Ukázková aplikace s využitím Clean Architecture

Bc. Patrik Procházka

Diplomová práce
2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Patrik Procházka**
Osobní číslo:	**A21503**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Ukázková aplikace s využitím Clean Architecture**
Téma práce anglicky:	**Sample Application Using Clean Architecture**

Zásady pro vypracování

1. Proveďte rešerši základních architektur pro návrh aplikací.
2. Vypracujte stručný rozbor technologií, které budou použity k návrhu.
3. Proveďte rozbor a analýzu požadavků na zvolené řešení.
4. Zpracujte aplikaci na základě výsledků analýzy.
5. Věnujte pozornost zabezpečení aplikace.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. MARTIN, Robert C., 2018. Clean architecture: a craftsman's guide to software structure and design. 1. Robert C. Martin series. London, England: Prentice Hall. ISBN 01-344-9416-4.
2. FOWLER, Martin, 2003. Patterns of enterprise application architecture. Addison-Wesley signature series. Boston: Addison-Wesley. ISBN 03-211-2742-0.
3. COPLIEN, James O. a BJØRNVIG, Gertrud, 2010. Lean Architecture: for Agile Software Development. 1. John Wiley. ISBN 0470684208.
4. CERVANTES, Humberto a KAZMAN, Rick, 2003. Designing software architectures: a practical approach. Boston: Addison-Wesley. ISBN 978-0-13-439078-9.
5. VAUGHN, Vernon, 2003. Implementing Domain-Driven Design. Boston: Addison-Wesley. ISBN 978-0321834577.

Vedoucí diplomové práce: **doc. Ing. Petr Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše), bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

Patrik Procházka, v.r.

.....

podpis autora

ABSTRAKT

Diplomová práce se zabývá aplikací principů Clean Architecture v procesu vývoje softwaru. Zkoumá, jak může tento architektonický styl zlepšit kvalitu softwaru a efektivitu vývojových procesů. V teoretické části je představena Clean Architecture a jsou vysvětleny klíčové koncepty i vrstvy. Porovnání s dalšími architektonickými přístupy, jako jsou Domain-Driven Design, Event-Driven Architecture a Hexagonal Architecture ilustruje silné a slabé stránky Clean Architecture, což umožňuje lépe pochopit její relativní pozici v softwarovém inženýrství. Praktická část popisuje vývoj konkrétní aplikace s využitím Clean Architecture, včetně analýzy požadavků, návrhu systému a implementace. Pro vývoj aplikace byly vybrány technologie jako Java, Spring Boot a PostgreSQL, které podporují principy Clean Architecture. Práce demonstruje, že aplikace Clean Architecture může výrazně zvýšit udržitelnost softwaru a jeho připravenost na budoucí rozšíření. Tato architektura nabízí silný základ pro vývoj robustních a flexibilních softwarových řešení, schopných adaptace na měnící se požadavky a technologický pokrok.

Klíčová slova: Clean Architecture, softwarová architektura, Java, Spring, Spring Boot, Hibernate, JWT, REST API, vývoj webových aplikací

ABSTRACT

The thesis deals with the application of Clean Architecture principles in the software development process. It explores how this architectural style can improve the quality of software and the efficiency of development processes. The theoretical part introduces Clean Architecture and explains the key concepts and layers. Comparisons with other architectural approaches such as Domain-Driven Design, Event-Driven Architecture and Hexagonal Architecture illustrate the strengths and weaknesses of Clean Architecture, allowing a better understanding of its relative position in software engineering. The practical section describes the development of a specific application using Clean Architecture, including requirements analysis, system design, and implementation. Technologies such as Java, Spring Boot, and PostgreSQL that support Clean Architecture principles were selected for application development. The thesis demonstrates that Clean Architecture applications can significantly increase the software's

sustainability and readiness for future enhancements. This architecture offers a strong foundation for developing robust and flexible software solutions capable of adapting to changing requirements and technological advances.

Keywords: Clean Architecture, software architecture, Java, Spring, Spring Boot, Hibernate, JWT, REST API, web development

Tímto bych chtěl poděkovat vedoucímu práce doc. Ing. Petrovi Šilhavému, Ph.D., za veškerou podporu a konstruktivní zpětnou vazbu, kteoru mi při psaní práce poskytnul. Zároveň velké díky patří i mé úžasné ženě, manažerovi a kolegům, protože bez jejich podpory, by bylo dokončení diplomové práce mnohem náročnější.

OBSAH

ÚVOD

V dnešní době je vývoj softwaru neoddělitelnou součástí technologického pokroku a inovací. Kvalita a udržitelnost softwaru se stávají klíčovými faktory pro úspěch jakékoli aplikace. S rostoucí složitostí softwarových projektů roste i potřeba efektivních architektonických řešení, které podporují snadnou údržbu a rozšiřitelnost systémů. V této diplomové práci se zaměřuji na implementaci aplikace využívající principy Clean Architecture.

Hlavním cílem této práce je prozkoumat efektivitu a aplikovatelnost Clean Architecture ve vývoji moderních softwarových aplikací. Analyzuji, jak tento architektonický styl ovlivňuje celkový design, výkon a udržitelnost aplikace ve srovnání s jinými populárními přístupy, jako jsou Domain-Driven Design (DDD), Event-Driven Architecture (EDA) nebo Hexagonal Architecture.

V první kapitole se věnuji základním principům softwarové architektury, představuji koncept Clean Architecture a srovnávám jej s dalšími architektonickými styly. Tato kapitola pomáhá identifikovat klíčové charakteristiky a výhody, které Clean Architecture nabízí v rámci softwarového inženýrství.

Druhá kapitola popisuje technologický stack, který byl použit pro vývoj serverové a frontendové části aplikace. Zde podrobně rozebírám nástroje a technologie, které byly zvoleny pro realizaci projektu, a zdůvodňuji jejich výběr v kontextu požadavků na projekt.

Třetí kapitola se zabývá analýzou funkčních a nefunkčních požadavků aplikace, identifikací hlavních person a vytvořením user stories. Tato část poskytuje ucelený přehled o tom, jak aplikace reaguje na potřeby uživatelů a jaké klíčové funkce musí být implementovány.

Ve čtvrté kapitole je popsána implementace serverové části aplikace. Zde se podrobněji věnuji tomu, jak principy Clean Architecture ovlivňují design a implementaci serverové části a jaké výhody z toho plynou pro celý vývojový proces.

Pátá kapitola se soustředí na vývoj frontendové části aplikace. Přiblížím zde přístupy i architektonické principy použité při vývoji frontendu.

Tato diplomová práce nabízí ucelený pohled na implementaci aplikace s využitím Clean Architecture i její význam pro moderní softwarový vývoj. Účelem práce je poskytnout porozumění implementaci a výhodám tohoto architektonického stylu, který se může stát klíčem k úspěchu v dynamicky se vyvíjejícím světě softwaru.

I. TEORETICKÁ ČÁST

1 Softwarová architektura

Softwarová architektura je základním kamenem každého systému. Její účel a struktura jsou klíčové pro pochopení, jak software funguje a jaké má vlastnosti. Softwarová architektura je definována jako soubor struktur potřebných k porozumění systému, které zahrnují softwarové prvky, vztahy mezi nimi a jejich vlastnosti.[1]

Zároveň poskytuje plán pro systém, jehož návrh, implementace a testování se řídí stanovenou strukturou. Tato struktura nejen že určuje, jak bude systém vybudován, ale také předurčuje, jak bude možné systém v budoucnu rozvíjet a udržovat. Architektura zahrnuje výběr technologií, návrh komponent, definici interakcí mezi těmito komponentami a strategie pro jejich integraci.[2]

Softwarová architektura má zásadní význam v procesu vývoje softwaru, protože ovlivňuje jak kvalitu, tak rozsah projektu. Dobře navržená architektura umožňuje efektivní vývoj a snazší údržbu systému. Naopak, nedostatečná nebo špatně navržená architektura může vést k technickému dluhu a vysokým nákladům na úpravy a rozšíření těchto problémů. [1, 2]

Architektura hraje klíčovou roli ve vývojovém procesu tím, že definuje, jak budou jednotlivé komponenty systému spolupracovat. Zajišťuje, že systém bude schopen splnit stanovené požadavky na funkčnost i výkon, zatímco zůstává flexibilní pro budoucí rozšíření a údržbu. Je to základ, na kterém stojí celý software, a jakákoli změna v architektuře může mít značné dopady na celkovou funkcionalitu a výkon systému.[1]

1.1 Clean architecture

Tato kapitola bude zaměřena na přístup Clean Architecture, který byl představen Robertem C. Martinem a dále rozšířen dalšími autory[3]. Clean Architecture představuje architektonický styl, který usiluje o vytvoření software s velkým důrazem na oddělení závislostí, testovatelnost a snadnou údržbu aplikace. Tento architektonický přístup je založen na principu nezávislosti byznys pravidel na technologické infrastruktuře, což umožňuje vyšší flexibilitu a robustnost výsledné aplikace.

Clean Architecture se skládá z několika koncentrických vrstev, kde každá vrstva má specifický účel a je oddělena od ostatních vrstev pomocí definovaných rozhraní. To znamená, že změny v jedné vrstvě nemají přímý dopad na vrstvy ostatní.[3]

1.1.1 Klíčové koncepty

Clean Architecture usiluje o maximální oddělení a nezávislost různých částí systému. Tento přístup se zakládá na několika základních konceptech: nezávislost na UI, nezá-

vislost na databázi, nezávislost na jakémkoliv agentu vnějšího světa, testovatelnost, a nezávislost na frameworku.[3]

Nezávislost na uživatelském rozhraní klade důraz na oddělení byznys logiky aplikace od uživatelského rozhraní. Tento princip znamená, že byznys pravidla a případy užití systému nejsou vázány na specifický způsob, jakým jsou data prezentována uživateli. Díky tomu může být logika aplikace snadno přizpůsobena novým požadavkům z pohledu uživatelského rozhraní bez nutnosti zásahů do samotné byznys logiky.

Nezávislost na databázi stanovuje, že doménová logika by neměla být závislá na konkrétních technologiích používaných pro ukládání dat. Využívá se abstrakce, která umožňuje byznys logice komunikovat s databází skrze interface, což usnadňuje eventuální změny v databázových technologiích bez dopadu na byznys logiku.

Nezávislost na vnějším světě specifikuje, že systém by měl být navržen tak, aby byl nezávislý na externích agentech, jako jsou API třetích stran nebo externí knihovny. Toto se dosahuje pomocí tzv. portů a adaptérů, které izolují aplikaci od externích závislostí a umožňují jejich snadnou výměnu.

Testovatelnost klade důraz na snahu o vysokou míru testovatelnosti všech komponent systému. Byznys logiku lze testovat nezávisle na uživatelském rozhraní a databázi, což znamená, že testy mohou být prováděny rychle a efektivně bez nutnosti komplikovaného nastavování nebo simulace vnějších závislostí.

Nezávislost na frameworku zdůrazňuje, že aplikace by neměla být závislá na konkrétním softwarovém frameworku. Architektura by měla být navržena tak, aby bylo možné framework kdykoliv snadno vyměnit, což přináší větší flexibilitu a snížené riziko zastarávání technologií.

