. Dostupné z: https://css-tricks.com/sass-vs-less/

[10] Wikipedia contributors. Bootstrap (front-end framework), *Wikipedia*. [online]. [cit. 2021-08-11]. Dostupné z: https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

[11] TailwindCSS, *tailwindcss*. [online]. [cit. 2021-08-11]. Dostupné z: https://tailwindcss.com/

[12] Mourya, N. Tailwind CSS vs. Bootstrap: Which Is a Better Framework?, *MUO*, 2021. [online]. [cit. 2021-08-11]. Dostupné z: https://www.makeuseof.com/tailwind-css-vs-bootstrap-which-is-a-better-framework/

[13] React, *React*. [online]. [cit. 2021-08-11]. Dostupné z: https://reactjs.org/

[14] Vue Guide, *Vue.js*. [online]. [cit. 2021-08-11]. Dostupné z: https://vuejs.org/v2/guide/

[15] Mistry, J. Vue vs React: Comparison of Best JavaScript Frameworks, *Monocubed*, 2021. [online]. [cit. 2021-08-11]. Dostupné z: https://www.monocubed.com/vue-vs-react/

[16] Svelte Basics, *Svelte*. [online]. [cit. 2021-08-11]. Dostupné z: https://svelte.dev/tutorial/basics

[17] Li, A. Vue vs Svelte: Comparing Framework Internals, *Vue Mastery*, 2021. [online]. [cit. 2021-08-11]. Dostupné z: https://www.vuemastery.com/blog/vue-vs-svelte-comparing-framework-internals/

[18] Technostacks. React vs Angular: Which Is A Better JavaScript Framework?, *Technostacks*, 2020. [online]. [cit. 2021-08-12]. Dostupné z: https://technostacks.com/blog/react-vs-angular

[19] MDN contributors. HTTP, *mdn*, 2021. [online]. [cit. 2021-08-13]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP

[20] Wikipedia contributors. Java (programming language), *Wikipedia*. [online]. [cit. 2021-08-13]. Dostupné z: https://en.wikipedia.org/wiki/Java_(programming_language)

[21] Spring Framework Documentation Core, *Spring*. [online]. [cit. 2021-08-14]. Dostupné z: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html

[22] Spring Framework Documentation Web, *Spring*. [online]. [cit. 2021-08-14]. Dostupné z: https://docs.spring.io/spring-framework/docs/current/reference/html/web.html

[23] Wikipedia contributors. Jakarta EE, *Wikipedia*. [online]. [cit. 15. 08. 2021]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Jakarta_EE&oldid=1034427545

[24] Brown, J. Java EE vs. Spring: Which is more popular?, *Xperti*, 2021. [online]. [cit. 15. 08. 2021]. Dostupné z: https://xperti.io/blogs/java-ee-vs-spring/

[25] MDN contributors. Express/Node introduction, *mdn*. [online]. [cit. 2021-08-14]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

[26] MDN contributors. Concurrency model and the event loop, *mdn*. [online]. [cit. 15. 08. 2021]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

[27] StJs2020, *State Of JavaScript*. [online]. [cit. 2021-08-14]. Dostupné z: https://2020.stateofjs.com/en-US/technologies/

[28] Create a Next.js App, *NEXT.js*. [online]. [cit. 2021-08-14]. Dostupné z: https://nextjs.org/learn/basics/create-nextjs-app

[29] NuxtJS, *Nuxt*. [online]. [cit. 2021-08-14]. Dostupné z: https://github.com/nuxt/nuxt.js

[30] Wikipedia contributors. PHP, *Wikipedia*. [online]. [cit. 2021-08-14]. Dostupné z: https://en.wikipedia.org/wiki/PHP

[31] Bhatia, S. Best PHP Frameworks for Web Development, *hackr.io*. [online]. [cit. 2021-08-14]. Dostupné z: https://hackr.io/blog/best-php-frameworks

[32] Proč používat Nette, *Nette*. [online]. [cit. 2021-08-14]. Dostupné z: https://doc.nette.org/cs/3.1/why-use-nette

[33] Laravel vs Symfony in 2021, *Asper Brothers*, 2019. [online]. [cit. 2021-08-14]. Dostupné z: https://asperbrothers.com/blog/laravel-vs-symfony/