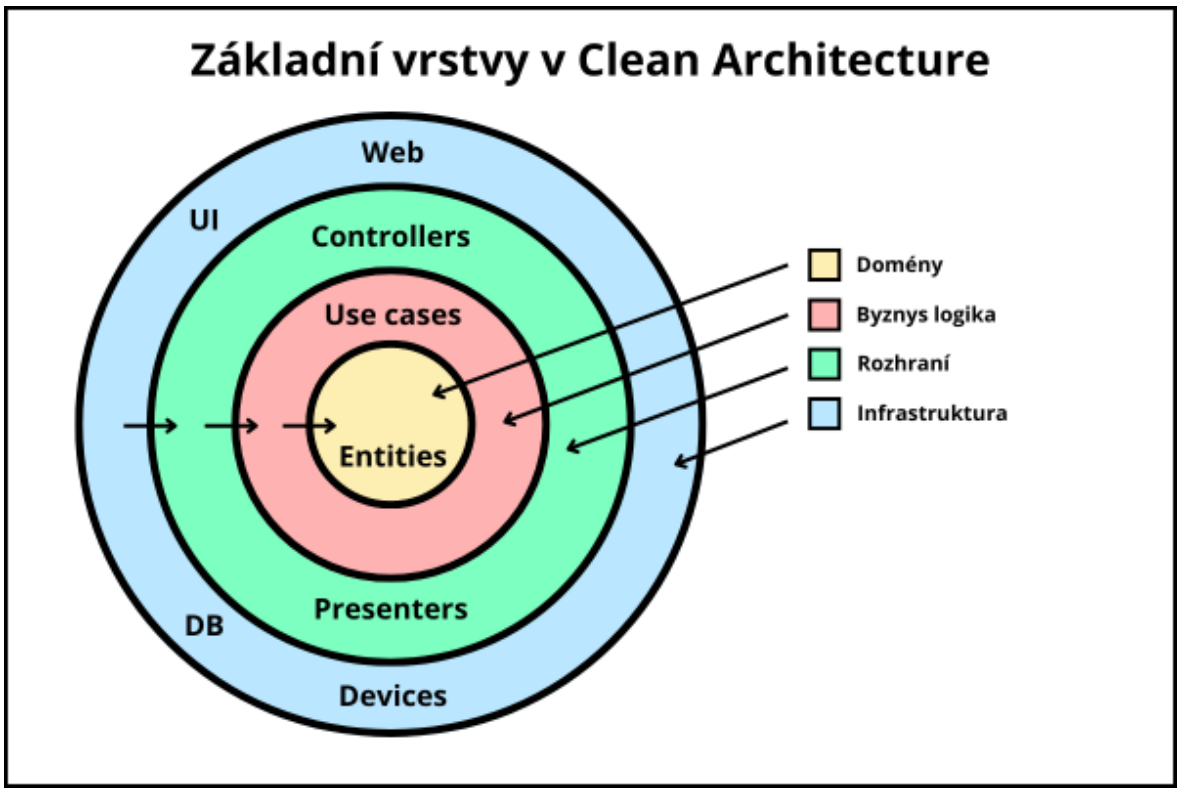
1.1.2 Architektonické vrsty

Clean Architecture je založena na čtyřech základních vrstvách[3], kterými jsou:

- **Doménová vrstva (Entities):** Tato vrstva obsahuje základní byznys pravidla a logiku systému. Jsou zde definovány základní objekty (entity), které představují koncepty z domény aplikace.
- **Aplikační vrstva (Use Cases):** Zahrnuje aplikovaná byznys pravidla. Tato vrstva orchestruje tok dat mezi doménovou vrstvou a vrstvou infrastruktury a/-

nebo prezentace, řídí aplikovanou logiku a zajišťuje, že požadavky uživatelů jsou splněny.

- **Infrastrukturní vrstva:** Obsahuje všechnu logiku a nástroje potřebné ke komunikaci s externím světem, jako jsou databáze, souborové systémy a externí služby. Vrstva zajišťuje, že data potřebná pro aplikaci jsou správně načítána a ukládána.
- **Prezentační vrstva (Controllers):** Řídí interakce s uživateli a deleguje pracovní postupy na aplikaci. Tato vrstva transformuje data z modelu na formu, která je vhodná pro uživatelské rozhraní.



Obr. 1.1 Schéma Clean Architecture. Autor Robert C. Martin[3]

Tyto vrstvy lze rozšiřovat nebo zvyšovat jejich počet, ale je potřeba vždy dodržet základní koncepty architektonického přístupu a také pravidlo o závislostech. To zní: *"Source code dependencies must point only inward, toward higher-level policies."*[3]

Z toho lze vyvodit, že seskupení tříd a funkcí do vrstev nestačí k přehlednému a udržovatelnému kódu. Je zřejmé, že řídící struktura musí procházet alespoň několika (ne-li všemi) vrstvami, aby se něco událo. V projektech využívajících Clean Architecture je potřeba mít výhody a cíle tohoto přístupu na paměti, nelze interakce mezi vrstvami ponechat náhodě. Tedy, žádná nižší vrstva nesmí znát a používat nic z žádné vyšší vrstvy. Například není dovoleno použít třídu, funkci nebo modul z infrastruktury, pokud se

nacházíme v doménové vrstvě. Pravidlo závislosti nejenže zakazuje vývojáři explicitně importovat prvky z vnější vrstvy, ale také nedoporučuje přijímat je jako argumenty funkcí. Pravidlo závislosti je v diagramu architektury znázorněno šipkami. Směr šipek je stejný jako směr závislostí: infrastruktura používá aplikaci, aplikace používá doménu, ale není dovoleno, aby doména používala infrastrukturu nebo aplikaci a podobně.[4]

1.1.3 Výhody a benefity

Ze základních konceptů, rozdělení vrstev a pravidlu o závislostech lze vyvodit několik výhod a benefitů, které jsou zásadní pro vývoj moderních a robustních softwarových aplikací. Lze je pojmenovat jako: oddělení závislostí, testovatelnost, flexibilita a udržitelnost, nezávislost na konkrétních technologiích, zvýšení bezpečnosti.

Klíčovým přínosem Clean Architecture je její schopnost oddělit závislosti mezi různými částmi systému. Tím se snižuje vzájemná provázanost komponent, což vede k vyšší modularitě a snazší údržbě kódu. Díky striktnímu oddělení byznys logiky od uživatelského rozhraní, databází a externích frameworků lze jednotlivé části systému testovat nezávisle. To umožňuje detailnější a efektivnější testovací procesy, které jsou méně náchylné k chybám způsobeným změnami v jiných částech aplikace.

Clean Architecture umožňuje snadné začlenění nových technologií a frameworků bez potřeby zásadních změn v byznys logice nebo datových modelech. Zároveň není vázána na konkrétní technologie uživatelského rozhraní nebo databázové platformy. Tato flexibilita zajišťuje, že aplikace může růst a vyvíjet se s minimálními náklady na odstraňování technického dluhu. Stejně tak zvyšuje možnosti opětovného použití kódu a usnadňuje změny nebo nahrazení těchto externích vrstev bez dopadu na doménovou logiku.

Tím, že byznys logika a data jsou izolovány od uživatelského rozhraní a externích vstupů, Clean Architecture přirozeně vede k lepší ochraně proti bezpečnostním hrozbám, jako je SQL injection nebo cross-site scripting (XSS). Nezávislé vrstvy umožňují efektivnější implementaci bezpečnostních kontrol a auditů.[3, 4]

1.1.4 Limitace a výzvy

Ačkoli Clean Architecture přináší mnoho výhod, existují také některé limitace a výzvy, které je třeba zvážit při jejím zavádění do projektů. Dle Sebastiana Buczyńského a Roberta C. Martina jsou to: složitost architektury, vyšší nároky na design, údržba a závislosti, over-engineering, zkušenosti a znalosti.

Implementace Clean Architecture může vést ke zvýšené složitosti systému, protože vyžaduje definování mnoha vrstev, komponent a interakcí mezi nimi. Pro týmy nezvyklé na tuto úroveň abstrakce může být tento přístup obtížně uchopitelný a může způsobit

zpoždění v raných fázích vývoje. Clean Architecture totiž vyžaduje pečlivé plánování a detailní návrh systému. Tato potřeba může taktéž zvýšit počáteční náklady na vývoj, protože týmy musí strávit více času návrhem a testováním architektury, než začnou skutečně implementovat funkcionality.

Ačkoli princip nezávislosti vrstev umožňuje snazší údržbu jednotlivých komponent, může také vést k náročnější správě závislostí mezi vrstvami. Správné řízení těchto závislostí je klíčové pro zachování integrity systému a vyžaduje dobře definovaná rozhraní a dodržování pravidla o závislostech. Důsledkem může být riziko "over-engineeringu", kdy týmy mohou být příliš zaměřeny na perfektní dodržování architektonických pravidel na úkor pružnosti a rychlosti vývoje. Tento přístup tedy může být kontraproduktivní, zejména v projektech s omezenými zdroji nebo tam, kde je požadována rychlá reakce na měnící se požadavky trhu. Současně úspěšná implementace Clean Architecture vyžaduje zkušenosti a znalosti v oblasti softwarového designu a architektury. Může se stát, že bude nutné poskytnout dodatečné školení nebo najmout specialisty, což může být pro některé organizace nákladné jak finančně, tak časově.

1.2 Porovnání s jinými přístupy

V této kapitole dojde k porovnání Clean Architecture se třemi dalšími prominentními architektonickými přístupy, které jsou běžně používány v softwarovém inženýrství: Domain-Driven Design, Event-Driven Architecture a Hexagonal Architecture (také známá jako Ports and Adapters). Každý z těchto přístupů nabízí unikátní perspektivy a techniky pro řešení běžných výzev v návrhu a implementaci softwarových systémů. Cílem je poskytnout porozumění klíčovým charakteristikám každého přístupu a způsobům, jakými řeší otázky oddělení závislostí, modularity, testovatelnosti a udržitelnosti aplikací. Tato analýza umožní objevit, jak různé architektonické strategie přistupují k integraci technologií a jak se vyrovnávají se složitostí softwarových projektů. Porovnání těchto přístupů nám poskytne ucelenější pohled na to, jak mohou ovlivnit celkovou kvalitu a výkon aplikací ve specifických vývojových kontextech.

Tato kapitola popisuje výhody, nevýhody a limitace velmi zběžně. Pro každý jeden architektonický přístup by bylo možné napsat samostatnou závěrečnou práci. Zároveň je důležité podotknout, že v praxi se přístupy navzájem doplňují a není nic neobvyklého, že se na projektu používá jak Clean Architecture, tak Domain-Driven Design, či například Domain-Driven Design v kombinaci s hexagonální architekturou, nebo microservisy, které využívají Clean Architecture jako strukturu kódu jednotlivých služeb, ale zároveň komunikace mezi službami dodržuje principy Event-Driven Architecture.

1.2.1 Domain-Driven Design

Domain-Driven Design je metodika návrhu softwaru, která se zaměřuje na komplexní potřeby podnikového prostředí a na vytváření softwarových řešení, která jsou hluboce spojená s podnikovým modelem, pro který je aplikace využívající Domain-Driven Design vyvíjena. Domain-Driven Design propaguje modelování založené na skutečném porozumění domény, což je zásadní pro vytváření efektivních a flexibilních softwarových systémů. Tento přístup klade důraz na spolupráci mezi odborníky na doménu a vývojáři[5], aby bylo dosaženo jasného a funkčního modelu, který odráží skutečné podnikové procesy a pravidla.

Výhody a benefity Tento přístup zdůrazňuje významnou výhodu v podobě vylepšené komunikace mezi vývojáři a odborníky na doménu, což je zprostředkováno používáním jednotného jazyka. Jednotný jazyk, který je založen na modelu, umožňuje přirozenou lingvistickou schopnost použít k dolaďování modelu. Tento aspekt je neocenitelný, protože zajišťuje, že veškeré diskuse a rozhodování během vývoje jsou přímo vázané na model a jeho aplikaci v reálném softwaru. Další klíčovou výhodou je schopnost podporovat flexibilitu a udržitelnost softwarových řešení. Doménové modely jsou navrženy tak, aby odrážely složité podnikové procesy a poskytovaly základ pro snadnou adaptaci a rozšíření systému v reakci na měnící se podnikové požadavky. Tato schopnost rychle reagovat na změny a integrace nových funkcí bez rozsáhlých přestaveb systému je zásadní pro udržení konkurenceschopnosti v dynamických podnikových prostředích.[6]

Limitace a výzvy Při vývoji s využitím Domain-Driven Designu se vývojáři setkávají s řadou výzev a limitací, které mohou komplikovat implementaci a zvyšovat náročnost projektů. Jednou z hlavních výzev je složitost správy doménového modelu, zvláště v prostředí s mnoha provázanými a dynamicky se měnícími podnikovými pravidly. Komplexní asociace a interakce mezi objekty mohou ztížit udržení konzistence a integrity modelu, což je nezbytné pro účinné využití. Správa životního cyklu objektů vyžaduje důkladné plánování a návrh, aby bylo možné efektivně řešit změny a aktualizace stavů objektů.[6]

Další výzvou je technická složitost, která může převážit nad jasností a přehledností doménového modelu. Rozsáhlé a technicky náročné architektury mohou vést k tomu, že kód již neodráží doménový model jasně a přehledně. To může ztěžovat vývojářům práci s modelem a snižovat schopnost systému adaptovat se na změny v doméně. Navíc, přílišná rozdrobenost kódu do mnoha modulů může způsobit ztrátu celkového přehledu a komplikovat správu a rozvoj aplikace.

Jedním z nejzávažnějších omezení je riziko nadměrného technického zaměření, které může vést k ignorování podstatných aspektů domény. Toto riziko se projevuje zejména v projektech, kde infrastruktura a frameworky předepisují podmínky struktury řešení příliš rigidně, což může bránit vývojářům ve vytváření efektivních a inovativních řešení, která by lépe odpovídala reálným potřebám a podmínkám domény.[6]

Porovnání s Clean Architecture Oba přístupy, Domain-Driven Design a Clean Architecture, se snaží o vytvoření dobře navrženého a udržitelného softwaru, ale každý se zaměřuje na jiné aspekty systému. Zatímco DDD se soustředí na hluboké porozumění a modelování domény, Clean Architecture klade důraz na oddělení závislostí a organizaci kódu tak, aby byl systém nezávislý na externích faktorech a snadno testovatelný. Clean Architecture poskytuje jasný rámec pro oddělení oblastí zodpovědnosti, což usnadňuje budoucí údržbu a rozvoj systému, zatímco Domain-Driven Design se zaměřuje na vytvoření robustního doménového modelu, který podporuje komplexní podnikové procesy a pravidla. Důležité je podotknout, že oba přístupy mohou být účinně kombinovány, jelikož Domain-Driven Design poskytuje model a strukturu domény, zatímco Clean Architecture organizuje tento model do udržitelné a testovatelné architektury.

1.2.2 Event-Driven Architecture

Event-Driven Architecture je architektonický vzor, který využívá události pro koordinaci a správu změn v programu a jeho stavech. Tento přístup umožňuje aplikacím reagovat na změny ve svém prostředí nebo interní stavy téměř okamžitě. Odděluje komponenty aplikace, což zvyšuje jejich modularitu a snadnost údržby. V Event-Driven Architecture jsou komponenty spojeny pouze prostřednictvím událostí, což znamená, že jedna komponenta vysílá událost, zatímco jiné komponenty ji poslouchají a reagují na ni podle potřeby.

Výhody a benefity Architektura založená na eventech nabízí řadu výhod, které jsou zvláště důležité pro systémy, jež vyžadují vysokou škálovatelnost, flexibilitu a efektivní reakci na změny v reálném čase. Jednou z hlavních předností je schopnost umožnit systémům reagovat na události v reálném čase, což znamená, že systémy mohou být navrženy tak, aby byly více interaktivní a responzivní. Tato vlastnost je neocenitelná v prostředích, kde je čas reakce kritický.[7, 8]

V Event-Driven Architecture jsou služby a komponenty navrženy tak, aby komunikovaly především prostřednictvím událostí, což znamená, že změny v jedné části systému mohou být zpracovány jinými částmi bez přímého ovlivnění nebo potřeby synchro-

nizace. Toto umožňuje lepší škálovatelnost a snadnější integraci nových funkcí nebo služeb bez zásadního zasahování do existujícího kódu.

Další výhodou je výrazné zlepšení v oblasti odolnosti systému proti chybám. Díky asynchronnímu zpracování událostí mohou systémy efektivněji zvládat chybové stavy a pokračovat v provozu i při selhání jedné nebo více komponent.[7] To zvyšuje celkovou spolehlivost systému a minimalizuje dopady potenciálních problémů na uživatele a provoz.

Limitace a výzvy Event-Driven Architecture může přinést značné výhody pro systémovou architekturu, ale také s sebou přináší určité výzvy a omezení. Jednou z hlavních výzev je zvýšená složitost správy a debugování, která vyplývá z asynchronní povahy této architektury. Tato charakteristika může komplikovat proces lokalizace a opravy chyb ve vysoce distribuovaných a decentralizovaných systémech, ve kterých mohou být události zpracovány v různých časech a v různém pořadí.[8]

Dalším problémem v Event-Driven Architecture je testování a zajištění kvality, které vyžaduje specifické strategie, neboť standardní metody mohou být nedostatečné pro identifikaci chyb v asynchronních operacích, což ztěžuje testování koncových bodů bez pevně stanovených interakcí mezi komponentami. Distribuovaná povaha Event-Driven Architecture navíc může vést k problémům s udržením konzistence dat, kde různé části systému mohou vidět různé verze dat a vyžadují speciální řešení pro udržení integrity dat napříč službami.

Za další nevýhodu lze považovat závislost na infrastruktuře, která je kritickým aspektem Event-Driven Architecture, zejména co se týče robustních systémů pro zpracování a doručování zpráv. Jakékoliv selhání v těchto komponentách může mít závažný dopad na celkovou funkčnost systému, protože bezchybné zpracování a doručování zpráv je zásadní pro správnou komunikaci a koordinaci mezi různými částmi systému.[8] Výzvou je také řízení a monitorování zpráv a událostí, zejména v rozsáhlých systémech, v nichž jsou efektivní nástroje pro monitorování a logování klíčové pro rychlou a účinnou reakci na potenciální problémy.

Porovnání s Clean Architecture Oproti Clean Architecture, která klade důraz na oddělení závislostí a strukturu aplikace, se Event-Driven Architecture zaměřuje především na asynchronní zpracování událostí a komunikaci mezi komponentami. Obě architektury mohou být účinně kombinovány. Clean Architecture poskytuje jasný strukturální rámec jednotlivým prvkům systému a Event-Driven Architecture dodává reaktivitu a flexibilitu mezi jednotlivými prvky systému.

1.2.3 Hexagonal Architecture

Hexagonální architektura, známá také jako architektura portů a adaptérů, má za cíl oddělit logiku aplikace od způsobů, jakými aplikace komunikuje s vnějším světem.[9] Tímto oddělením se usnadňuje nezávislost komponent a umožňuje se snadnější testování a údržba softwaru. Aplikace je středem architektury, kolem níž jsou rozmístěny porty reprezentující rozhraní pro vstupy a výstupy, které pak komunikují s adaptéry, jež mohou být specificky navrženy pro různé typy technologií, jako jsou databáze, webové služby nebo uživatelské rozhraní.

Výhody a benefity Hexagonální architektura přináší několik klíčových výhod, které se zaměřují na udržitelnost a flexibilitu aplikací. Tento architektonický vzor efektivně odděluje byznys logiku aplikace od externích technologií, což zvyšuje adaptabilitu aplikace na změny technologického prostředí bez nutnosti zásahů do jejího jádra. Tato modularita zjednodušuje procesy testování a údržby, neboť jednotlivé komponenty aplikace – adaptéry a porty – lze testovat nezávisle na sobě. Vývojáři tak mohou implementovat nebo upravovat adaptéry pro nové technologie, aniž by riskovali stabilitu částí systému.[9]

Další výhodou je, že hexagonální architektura podporuje principy čistého kódu a objektově orientovaného návrhu tím, že jasně definuje rozhraní mezi různými částmi systému. Díky tomu je systém lépe organizovaný a snadnější na pochopení a správu, což je zásadní zejména v dlouhodobých projektech. Tato vlastnost také usnadňuje onboarding nových vývojářů, kteří mohou rychleji pochopit architekturu a logiku aplikace. Tyto výhody vedou k efektivnějšímu a produktivnějšímu vývojovému procesu.

Navíc, architektura umožňuje lepší škálovatelnost a rozšiřitelnost aplikací. Vzhledem k tomu, že komunikace mezi jádrem aplikace a vnějšími službami probíhá prostřednictvím portů a adaptérů, lze snadno přidávat nové funkce nebo integrace bez zásahu do základních byznys logik aplikace. To znamená, že aplikace může růst a vyvíjet se s minimálním dopadem na stávající operace a bez nutnosti rozsáhlých změn v kódu.

Limitace a výzvy Jedním z hlavních omezení Hexagonální architektury je zvýšená složitost v návrhu a implementaci systému. Hexagonální architektura vyžaduje pečlivé plánování a definici portů a adaptérů, což může způsobit, že vývojový proces bude náročnější a delší, než je tomu u jednodušších architektonických vzorů.

Další výzvou je správa závislostí a interakcí mezi porty a adaptéry. Přestože jeden z cílů hexagonální architektury je minimalizovat závislosti mezi jednotlivými částmi systému, špatně navržené rozhraní může vést k nejasnostem v komunikaci a neefektiv-

ním voláním mezi komponentami. To může zkomplikovat testování a údržbu systému, zvláště pokud adaptéry nejsou správně izolovány od byznys logiky.[9]

Porovnání s Clean Architecture Hexagonální architektura je často srovnávána s Clean Architecture, protože obě se snaží dosáhnout oddělení obchodní logiky od technologických implementací. Clean Architecture rovněž definuje striktní pravidla pro oddělení závislostí, ale je koncipována s větším důrazem na rozdělení software do vrstev s jednosměrnými závislostmi, které zpřehledňují integraci nových technologií a změn. Obě architektury poskytují robustní základ pro vytváření modulárního a testovatelného softwaru, ale hexagonální architektura nabízí více flexibility v konfiguraci portů a adaptérů, což může být výhodné v prostředích s rychlým technologickým vývojem. Na druhou stranu, Clean Architecture může být efektivnější v situacích, kde je třeba striktnější struktura a kontrola závislostí mezi jednotlivými vrstvami aplikace.

2 Popis technologií

V této kapitole bude podrobně rozebrán technologický stack, který byl zvolen pro vývoj aplikace. Detailní pohled bude poskytnut na každou technologii, která byla vybrána pro zajištění vysoké výkonnosti, škálovatelnosti, uživatelské přívětivosti aplikace a vzájemné synergie. Na serverové straně bylo rozhodnuto využít Java 21 a Spring Boot 3, což jsou technologie známé pro svou robustnost a efektivitu při tvorbě serverových aplikací. Hibernate byl zvolen pro objektově-relační mapování, umožňující snadnější práci s daty v PostgreSQL databázi, která byla vybrána pro svou spolehlivost a podporu komplexních dotazů.

Na prezentační vrstvu byl výběr učiněn ve prospěch Reactu, Typescriptu v kombinaci s Tailwindem, které společně tvoří základ pro vytváření interaktivních a vizuálně přitažlivých uživatelských rozhraní. React byl vybrán pro svou schopnost vytvářet dynamické uživatelské rozhraní s vysokým výkonem, zatímco Typescript přidává silné typování a lepší nástroje pro ladění, čímž zvyšuje kvalitu a udržitelnost kódu. Tailwind, jako framework pro design, byl zvolen pro svou flexibilitu a efektivitu při stylizaci aplikací.

2.1 Serverová a databázová část

Serverová část aplikace je založena na programovacím jazyku Java (verze 21) a frameworku Spring Boot 3, které byly vybrány pro svou výkonnost a flexibilitu v rámci vývoje serverových aplikací. Hibernate, jakožto nástroj pro objektově-relační mapování, bude rozebrán v kontextu jeho využití pro zjednodušení práce s databázovými operacemi v kombinaci s PostgreSQL.

O technologiích bylo rozhodnuto na základě využití daných technologií v oboru a jejich schopnosti poskytovat robustní, bezpečné a škálovatelné řešení pro serverovou část aplikace. Dále bude probráno, jakým způsobem přispívají tyto technologie k celkové architektuře aplikace a jak byla zajištěna jejich integrace s ostatními částmi systému.

2.1.1 Databáze PostgreSQL

PostgreSQL, začínající svou cestu jako projekt POSTGRES na Univerzitě v Kalifornii v Berkeley v pozdních 80. letech 20. století, představuje výsledek více než třicetiletého vývoje v oblasti databázových systémů. Původně zaměřený na rozšíření možností relačních databází skrze pokročilé objektově-relační funkce, PostgreSQL se vyvinul v jednu z nejvíce pokročilých open-source SQL databází dostupných na trhu. Jeho schopnost podporovat širokou škálu datových modelů – od tradičních relačních až po post-relační struktury, jako jsou pole, JSON nebo XML dokumenty – umožňuje vývojářům flexi-

bilitu v návrhu a škálování jejich aplikací. Tato robustní platforma, známá svou spolehlivostí, vysokým výkonem a širokou podporou pro SQL standardy, se rozrostla díky příspěvkům od mnoha vývojářů a uživatelů z celého světa. Dnes je PostgreSQL ceněn pro svou výjimečnou škálovatelnost, bezpečnostní funkce a nízké celkové náklady na vlastnictví, což z něj dělá oblíbenou volbu pro řadu aplikací, od jednoduchých webů po složité analytické platformy.[10] Z těchto důvodu byl zvolen i pro ukázkovou aplikaci této diplomové práce, a to ve verzi 16.

PostgreSQL 16 dosahuje, v závislosti na hardwaru, možnosti více než 1 000 000 čtení za sekundu a více než 50 000 zápisových transakcí za sekundu. S pokročilým hardwarem je možno dosáhnout ještě lepšího výkonu. Díky pokročilému optimalizátoru, který zvažuje různé typy spojení využívající statistiky uživatelských dat, a široké škále typů indexů podporujících všechny datové typy, umožňuje PostgreSQL efektivní zpracování transakčních systémů i komplexních vyhledávacích a analytických úloh. Podpora pro MVCC (Multi-Version Concurrency Control) navíc zajišťuje, že operace čtení a zápisu si navzájem neblokují cestu, což umožňuje smíšené pracovní zatížení bez potřeby přenášení dat z produkčních systémů do analytických úložišť pro vykonání několika ad hoc dotazů.[10]

Bezpečnostní prvky a mechanismy představují klíčový aspekt architektury PostgreSQL, zajišťující ochranu dat a systémových zdrojů před neoprávněným přístupem a útoky. PostgreSQL implementuje rozsáhlý systém autentizace, podporující různé metody od tradičních hesel po sofistikovanější metody jako certifikáty a integrovaná autentizace Kerberos. Systém autorizace pak umožňuje detailní kontrolu nad tím, co mohou uživatelé a aplikace dělat, prostřednictvím oprávnění přiřazených k objektům databáze, což zahrnuje tabulky, pohledy a funkce. Kromě toho poskytuje PostgreSQL možnosti šifrování dat na úrovni sloupce a šifrování dat na úrovni disku, což zajišťuje, že data uložená v databázi jsou chráněna jak při přenosu, tak i v klidu. Síťové zabezpečení s podporou pro šifrování SSL spojení mezi klientem a serverem zabraňuje odposlechu a zajišťuje integritu dat při přenosu. Tyto bezpečnostní prvky, spolu s pravidelnými aktualizacemi a záplatami, činí PostgreSQL řešením pro aplikace vyžadující vysokou úroveň bezpečnosti.