[34] ASP.NET, *Microsoft*. [online]. [cit. 2021-08-15]. Dostupné z: https://dotnet.microsoft.com/apps/aspnet

[35] Blazor, *Microsoft*. [online]. [cit. 2021-08-15]. Dostupné z: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor

[36] Introduction to SQL, *W3Schools*. [online]. [cit. 2022-03-25]. Dostupné z: https://www.w3schools.com/sql/sql_intro.asp

[37] Schaefer, L. What is NoSQL, *MongoDB*. [online]. [cit. 2022-03-25]. Dostupné z: https://www.mongodb.com/nosql-explained

[38] Understanding the Different Types of NoSQL Databases, *MongoDB*. [online]. [cit. 2022-03-25]. Dostupné z: https://www.mongodb.com/scale/types-of-nosql-databases

[39] MariaDB. What is ACID Compliance? What It Means and Why You Should Care, *MariaDB*, 2018. [online]. [cit. 2022-03-25]. Dostupné z: https://mariadb.com/resources/blog/acid-compliance-what-it-means-and-why-you-should-care/

[40] When to Use a NoSQL Database, *MongoDB*. [online]. [cit. 2022-03-25]. Dostupné z: https://www.mongodb.com/nosql-explained/when-to-use-nosql

[41] Mei, S. Why You Should Never Use MongoDB, *Sarah Mei*, 2013. [online]. [cit. 2022-03-25]. Dostupné z: http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/

[42] Data in: documents and indices, *elastic*. [online]. [cit. 2022-03-27]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html

[43] Algolia, *Algolia*. [online]. [cit. 2022-03-27]. Dostupné z: https://www.algolia.com/

[44] Matayoshi, M. ElasticSearch vs Algolia, *Medium*, 2017. [online]. [cit. 2022-03-27]. Dostupné z: https://medium.com/@matayoshi.mariano/elasticsearch-vs-algolia-96364f5567a3

[45] typesence, *Typesence*. [online]. [cit. 2022-03-27]. Dostupné z: https://github.com/typesense/typesense#features

[46] Taylor, M. 7 Best Transactional Email Services: Sendgrid vs. Mandrill & More, *VentureHarbour*. [online]. [cit. 2022-04-19]. Dostupné z: https://www.ventureharbour.com/transactional-email-service-best-mandrill-vs-sendgrid-vs-mailjet/

[47] Ahlgren, M. Mailchimp vs Sendinblue (Which One is Better.. and Cheaper?), 2022. [online]. [cit. 2022-04-19]. Dostupné z: https://www.websiterating.com/email-marketing/mailchimp-vs-sendinblue/

[48] What is a Container?, *docker*. [online]. [cit. 2022-03-27]. Dostupné z: https://www.docker.com/resources/what-container/

[49] Pingdom, *solarwinds pingdom*. [online]. [cit. 2022-03-28]. Dostupné z: https://www.pingdom.com

[50] Gupta, L. What is REST, *REST API Tutorial*. [online]. [cit. 2021-08-14]. Dostupné z: https://restfulapi.net/

[51] GraphQL, *GraphQL*. [online]. [cit. 04. 04. 2022]. Dostupné z: https://graphql.org/

[52] What Is SEO / Search Engine Optimization?, *Thirt Door Media*. [online]. [cit. 2021-08-15]. Dostupné z: https://searchengineland.com/guide/what-is-seo

[53] Lawrence, J. The UX Infinity Gems 6 Ways to Create Great UX, *YouTube*, 2017. [online]. [cit. 2022-04-09]. Dostupné z: https://www.youtube.com/watch?v=aZZCZpc0AcY

[54] Augusta, L. Vizuální rovnováha v návrhu UI, *Designui*. [online]. [cit. 2022-04-09]. Dostupné z: https://www.designui.cz/lekce/vizualni-rovnovaha-v-navrhu-ui

[55] MDN Contributors. Object-oriented programming, *mdn*. [online]. [cit. 2022-04-12]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming

[56] Avran, A.; Marinescu, F. *Domain-Driven Design Quickly*. 2006.