2.1.2 Programovací jazyk Java

Java je jedním z nejpopulárnějších a nejrozšířenějších programovacích jazyků. Základy má v polovině 90. let 20. století, kdy byl vyvinut týmem inženýrů v Sun Microsystems. Od svého uvedení na trh v roce 1995 zaznamenala Java mimořádný růst, díky své nezávislosti na platformě, objektově orientovanému přístupu a bezpečnostním prvkům. Jazyk byl navržen s heslem "Write Once, Run Anywhere", což znamená, že kompilovaný Java kód lze spustit na jakékoliv platformě, která je vybavena Java virtuálním strojem

(JVM). Tato vlastnost umožňuje vývojářům vytvářet aplikace, které jsou přenositelné mezi různými operačními systémy bez nutnosti modifikace kódu.

Java se v průběhu let vyvinula do rozsáhlého ekosystému zahrnujícího standardní edici (Java SE) pro desktopové a serverové aplikace, podnikovou edici (Java EE) pro velké distribuované podnikové systémy, a mikroedici (Java ME) pro vývoj aplikací pro mobilní zařízení a embedded systémy. Vývojářům nabízí rozsáhlou standardní knihovnu tříd (Java API), která pokrývá širokou škálu funkčnosti, od základních operací s daty přes síťovou komunikaci až po grafické uživatelské rozhraní. Bezpečnost je dalším klíčovým prvkem jazyka Java, s robustním modelem bezpečnosti, který izoluje spuštěné aplikace, chrání uživatele před nebezpečným kódem a zajišťuje bezpečný přístup k systémovým zdrojům. Java také podporuje moderní programovací paradigma, včetně lambda výrazů a stream API, což umožňuje vývojářům psát čistší a efektivnější kód.

Díky své spolehlivosti, výkonu a škálovatelnosti je Java preferovanou volbou pro vývoj řady aplikací, od webových a mobilních aplikací přes podnikové systémy až po aplikace pro velké datové analýzy. Java se neustále vyvíjí, aby držela krok s požadavky moderního softwarového vývoje, což zahrnuje pravidelné aktualizace jazyka a JVM, zlepšení v oblasti výkonu i škálovatelnosti a dostupných rozšíření.

Při srovnání Javy s jinými populárními programovacími jazyky, jako jsou Python, Node.js a Go, lze najít rozdíly a podobnosti z hlediska výkonu, škálovatelnosti, komunity a dostupnosti knihoven. Java, se svou dlouhou historií a širokou podporou pro různé typy aplikací, je obecně považována za velmi výkonný a škálovatelný jazyk, zejména pro velké a komplexní podnikové aplikace. Díky JVM a optimalizacím kompilátoru může Java nabídnout vysoký výkon pro náročné aplikace. Python je naopak oblíbený pro svou jednoduchost a rychlost vývoje, avšak může zaostávat v přímém srovnání výkonu s Javou, zejména v CPU-intenzivních aplikacích. Node.js, založený na JavaScriptu, vyniká ve výkonu pro input-output aplikace díky svému neblokujícímu I/O modelu, což jej činí vhodným pro vývoj rychlých a škálovatelných webových aplikací. Go, navržený pro jednoduchost a vysoký výkon, nabízí vynikající škálovatelnost a výkon pro souběžně zpracovávané aplikace. Go je kompilovaný jazyk, a to jej činí rychlejším v některých scénářích než interpretované jazyky.

Komunita a dostupnost knihoven jsou dalšími klíčovými faktory při výběru programovacího jazyka. Java se může pochlubit obrovskou a aktivní komunitou vývojářů a bohatým ekosystémem knihoven a frameworků pro téměř jakékoli použití, což usnadňuje vývoj aplikací. Python má rovněž silnou a rozmanitou komunitu s bohatou sadou knihoven, zejména v oblastech, jako jsou data science, strojové učení a webový vývoj. Node.js, díky své popularitě ve vývoji webových aplikací, má rozsáhlý repozitář balíčků npm, který umožňuje rychlý vývoj moderních webových aplikací. Go, i když je novější

než ostatní zmíněné jazyky, rychle rozvíjí svou komunitu a nabídku knihoven, zejména pro vývoj cloudových a souběžně zpracovávaných aplikací. Každý z těchto jazyků má tedy své jedinečné výhody a nevýhody, a výběr nejlepšího jazyka závisí na specifických potřebách projektu, preferencích vývojářů a požadovaném výkonu a škálovatelnosti aplikace.

2.1.3 Spring Boot Framework

Spring Boot, jako rozšíření Spring Frameworku, představuje klíčovou technologii v oblasti vývoje moderních aplikací v Javě. Jeho zásadním přínosem je značné zjednodušení procesu konfigurace a nasazování aplikací. Tento framework se vyznačuje především automatickou konfigurací a šablonami, známými jako *starters*, což umožňuje vývojářům rychlejší a efektivnější práci. Spring Boot eliminuje množství explicitního boilerplate kódu, který je typicky vyžadován při inicializaci Spring aplikací, a nabízí osvědčené vzory pro běžné vývojářské úkoly. Význam Spring Bootu ve vývoji softwaru lze přičítat jeho schopnosti podporovat rychlost a agilitu vývoje, čímž významně přispívá ke snížení času potřebného pro uvedení produktu na trh nebo jeho údržbu. Jeho architektura je navržena s ohledem na *opinionated* přístup k vývoji, který preferuje konvence před konfigurací. Tento přístup umožňuje vývojářům soustředit se na doménově specifickou logiku aplikace, aniž by byli zatíženi komplexním nastavením a konfigurací. Dalším klíčovým aspektem Spring Bootu je jeho integrace s různými operačními systémy a cloudovými platformami, což umožňuje vývoj cloud-native aplikací. Tento framework podporuje širokou škálu scénářů nasazení od tradičních monolitických aplikací po mikroslužby, a tím podporuje současné trendy v softwarovém inženýrství, jako jsou kontejnerizace a orchestrace. Spring Boot rovněž nabízí rozsáhlou dokumentaci a komunitní podporu, což výrazně usnadňuje adopci a implementaci tohoto frameworku ve vývojových projektech. Díky své modularitě a rozšiřitelnosti se jedná o flexibilní nástroj, který je možné efektivně využít ve škále aplikací, od jednoduchých webových služeb až po složité distribuované systémy.[11]

Spring Boot se etabluje jako klíčová platforma pro integraci s různými databázovými systémy, čímž demonstruje svou adaptabilitu a univerzálnost v kontextu různorodých vývojových paradigmat. Jeho architektura efektivně navazuje připojení k širokému spektru databází, od tradičních relačních databázových systémů, jako jsou MySQL nebo PostgreSQL, po moderní NoSQL databáze jako například MongoDB, Cassandra nebo Redis. Tento rozsah podpory je zajištěn prostřednictvím Spring Data, což je abstrakce, která nabízí konzistentní programovací model nezávislý na specifickém typu databáze, umožňující vývojářům efektivně a transparentně manipulovat s daty napříč různými databázovými technologiemi.

Navíc Spring Boot zjednodušuje konfiguraci databáze automatickým nastavením datových zdrojů a integrací s nástroji pro migraci databází, jako jsou Flyway nebo Liquibase, což umožňuje bezproblémovou správu a evoluci databázových schémat. Podpora pro JPA (Java Persistence API) a *repository pattern* usnadňuje manipulaci s daty a implementaci složitých databázových operací s minimální potřebou boilerplate kódu. Kromě toho Spring Boot přináší podporu pro reaktivní práci s daty skrze projekty Spring Data Reactive Repositories, což umožňuje vývoj vysoce výkonných a škálovatelných aplikací, schopných efektivně zpracovávat velké objemy dat v reálném čase. Tato reaktivní podpora je zásadní pro vývoj cloud-native aplikací a mikroslužeb, kde jsou prioritou asynchronní zpracování a minimalizace latence, což dále potvrzuje význam Spring Bootu jako přední platformy pro vývoj moderních, datově intenzivních aplikací.

Integrace bezpečnostního frameworku Spring Security do ekosystému Spring Boot představuje klíčový prvek v architektuře zabezpečení aplikací vyvíjených v rámci tohoto frameworku. Spring Security poskytuje rozsáhlou podporu pro autentizaci a autorizaci, která je zásadní pro ochranu aplikací a služeb před neoprávněným přístupem. Díky těsné integraci se Spring Bootem je možné tuto podporu efektivně implementovat s minimálním úsilím ze strany vývojáře. Spring Boot automaticky konfiguruje Spring Security s výchozími nastaveními, která poskytují základní úroveň zabezpečení.[12] Tato výchozí konfigurace zahrnuje například zabezpečení HTTP endpointů, základní HTTP autentizaci a generování CSRF tokenů, což přispívá k zajištění aplikace proti běžným bezpečnostním hrozbám.

Pro specifické bezpečnostní požadavky umožňuje Spring Boot vývojářům snadné přizpůsobení konfigurace pomocí externích konfiguračních souborů nebo programově, prostřednictvím Java konfigurace. Vývojáři mohou definovat vlastní pravidla pro autentizaci a autorizaci či nastavit pokročilé metody autentizace, jako je OAuth2, OpenID Connect nebo JWT.[12] Tímto způsobem Spring Boot společně se Spring Security umožňuje vývojářům vytvářet vysoce bezpečné aplikace, které přesně odpovídají požadavkům jejich specifického aplikačního prostředí.

Kromě toho Spring Boot a Spring Security nabízejí rozšířenou podporu pro bezpečnostní testování, což vývojářům umožňuje systematicky ověřovat bezpečnostní aspekty aplikací během vývojového cyklu. Pomocí nástrojů, jako je Spring Security Test, mohou vývojáři psát a spouštět testy, které simulují různé bezpečnostní scénáře, včetně testů autentizace, autorizace a přístupových kontrol.[12] Tato integrovaná testovací podpora umožňuje efektivní identifikaci a řešení potenciálních bezpečnostních hrozeb před nasazením aplikace, což výrazně přispívá k celkové bezpečnosti aplikací vyvinutých v rámci Spring Boot ekosystému.

Mimo jiné je Spring Boot i robustní platforma pro testování aplikací, která různými typy testů (od unit testů po integrační testy) zásadně přispívá k zajištění vysoké kvality a bezpečnosti vývoje. Integrace s hlavními testovacími frameworky, jako jsou JUnit pro jednotkové testování, Mockito pro mockování závislostí a AssertJ pro vyjadřování asertací v expresivním stylu, umožňuje vývojářům efektivně provádět izolované testy jednotlivých komponent a funkcionalit aplikace. Specifické anotace poskytované Spring Boot, jako *@WebMvcTest* a *@DataJpaTest*, pak usnadňují cílené testování aspektů aplikací. Použití *@SpringBootTest* a možnost definování vlastních testovacích konfigurací prostřednictvím *@TestConfiguration* přináší flexibilitu v přizpůsobení testů konkrétním potřebám projektů.[13] Tato metodologie nejenže minimalizuje riziko regresních chyb, ale rovněž podporuje kontinuální vývojový cyklus tím, že umožňuje rychlou identifikaci a nápravu problémů, čímž se zvyšuje celková spolehlivost a uživatelská spokojenost s výsledným softwarem.

2.1.4 Hibernate

Hibernate je ORM nástroj pro jazyk Java, který umožňuje transparentní mapování mezi objektovým modelem aplikace a relační databází. Tento nástroj je v současnosti považován za standard v oblasti objektově relačního mapování v Java ekosystému. Jeho role a mechanismy jsou fundamentální pro správný návrh a implementaci databázové vrstvy aplikace. Hlavní síla Hibernate spočívá ve zjednodušení vývoje perzistentní vrstvy aplikace tím, že převádí břemeno práce s databází z vývojáře na framework. Tím umožňuje vývojářům soustředit se více na logiku aplikace než na jednotlivé operace s databází. To znamená, že místo psaní SQL dotazů mohou vývojáři pracovat přímo s Java objekty, které Hibernate mapuje na databázové tabulky. Tento přístup nejenže zjednodušuje vývoj a zvyšuje čitelnost kódu, ale také redukuje pravděpodobnost chyb spojených s manuálním mapováním dat. Přesto Hibernate nabízí možnost využití HQL, což je objektově orientovaný dotazovací jazyk, který je součástí frameworku. Umožňuje vytvářet komplexní dotazy na databázi, aniž by bylo nutné psát nativní SQL. Vývojáři mohou formulovat dotazy přímo v kontextu Java objektů, což zjednodušuje práci s daty a zvyšuje efektivitu vývoje. Hibernate při komunikaci s databází spravuje transakce a zajišťuje konzistenci dat mezi aplikací a databází prostřednictvím robustního transakčního managementu. To zahrnuje podporu pro deklarativní transakce, které mohou být konfigurovány pomocí XML nebo anotací v Java kódu. Hibernate se postará o všechny potřebné operace, jako je otevření a zavření transakcí, rollback v případě chyby, čímž se výrazně snižuje riziko datových konfliktů a zvyšuje odolnost aplikace.

2.2 Prezentační část

Prezentační vrstva aplikace je klíčovou částí, která slouží pro interakci a komunikaci s uživateli. V současné době existuje široká škála technologií a nástrojů určených pro vývoj frontendových aplikací, z nichž každý přináší specifické výhody i omezení. Pro účely aplikace vyvinuté v rámci této diplomové práce byl výběr technologií proveden s ohledem na moderní trendy vývoje, výkonnost, flexibilitu a kompatibilitu s ostatními částmi systému. Vybrané technologie slouží v současnosti jako technologický standard pro frontendový vývoj.

2.2.1 Programovací jazyk TypeScript

TypeScript je open-source programovací jazyk vyvinutý společností Microsoft. Rychle se stal klíčovým v ekosystému moderního webového vývoje. Jako nadstavba JavaScriptu přináší TypeScript prvky, které umožňují vývojářům psát čistší a bezpečnější kód, specificky přidáním statické typové kontroly, rozhraní, enumerací a generik. TypeScript rozšiřuje JavaScript a zlepšuje vývojářskou zkušenost tím, že umožňuje odhalení chyb již během kompilace kódu, nikoli až při jeho spuštění.[14]

TypeScript byl navržen tak, aby byl snadno přijatelný pro javascriptové vývojáře, což umožňuje hladký přechod mezi jazyky. Jeho syntaxe a funkce jsou navrženy s ohledem na maximální kompatibilitu a efektivitu. Díky tomu je TypeScript ideální volbou pro rozsáhlé webové aplikace, v nichž složitost kódu může vést k chybám, které by v čistém JavaScriptu mohly být obtížně identifikovatelné, například definování datových typů pro data, které webová aplikace obdrží ze serveru.[15]

Vývojářská komunita přijala TypeScript rychle díky jeho schopnosti zvýšit produktivitu a kvalitu kódu. Možnost integrovat TypeScript s existujícími javascriptovými knihovnami a frameworky, jako jsou React, Angular a Vue.js, dále rozšiřuje jeho použitelnost a utvrzuje jeho pozici v moderním webovém ekosystému.

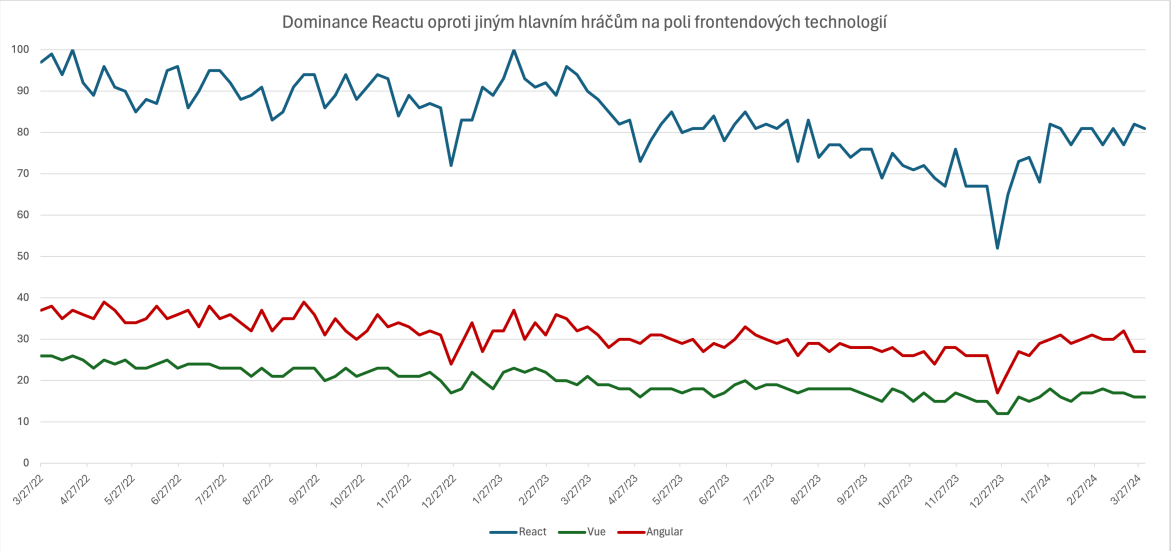
2.2.2 React

React, javascriptová knihovna vyvinutá a udržovaná společností Meta (dříve známá jako Facebook), představuje inovativní nástroj, který zásadním způsobem změnil paradigma vývoje pokročilých webových aplikací. Původním impulsem pro jeho vznik byla snaha o překonání limitací tehdejších přístupů k vývoji webových aplikací, které byly závislé na použití čistého JavaScriptu nebo knihoven, jako je jQuery. Na rozdíl od svých konkurentů v době svého vzniku – Angular, Ember, Knockout či Backbone – React nebyl koncipován jako plnohodnotný framework. Místo toho vznikl jako specializovaná knihovna, která se zaměřuje na reaktivní a efektivní interakci s objektovým modelem dokumentu (DOM) webových stránek.

React se vyznačuje deklarativním paradigmatem, které umožňuje vývojářům snadněji předvídat, jak bude aplikace fungovat, a efektivněji spravovat vizuální komponenty. Klíčovou vlastností Reactu je jeho schopnost izolovat stavy komponent, což umožňuje aktualizovat a renderovat pouze ty části uživatelského rozhraní, které se skutečně změnily, což vede k výraznému zvýšení výkonnosti aplikace.[16]

Kromě toho je důležité zmínit, že React vytvořil základ pro rozsáhlý ekosystém doplňujících knihoven a nástrojů, které dále rozšiřují jeho funkčnost. Tento ekosystém zahrnuje řešení pro správu stavů aplikací, jako je Redux nebo MobX, které umožňují efektivní a předvídatelnou správu stavů ve velkých aplikacích. Dále zahrnuje knihovny a nástroje pro zabezpečení, routování, práci s daty a další aspekty, které jsou kritické pro moderní webové aplikace.

Ve světle těchto skutečností lze React považovat za klíčového hráče ve vývoji frontendových technologií, jehož přístup a filosofie výrazně ovlivnily současné trendy a praxe ve vývoji webových aplikací. Jeho schopnost poskytovat robustní, flexibilní a výkonné řešení pro vytváření uživatelských rozhraní z něj činí preferovanou volbu pro mnoho vývojářů a společností po celém světě. (Obr. 2.1)



Obr. 2.1 Porovnání trendů dle Google Trends[17]

2.2.3 Tailwind

Tailwind, framework pro tvorbu uživatelských rozhraní za pomoci kaskádových stylů, se v posledních letech etabloval jako přední nástroj pro rychlý a efektivní vývoj webových aplikací. Tailwind se odlišuje od tradičních CSS frameworků, jako jsou Bootstrap nebo Foundation, svým přístupem "utility-first", který nabízí vývojářům sadu nízko úrovňových utility tříd. Tyto třídy umožňují rychlé a intuitivní stylování elementů

bez nutnosti psát velké množství vlastního CSS kódu nebo opakovaně používat složité předdefinované komponenty.

Základem frameworku je princip, který podporuje myšlenku, že design by měl být sestaven z malých, znovupoužitelných a kombinovatelných kusů. Tento přístup usnadňuje vytváření konzistentních a esteticky příjemných uživatelských rozhraní tím, že poskytuje vývojářům přímý přístup k široké škále stylů, které lze snadno aplikovat a kombinovat přímo v HTML.

Jedním z klíčových benefitů používání Tailwindu je značné zrychlení vývojového procesu. Je možné okamžitě vidět výsledky práce bez potřeby přepínání mezi soubory nebo dlouhého ladění CSS. Navíc je navržen s ohledem na přizpůsobivost a rozšiřitelnost. Jeho konfigurační soubor umožňuje vývojářům definovat vlastní palety barev, velikosti fontů, okraje a mnoho dalších nastavení, čímž se zvyšuje flexibilita a zároveň vzniká možnost vytvářet specifické designy.

II. PRAKTICKÁ ČÁST

3 Popis aplikace

Tato kapitola se zaměřuje na důležitý aspekt vývoje softwaru, kterým je důkladné porozumění a definování funkčních a nefunkčních požadavků. V raných fázích návrhu je zásadní identifikovat a specifikovat, co software musí dělat (funkční požadavky) a jak by měl tyto funkce provádět (nefunkční požadavky) s ohledem na výkonnost, bezpečnost a uživatelskou přívětivost. Tato kapitola poskytuje vhled do procesu analýzy požadavků, což zahrnuje zohlednění různých typů uživatelů a způsobů, jakými budou uživatelé aplikaci používat, což je ilustrováno na personách. Každá z těchto person má specifické potřeby, které ovlivňují jak funkční, tak nefunkční aspekty aplikace.

3.1 Požadavky na aplikaci

V rané fázi návrhu a vývoje aplikace je klíčové správně pochopit a definovat funkční a nefunkční požadavky aplikace. Funkční požadavky specifikují konkrétní chování nebo funkce systému, například možnost přidávání, přiřazování a rozřazování úkolů uživateli. Na druhou stranu, nefunkční požadavky určují, jak systém tyto funkce provádí. Zahrnují například výkonnost, bezpečnost nebo uživatelskou přívětivost. Správné pochopení a definice těchto požadavků je zásadní pro návrh aplikace, která splňuje očekávání uživatelů a zároveň odpovídá technickým a provozním standardům.[18]

Při analýze požadavků je důležité zohlednit různé typy uživatelů a jak budou aplikaci používat, což musí být zohledněno při tvorbě požadavků. Využití metod, jako jsou uživatelské příběhy a scénáře, může pomoci při identifikaci a definici těchto požadavků, a zajišťuje to, že aplikace bude užitečná pro všechny cílové skupiny uživatelů.[19]

3.1.1 Funkční požadavky

Funkční požadavek je definován jako popis chování, které systém bude vykazovat za specifických podmínek. Tento typ požadavku je klíčový pro definování konkrétních funkcí a operací, které má software vykonávat.[19] Například, v kontextu aplikace pro správu úkolů, funkční požadavek může specifikovat, že systém musí umožnit uživatelům přidávat úkoly, přiřazovat je jiným uživatelům nebo nastavovat úkolům různé tagy. Každý funkční požadavek by měl být jasně definován tak, aby bylo možné jeho splnění ověřit a zároveň aby přesně popisoval jednu konkrétní funkci nebo operaci systému bez zbytečných nejasností nebo obecností. Tímto způsobem funkční požadavky tvoří základ pro návrh a vývoj softwaru, poskytují vývojářům přesné specifikace toho, co musí software dělat, a umožňují testování, zda software tyto specifikace splňuje.

Definované funkční požadavky pro aplikaci jsou tyto:

1. **Možnost vytvoření účtu:** Uživatel si bude moci vytvořit účet.

2. **Možnost přihlášení:** Uživatel se bude moci přihlásit k existujícímu účtu s použitím uživatelského jména a hesla.
3. **Vytvoření úkolu**: Uživatel bude moci přidat nový úkol s popisem a případně dalšími detaily.
4. **Přiřazení deadline:** Uživatel bude moci nastavit nebo změnit deadline pro každý úkol.
5. **Přiřazení osoby zodpovědné za úkol:** Uživatel bude moci přiřadit úkol sobě nebo jinému uživateli.
6. **Smazání úkolu:** Uživatel bude moci odstranit úkol.
7. **Editace úkolu:** Uživatel bude moci upravit detaily existujícího úkolu.
8. **Přiřazení tagů k úkolu:** Uživatel bude moci přiřadit jeden nebo více tagů k úkolu.
9. **Řazení úkolů dle kritérií:** Uživatel bude moci řadit úkoly dle různých kritérií, včetně deadline, priority či přiřazené kategorie.
10. **Přiřazení priority k úkolu:** Uživatel bude moci každému úkolu přiřadit prioritu.