[57] Mihalcea, V.; Ebersole, S.; Boriero, A.; et al. Hibernate ORM 6.0.0.Final User Guide, *Hibernate*. [online]. [cit. 2022-04-15]. Dostupné z: https://docs.jboss.org/hibernate/orm/6.0/userguide/html_single/Hibernate_User_Guide.html

[58] Getting started, *MyBatis*. [online]. [cit. 2022-04-15]. Dostupné z: https://mybatis.org/mybatis-3/getting-started.html

[59] Gajos, G. How Hibernate Almost Ruined My Career, *Developers*. [online]. [cit. 2022-04-15]. Dostupné z: https://www.toptal.com/java/how-hibernate-ruined-my-career

[60] Index and search analysis, *elastic*. [online]. [cit. 2022-04-16]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-index-search-time.html

[61] Anatomy of an analyzer, *elastic*. [online]. [cit. 2022-04-16]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/analyzer-anatomy.html

[62] Regular Expression Language - Quick Reference, *Microsoft Docs*, 2022. [online]. [cit. 2022-04-16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference

[63] Hamilton, T. Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE, *Guru99*. [online]. [cit. 2022-04-17]. Dostupné z: https://www.guru99.com/unit-testing-guide.html

[64] Hamilton, T. Integration Testing: What is, Types, Top Down & Bottom Up Example, *Guru99*. [online]. [cit. 2022-04-17]. Dostupné z: https://www.guru99.com/integration-testing.html

[65] KirstenS. Cross Site Request Forgery (CSRF), *OWASP*. [online]. [cit. 2022-04-17]. Dostupné z: `https://owasp.org/www-community/attacks/csrf`

[66] Dockerfile reference, *docker docs*. [online]. [cit. 2022-04-17]. Dostupné z: `https://docs.docker.com/engine/reference/builder/`

[67] Deployments, *kubernetes*. [online]. [cit. 2022-04-17]. Dostupné z: `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/`

[68] StatefulSets, *kubernetes*. [online]. [cit. 2022-04-17]. Dostupné z: `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/`

Seznam zkratek

ACID atomicita, konzistence, izolace, trvalost

API Application programming interface

CDN Content delivery network

CSRF Cross Site Request Forgery

CSS Cascading Style Sheets

DDD Domain-driven design

DI Dependency Injection

DTO Data transfer objekt

GUI grafické uživatelské rozhraní

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IČO Identifikační číslo osoby

JS Javascript

JSON JavaScript Object Notation

Less Leaner Style Sheets

MVC Model-View-Controller

NoSQL Not only SQL

OOP Objektově orientované programování

ORM Object-relational mapping

PHP PHP: Hypertext preprocessor

REST Representational State Transfer

Sass Syntactically Awesome Style Sheets

SEO Search engine optimization

SOAP Simple Object Access Protocol

SQL Structured Query Language

SSL Secure Sockets Layer

UI uživatelské rozhraní

URL Uniform Resource Locator

UX uživatelský zážitek

VPS Virtual private server

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Seznam obrázků

Seznam ukázek kódů

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání bakalářské práce

Autor: **Lukáš Hornych**

Studium: I1900191

Studijní program: B1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název bakalářské práce: **Vstupní stránky pro osoby a instituce s pokročilým vyhledáváním**

Název bakalářské práce AJ: Landing pages for individuals and institutions with advanced search

Cíl, metody, literatura, předpoklady:

Vytvoření portául pro osoby, firmy, organizace atd. pro vzájemné sdílení kontaktních a jiných údajů (rozcestník pro sociální sítě (a jiné aplikace), kontaktní údaje, údaje o struktuře firmy, podobné/příbuzné subjekty apod). Cílem je sjednotit funkce portálů jakými jsou Linktree, LinkedIn, Firmy.cz apod.. a poskytnout tak uživatelům jednodušší vyhledávání a poskytování ověřených informací o sobě na jednom místě

V návaznosti na přehled a zhodnocení konkurenčních řešení bude návrženo vlastní řešení, popsány použité technologie a způsob implementace.

Bude upřesněna

Garantující pracoviště: Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu

Vedoucí práce: prof. RNDr. PhDr. Antonín Slabý, CSc.

Datum zadání závěrečné práce: 15.12.2021