3.1.2 Nefunkční požadavky

Nefunkční požadavky popisují kvality produktu, které určují, jak dobře produkt vykonává své funkce. Na rozdíl od funkčních požadavků, které specifikují konkrétní akce nebo chování produktu, nefunkční požadavky definují vlastnosti, jako je atraktivnost, použitelnost, rychlost, spolehlivost nebo bezpečnost produktu. Tyto požadavky mohou zahrnovat specifikace časové odezvy, limity přesnosti výpočtů, zvláštní vzhled produktu, jeho použití v určitých okolnostech nebo dodržování zákonů relevantních pro daný obchodní sektor.

Nefunkční požadavky jsou klíčové pro určení, jak kvalitně produkt plní své funkce, a nejedná se přímo o funkční aktivity produktu, jako jsou výpočty nebo manipulace s daty. Jsou zahrnuty proto, že klient očekává, že tyto aktivity budou prováděny určitým způsobem a s určitým stupněm kvality.[18] Tyto požadavky tedy hrají zásadní roli ve vývoji produktu, který nejenže splňuje základní funkční očekávání, ale je také atraktivní, snadno použitelný, rychlý, spolehlivý a bezpečný pro své uživatele. Nefunkční požadavky by měly být popsány s ohledem na specifické potřeby produktu a jeho uživatele. Každý případ užití by měl být zvážen z hlediska jeho nefunkčních

potřeb. Při popisování těchto požadavků je důležité zahrnout jak popis požadavku, tak jeho odůvodnění a kritéria pro splnění.

1. **Vzhled a přístupnost:**
   - **Popis:** Aplikace by měla mít intuitivní a uživatelsky přívětivé rozhraní, které je snadno pochopitelné pro nové uživatele a je plně responzivní na různé velikosti obrazovek a zařízení.
   - **Odůvodnění:** Umožnit rychlou adaptaci uživatelů na aplikaci a minimalizovat zahlcení uživatele, zatímco se zajišťuje plynulý uživatelský zážitek na všech zařízeních, od mobilních telefonů po stolní počítače.
   - **Kritéria pro splnění:** Noví uživatelé by měli být schopni vytvořit a spravovat úkoly bez vnější pomoci do 10 minut od prvního použití aplikace. Aplikace by měla automaticky přizpůsobit svůj layout a navigaci tak, aby byly optimálně přístupné a snadno použitelné bez ohledu na velikost a orientaci obrazovky zařízení.
2. **Výkon:**
   - **Popis:** Aplikace by měla načítat seznam úkolů a reagovat na uživatelské akce do 2 sekund.
   - **Odůvodnění:** Zajištění plynulého a rychlého používání aplikace, což zvyšuje uživatelskou spokojenost.
   - **Kritéria pro splnění:** 90 % všech akcí prováděných v aplikaci by mělo být dokončeno do 2 sekund, měřeno na standardním internetovém připojení.
3. **Bezpečnost:**
   - **Popis:** Aplikace musí ověřit, zda uživatel může danou operaci provést například při ukládání nebo editování, aby nedocházelo ke ztrátě dat. Aplikace musí vyžadovat silné heslo při registraci.
   - **Odůvodnění:** Chránit osobní informace a data uživatelů před neoprávněným přístupem.
   - **Kritéria pro splnění:** Využití hashovací funkce BCrypt pro uložená uživatelská hesla. Hesla musí mít minimálně 8 znaků, obsahovat čísla, velká a malá písmena a speciální znaky.

3.1.3 Shrnutí požadavků

V této části diplomové práce jsem definoval a popsal klíčové funkční a nefunkční požadavky pro navrhovanou aplikaci. Tyto požadavky jsou základem pro návrh a vývoj aplikace, zajistí, že aplikace bude splňovat očekávání a potřeby jejích uživatelů, a zároveň se bude držet nejvyšších standardů kvality a uživatelského zážitku.

Funkční požadavky jsem specifikoval jako soubor akcí nebo operací, které aplikace musí být schopna vykonat. To zahrnuje možnost uživatelů přihlásit se a zaregistrovat, vytvářet, přiřazovat a spravovat úkoly, definovat deadliny a přiřadit úkoly specifickým osobám. Také jsem zdůraznil potřebu editace a mazání úkolů, i přidání priorit a tagů k úkolům pro lepší organizaci a filtrování.

Nefunkční požadavky jsem identifikoval jako nezbytné pro zajištění, že aplikace bude fungovat efektivně, bude bezpečná, responzivní a bude mít intuitivní uživatelské rozhraní. Zahrnul jsem požadavky na vzhled a přístupnost aplikace, že aplikace bude responzivní na různé velikosti a typy zařízení. Tímto způsobem jsem zajistil, že aplikace bude přívětivá a přístupná pro široké spektrum uživatelů, kteří používají různá zařízení.

Celkově, pečlivé definování a dokumentace těchto požadavků je klíčová pro úspěšný vývoj aplikace. Poskytuje jasnou představu o tom, co aplikace musí dělat a jak se musí chovat, což umožňuje efektivně plánovat a implementovat potřebné funkce a zajistit, že konečný produkt bude vyhovovat očekáváním uživatelů. Tato kapitola tak představuje základní kámen pro další fáze vývojového procesu, směřující k realizaci robustní, uživatelsky přívětivé a technicky odolné aplikace.

3.2 Persony

Persony jsou nástroj používaný v uživatelském výzkumu a designu, který reprezentuje archetypální uživatele produktu nebo služby. Vytvářejí se na základě shromážděných dat a výzkumu a mají za úkol pomoci vývojářům a designérům lépe rozumět potřebám, chování a motivacím skutečných uživatelů.[20] Akademický přístup k personám zdůrazňuje jejich význam pro vytváření empatie u týmu, což umožňuje informovanější a uživatelsky orientované rozhodování. Persony obvykle zahrnují podrobné informace, jako jsou demografické údaje, profesní pozadí, uživatelské cíle, preference, frustrace a typické chování. Používání person v designovém procesu vede k vývoji produktů, které jsou více přizpůsobeny potřebám uživatelů, a funkcí, které jsou pro uživatele skutečně užitečné a přínosné. Rozsáhlé nasazení person tak podporuje efektivní komunikaci v rámci týmu a zajišťuje, že uživatelské zkušenosti jsou klíčovým prvkem všech fází vývoje produktu.[20]

Tvorba person je systematický proces, který začíná shromažďováním a analýzou dat o skutečných uživatelích. Tento proces obvykle zahrnuje kvalitativní metody, jako jsou rozhovory, pozorování a průzkumy, aby se získal hluboký vhled do uživatelských potřeb, chování a motivací. Na základě těchto dat se identifikují klíčové segmenty uživatelů a pro každý z nich se vytvoří detailní profily. Tyto profily se pak dále rozvíjejí do narativních osobností, které představují specifické uživatele v rámci cílové skupiny. Validace person s reálnými uživateli je klíčová, aby se zajistilo, že tyto profily přesně odrážejí realitu. Celý proces je iterativní, což znamená, že persony se mohou upravovat a zpřesňovat na základě nově získaných informací nebo změn v projektu.

V kontextu diplomové práce jsem zvolil jako základní strategii pro počáteční fázi projektu metodu tvorby person od stolu. Tato metoda, která se opírá o předpoklady a odborné znalosti namísto reálných dat, se ukázala jako klíčová pro identifikaci a pochopení potřeb potenciálních uživatelů aplikace. Přestože jsou persony generované touto technikou považovány za provizorní a očekává se, že budou v průběhu času aktualizovány či nahrazeny na základě shromážděných reálných dat, tento přístup mi umožňuje zajistit, že design a vývoj produktu jsou od samého počátku orientovány na uživatele.

Použití person od stolu mi umožnilo vytvořit základní rámec pro analýzu a návrh uživatelských interakcí, který je nezbytný pro vývoj intuitivně použitelné aplikace, jež reflektuje skutečné potřeby uživatelů. Tato metodologie podporuje hloubkové pochopení uživatelských požadavků a představuje základ pro produkt empaticky navržený a uživatelsky zaměřený.

3.2.1 Jana

Jana je maminka na mateřské dovolené.

- **Demografie:** Jana je ve věku 30–35 let, žije ve městě a je momentálně na mateřské dovolené.
- **Chování:** Jana každodenně řídí chod domácnosti a stará se o své děti. Její denní rutina zahrnuje plánování rodinných aktivit, nákupů a jídelníčku.
- **Cíle:** Jana hledá způsoby, jak efektivně spravovat své denní úkoly, aby mohla trávit více času se svou rodinou.
- **Potřeby:** Potřebuje jednoduchý a intuitivní nástroj pro organizaci svých každodenních úkolů, který jí umožní snadno přidávat, upravovat a sledovat aktivity a nákupy pro rodinu.
- **Frustrace:** Má omezený čas a čelí výzvám s udržením přehledu o všech rodinných plánech a povinnostech.

3.2.2 Josef

Josef je manažer v malé rodinné firmě.

- **Demografie:** Josef je ve věku 40–45 let, pracuje jako manažer v malé rodinné firmě.
- **Chování:** Josef má na starosti řízení týmu a projekty firmy. Denně komunikuje s klienty a koordinuje práci svých zaměstnanců.
- **Cíle:** Jeho hlavním cílem je zvýšení produktivity práce a efektivní správa projektů a úkolů zaměstnanců.
- **Potřeby:** Josef potřebuje robustní nástroj pro správu úkolů, který mu umožní přidělovat úkoly zaměstnancům, sledovat jejich pokrok a zajistit, že projekty jsou dokončeny včas.
- **Frustrace:** Josef se potýká s výzvami při udržování přehledu o mnoha současně probíhajících projektech a při efektivní komunikaci s týmem.

3.2.3 Alžběta

Alžběta je studentkou vysoké školy, obor medicína.

- **Demografie:** Alžběta je ve věku 22–25 let, je studentkou medicíny, která ji časově vytěžuje.
- **Chování:** Svůj čas musí pečlivě rozdělit mezi studium, přednášky, praktická cvičení a přípravu na zkoušky.
- **Cíle:** Alžběta chce zlepšit svou studijní efektivitu a ujistit se, že splní všechny své akademické a osobní závazky včas.
- **Potřeby:** Vyžaduje flexibilní nástroj pro správu času, který jí pomůže organizovat její studijní plány, zkoušky a osobní úkoly.
- **Frustrace:** Alžběta se často cítí přetížená množstvím studijních materiálů a obtížně si plánuje čas tak, aby stihla všechny své povinnosti.

3.3 User stories

User stories jsou krátké, popisné scénáře používané převážně v agilním přístupu při vývoji softwaru. Jedná se o definování požadavků z pohledu uživatele. Tyto příběhy usnadňují pochopení definovaných požadavků, protože lidský mozek vnímá příběhy

mnohem snadněji než strohé technické popisy požadavků. User stories popisují, co uživatel chce, aby software dělal, a proč to chce, čímž se zaměřují na přidanou hodnotu pro uživatele. Dobré user stories jsou specifické, měřitelné, dosažitelné, relevantní a časově omezené.

Vytvoření dobrých user stories začíná identifikací cílů jednotlivých uživatelských rolí, což pomáhá pochopit, co uživatelé od softwaru očekávají a jaké mají potřeby. To umožňuje generovat další, specifičtější stories z těchto širších cílů. Efektivní user stories jsou psány z perspektivy jednotlivého uživatele (persony), aby byly co nejjasnější a nejzřetelnější.[21] Používání aktivního hlasu v user stories zvyšuje jejich čitelnost a srozumitelnost, což usnadňuje porozumění požadavkům a cílům uživatele.

Pro definované persony jsem tedy vytvořil následující user stories. Tyto user stories jsou definované v jednotné šabloně, která obsahuje vše potřebné k definici správné user story.

3.3.1 User story pro Janu

Potřeba: Efektivní správa denních úkolů a rodinných aktivit
User story: Jako maminka na mateřské dovolené chci mít možnost snadno přidávat, upravovat a sledovat každodenní úkoly a plány pro mou rodinu, abych mohla trávit více času se svými dětmi a manželem.
Cíl: Mít více času pro svou rodinu
Akceptační kritéria:

1. Jana může vytvořit nový úkol s názvem, popisem a termínem.
2. Jana může úkoly upravit.
3. Jana může úkoly smazat.
4. Jana může zobrazit seznam úkolů seřazený podle termínu splnění.

3.3.2 User story pro Josefa

Potřeba: Přiřazení a sledování úkolů zaměstnanců
User story: Jako manažer malé rodinné firmy potřebuji nástroj, který mi umožní efektivně přiřazovat úkoly mým zaměstnancům a sledovat jejich pokrok, abych zajistil, že projekty budou dokončeny včas a produktivita práce se zvýší.
Cíl: Efektivně sledovat stavy úkolů a kolik jich je stále ve zpracování
Akceptační kritéria:

1. Josef může v aplikaci snadno vytvořit nový úkol a přiřadit ho konkrétnímu zaměstnanci.

2. Aplikace umožňuje Josefovi nastavit deadline pro každý úkol a přidat poznámky nebo instrukce.

3. Josef může sledovat stav každého úkolu a aktualizovat jej podle potřeby.

3.3.3 User story pro Alžbětu

Potřeba: Správa a připomínání termínů
User story: Jako studentka medicíny potřebuji flexibilní nástroj pro správu termínů, který mi pomůže organizovat termíny zkoušek a úkolů, abych mohla zlepšit svou studijní efektivitu a splnit všechny akademické i osobní závazky včas. Jednotlivé úkoly si můžu vizuálně třídit podle značek.
Cíl: Efektivně spravovat termíny zkoušek a úkolů
Akceptační kritéria:

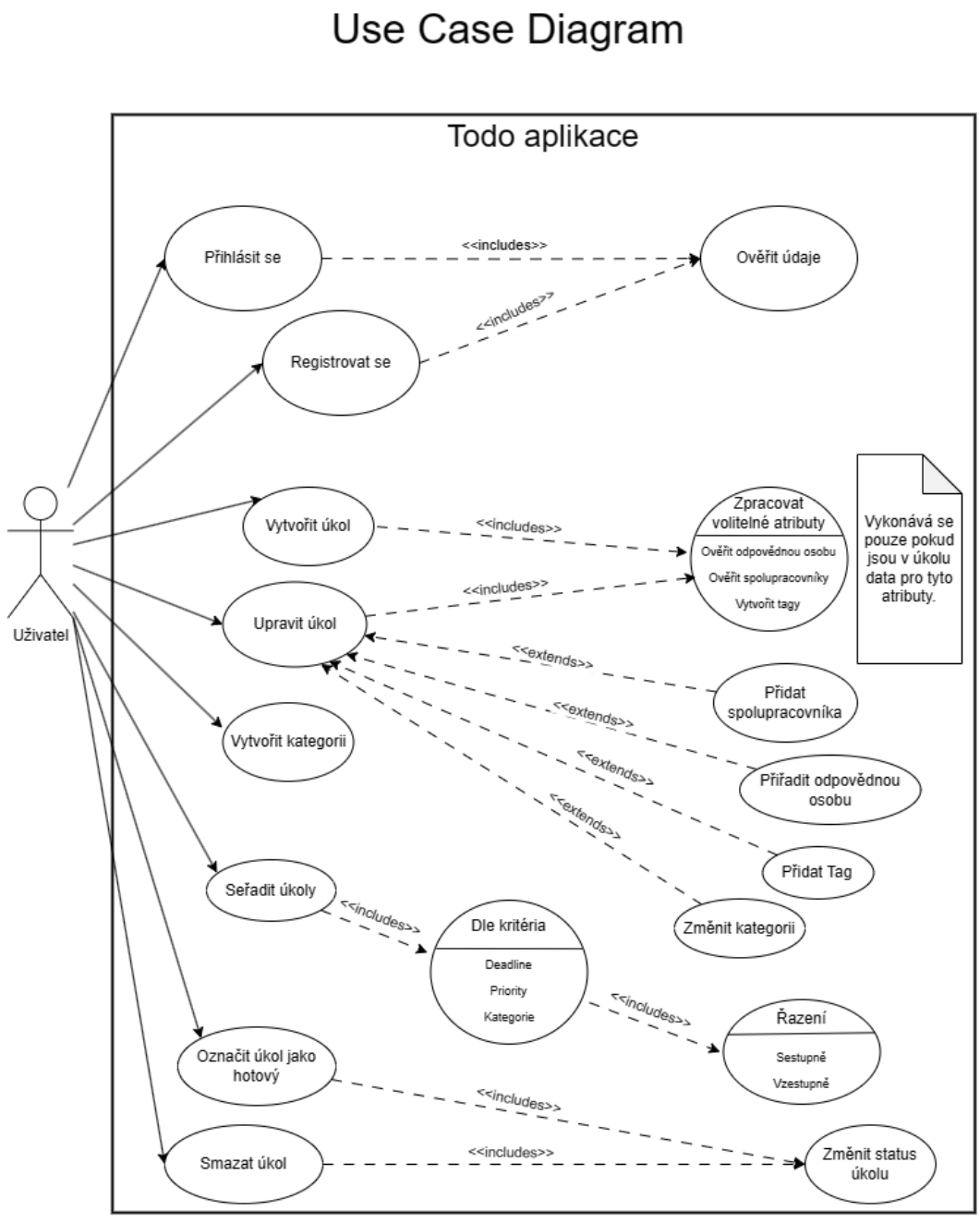
1. Alžběta může v aplikaci vytvořit úkoly a třídit je podle značek.

2. Aplikace umožňuje Alžbětě nastavit termín pro blížící se důležité úkoly.

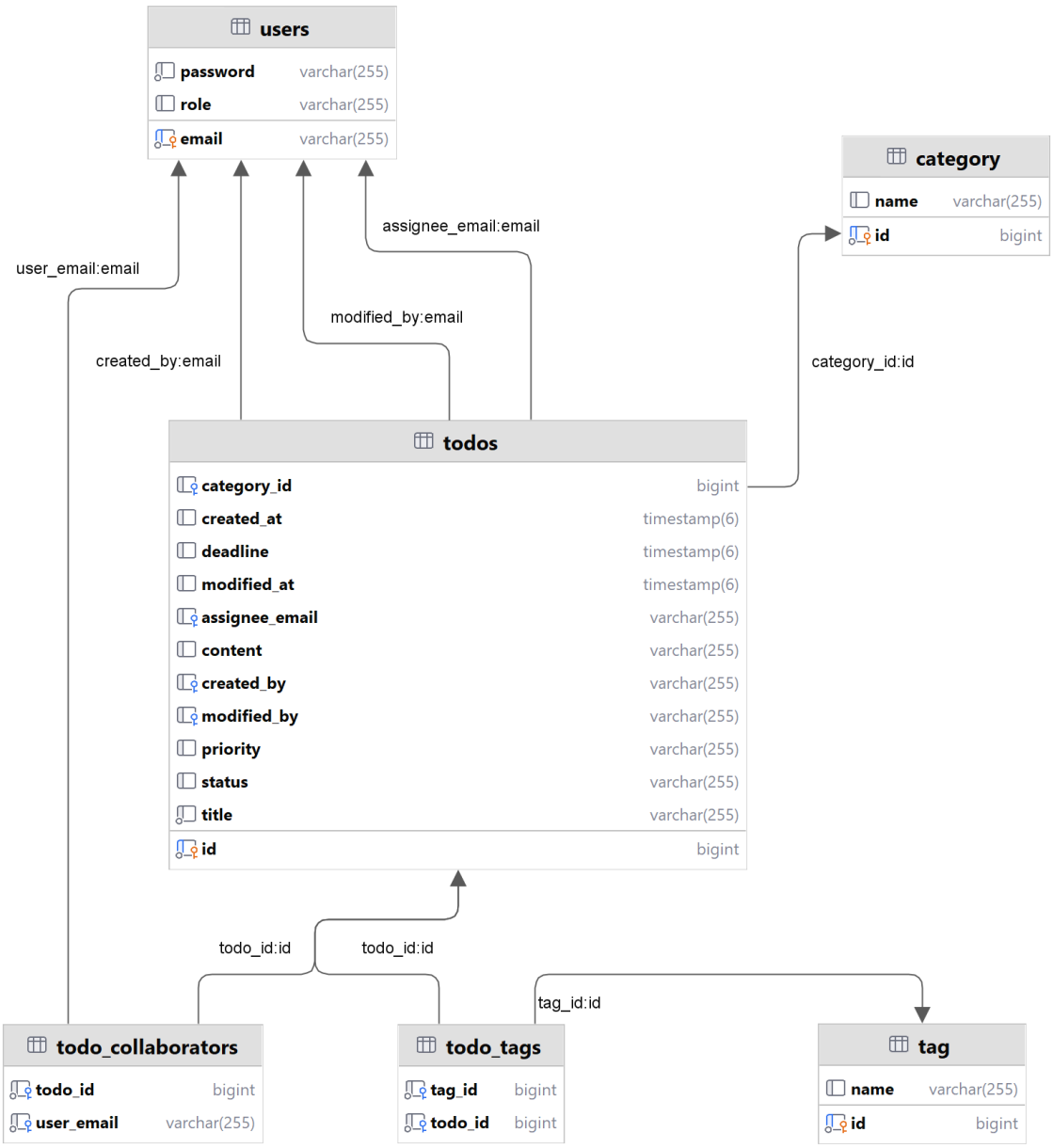
3. Alžběta může přidávat, upravovat a odstraňovat úkoly podle potřeby.

3.4 Diagramy

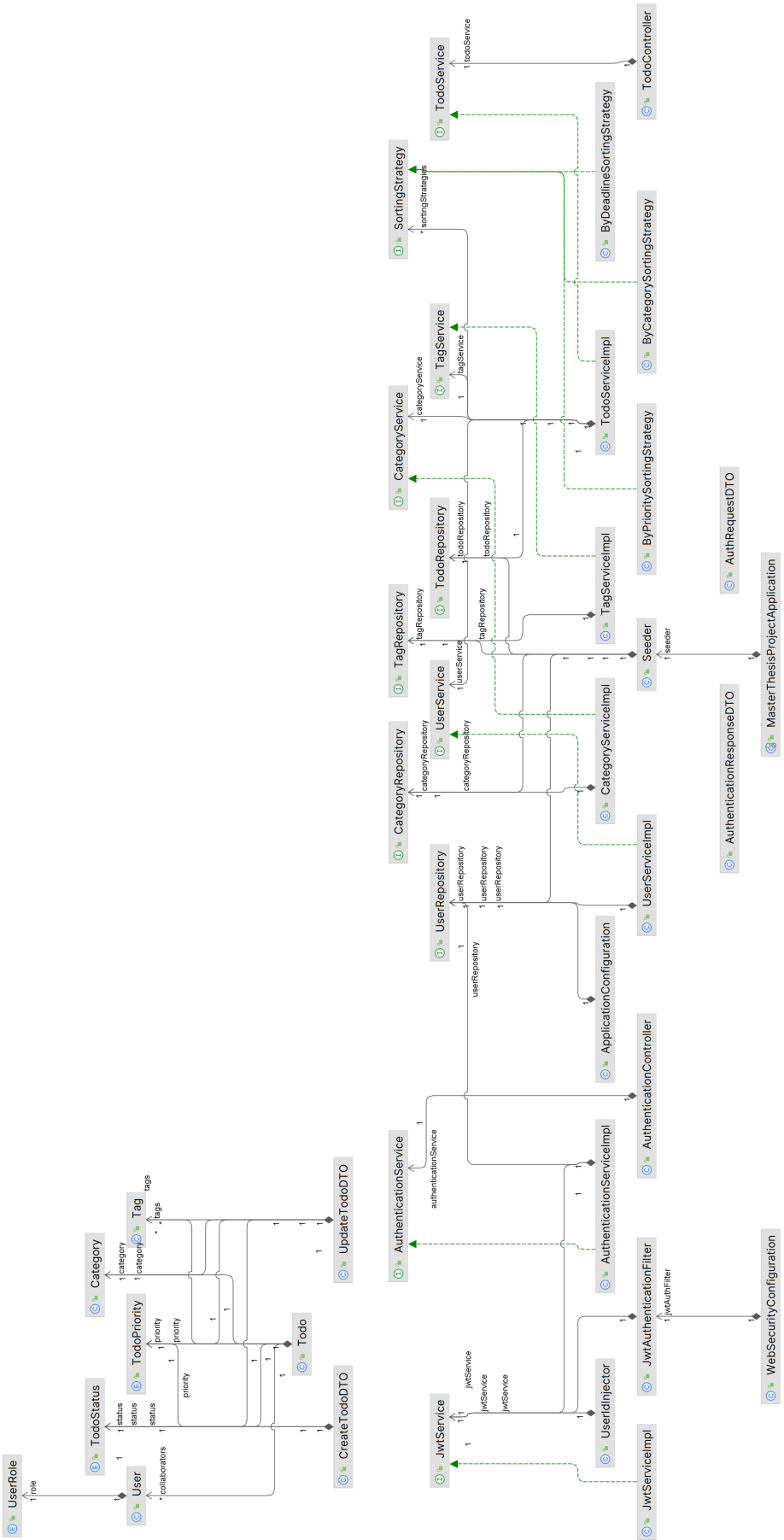
V této kapitole jsou diagramy, které formují základní stavební kameny pro design a strukturu navrhované aplikace. Je zde *Use Case diagram*, který poskytuje vizuální reprezentaci funkcionálních požadavků aplikace a jejich interakce s různými uživatelskými rolemi. Dále se zaměřím na *Entity-relationship diagram*, jenž detailně popisuje strukturu databáze a vztahy mezi entitami, a nakonec *Class diagram*, který ilustruje strukturu a vazby tříd v aplikaci, což umožňuje lepší porozumění architektuře softwarového řešení a jeho modulární konstrukci.



Obr. 3.1 Use Case Diagram, autor: Patrik Procházka



Obr. 3.2 Entity Relationship Diagram, autor: Patrik Procházka



Obr. 3.3 Class Diagram, autor: Patrik Procházka

4 Implementace aplikace

V této části práce se budu věnovat postupné implementaci projektu, jehož cílem je vytvoření aplikace na správu úkolů využívající principy Clean Architecture. Začnu založením projektu pomocí nástroje Spring Initializr, což mi umožní definovat potřebné závislosti a nastavení, které budou odpovídat architektonickým a technologickým požadavkům. Následně se budu zabývat implementací jednotlivých vrstev aplikace v souladu s principy Clean Architecture. Také se zaměřím na vytvoření a vystavení RESTful endpointů, které budou sloužit pro komunikaci s prezentační částí aplikace vyvinutou v Reactu. Důkladný popis každého kroku implementace poskytne hlubší porozumění tomu, jak kvalitně navrhovat a vyvíjet softwarové aplikace s důrazem na čistotu a strukturu kódu.

4.1 Struktura modulů v aplikaci

Projekt je strukturován do čtyř hlavních modulů (vrstev), které se překrývají s teoretickými vrstvami Clean Architecture. Jednotlivé moduly jsou nazvány `adapters`, `domain`, `infrastructure` a `usecases`.

Tato kapitola slouží k představení struktury projektu, přičemž podrobnější popis jednotlivých tříd a rozhraní bude poskytnut v dalších kapitolách.

4.1.1 Modul domain (Entities)

Doménová vrstva obsahuje základní byznys objekty, respektive entity, a logiku úzce spjatou s těmito entitami. Modul zahrnuje třídy jako `Category`, `Tag`, `Todo` a `User`, které reprezentují doménové entity a jejich úzce spjatá byznys logika je definovaná v rozhraních `CategoryService`, `TagService` a `UserService`. Byť se může zdát, že tato rozhraní by měla být v modulu `usecases`, tak logika, kterou poskytují, nemá žádný vliv na byznys logiku aplikace. Implementace těchto rozhraní mají pouze za úkol získávat data z repositářů, které jsou ve vrstvě adaptérů. Díky tomuto přístupu nebudou tato rozhraní nijak ovlivněna změnou ve vrstvě adaptérů. Dále modul zahrnuje výčtové typy jako `TodoPriority`, `TodoStatus` a `UserRole`, které určují předem definované hodnoty používané v celé aplikaci.

4.1.2 Modul usecases (Use Cases)

Modul `usecases` obsahuje definovaná byznys pravidla, která koordinují tok dat mezi doménovou vrstvou a vrstvou adaptérů. Obsahuje rozhraní jako `AuthenticationService`, `JwtService` a `TodoService` a jejich implementace. Tyto služby řídí realizaci specifických uživatelských scénářů, jako je autentizace, správa JWT tokenů, a správa úkolů.

`JwtService` může na první pohled působit jako součást infrastruktury, jelikož se týká autentizačních mechanismů, které jsou závislé na konkrétních technologiích. JWT jsou nicméně klíčové pro řízení přístupu a identifikaci uživatelů v rámci byznys logiky. Přestože implementace JWT může záviset na konkrétní technologii, její použití je řízeno byznys pravidly: jakým způsobem a kdy se tokeny vydávají, obnovují nebo ověřují.

4.1.3 Modul adapters (Interface Adapters)

Modul `adapters` slouží jako most mezi vnějším světem a interními vrstvami aplikace. Obsahuje controllery jako `AuthenticationController` a `TodoController`, které přijímají a zpracovávají HTTP požadavky a převádějí je na volání vhodných služeb, které poskytují byznys logiku. Dále tento modul zahrnuje objekty pro přenos dat (Data Transfer Objects, zkráceně DTO) jako `AuthRequestDTO`, `AuthenticationResponseDTO`, `CreateTodoDTO` a `UpdateTodoDTO`, které slouží k přenosu dat mezi vrstvami.

Zároveň tento modul obsahuje rozhraní jako `CategoryRepository`, `TagRepository`, `TodoRepository` a `UserRepository`, která slouží jako adaptéry pro ukládání a načítání dat z databáze.

4.1.4 Modul infrastructure (Frameworks & Drivers)

Modul `infrastructure` poskytuje nástroje a implementace potřebné pro spuštění a provoz aplikace, a zahrnuje třídy a komponenty pro konfiguraci bezpečnosti jako `ApplicationConfiguration`, `WebSecurityConfiguration` a `JwtAuthenticationFilter`, které nastavují zabezpečení aplikace.

Zároveň se v tomto modulu nachází i třída `Seeder`, která, jak už název napovídá, má za úkol naplnit databázi výchozí sadou dat. Toto v kombinaci s další konfigurací databáze usnadňuje vývoj aplikace, protože po každém znovusestavení aplikace je databáze v předdefinovaném stavu. Tento stav se dá kdykoliv ve třídě `Seeder` změnit ku potřebám vývojáře.

4.2 Clean Architecture a návrhové vzory v praxi

Tato kapitola uvede klíčové návrhové vzory v kontextu vývoje softwaru s využitím frameworku Spring. Zaměříme se především na návrhové vzory Strategy a Repository, které hrají zásadní roli v architektuře aplikací postavených na platformě Spring. Dále popíši, jak tyto vzory podporují principy Clean Architecture. Tyto vzory nejenže napomáhají v účinné modularizaci a správě závislostí, ale také zlepšují rozšiřitelnost a údržbu softwarových řešení. Na příkladech ukážu, jak mohou být tyto vzory využity pro zvýšení flexibility a znovupoužitelnosti komponent v různých aplikačních scénářích.

4.2.1 Návrhový vzor Strategy

Vzor Strategy umožňuje definovat rodinu algoritmů, zapouzdřit každý z nich jako samostatný objekt a pak tyto objekty zaměnitelně používat.[22] Tento vzor je používán při implementaci různých autentizačních strategií v rámci Spring Security.[12] Autentizační mechanismy mohou být snadno zaměněny bez nutnosti měnit ostatní části aplikace, což je v souladu s principem oddělení závislostí v Clean Architecture. Spring také umožňuje výběr různých chování (jako jsou implementace JPA nebo JDBC) prostřednictvím konfigurace[23], což koresponduje s principy návrhového vzoru Strategy.

Příklad Využití návrhového vzoru Strategy je demonstrováno na třídě `SortingStrategy` popsané v kapitole 4.7.2.

4.2.2 Návrhový vzor Repository

Návrhový vzor Repository se používá pro abstrakci a oddělení logiky přístupu k datům od byznys logiky.[24] V aplikaci je každá entita doprovázena svým rozhraním repositáře, které deklaruje metody pro přístup k datům. Tato rozhraní jsou poté implementována s využitím Spring Data JPA, což výrazně redukuje množství kódu potřebného pro implementaci datových operací.[23] Návrhový vzor Repository podporuje principy Clean Architecture tím, že izoluje datovou vrstvu a umožňuje lepší testovatelnost a údržbu aplikace.

Příklad Využití návrhového vzoru Repository a různé implementace rozhraní `JpaRepository` jsou popsané v kapitole 4.9.

4.2.3 Další vzory podporované frameworkem Spring

Spring Framework sám o sobě navádí k používání řady návrhových vzorů, které jsou kompatibilní s principy Clean Architecture. Patří mezi ně:

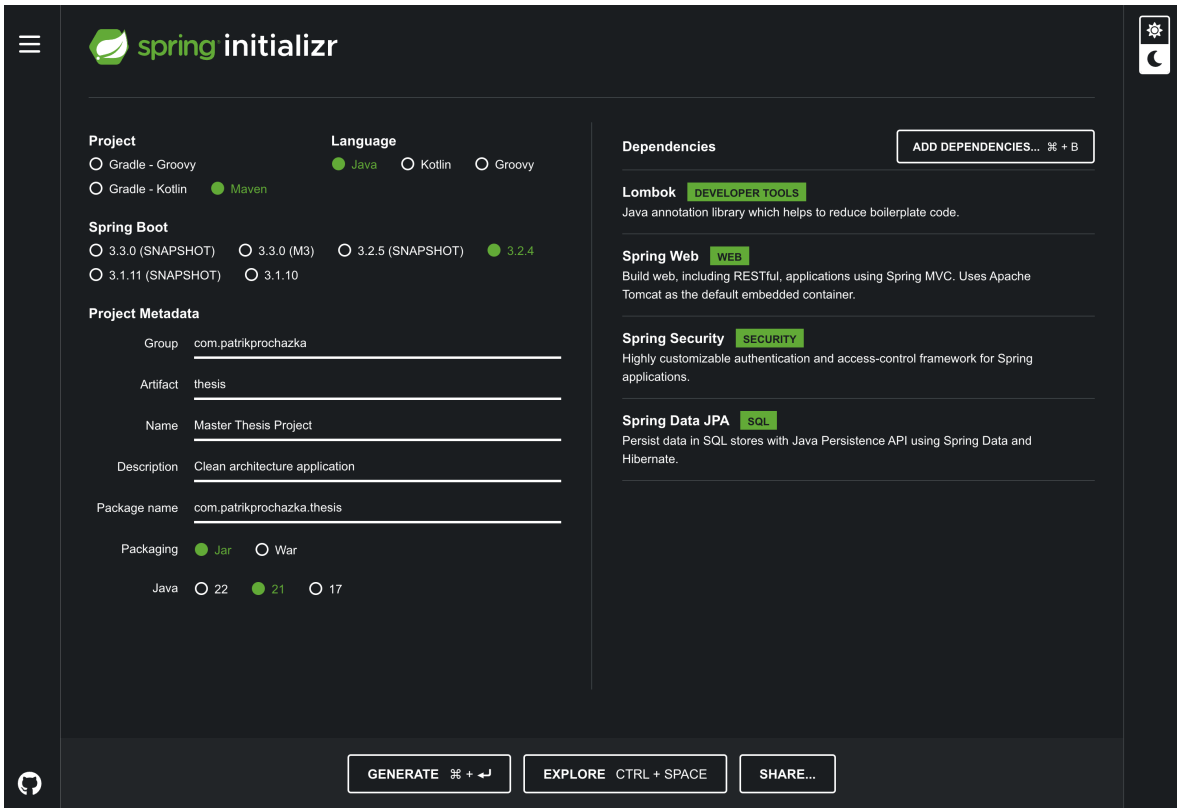
- **Singleton:** Spring kontejner implicitně spravuje beany jako singletony, což zajišťuje, že pro každý bean existuje právě jedna instance v celém životním cyklu aplikace.
- **Factory:** Je využíván v konfiguraci Springu pro tvorbu beanů, což umožňuje komplexní konfiguraci objektů bez přímé závislosti na konstruktoru objektu.

4.3 Založení Spring Boot aplikace přes Spring Initializr

Spring Initializr je webový nástroj poskytovaný projektem Spring, který slouží jako platforma pro rychlé generování základní struktury projektu pro aplikace založené na

Spring Bootu. Tento nástroj umožňuje snadno specifikovat parametry nového projektu, včetně verze Spring Bootu, programovacího jazyka (Java, Kotlin, nebo Groovy), závislostí a dalších projektových metadat. Po zadání těchto údajů Spring Initializr automaticky vytvoří základní kostru projektu, která obsahuje všechny potřebné konfigurační soubory a závislosti specifické pro zvolené nastavení. Nástroj je navržen tak, aby podporoval doporučené konvence používané ve frameworku Spring. To poskytne projektu pevný základ.

V teoretické části práce, při popisu jednotlivých technologií, byl kladen důraz na to, že Spring Boot upřednostňuje opinionated přístup, to znamená, že poskytuje předdefinované nastavení, které dodržuje standardy a vede k využívání nejlepších přístupů. Nástroj Spring Initializr je toho prvním důkazem. Na Obrázku 4.1 je vidět, jaké nastavení jsem zvolil pro tento projekt. Metadata projektu jsem vyplnil v souvislosti s touto prací, jako hlavní jazyk projektu byla zvolena Java ve verzi 21 v kombinaci s nástrojem Maven, který slouží pro řízení a automatizaci sestavení aplikace.



Obr. 4.1 Nastavení projektu v Spring Initializr

Na stejném obrázku jsou také vidět vybrané základní závislosti, které budou v projektu potřeba při vývoji. Knihovna Lombok nabízí spoustu užitečných anotací, které nahrazují boilerplate kód, jako výchozí gettery a settery, konstruktory a jiné. Spring Web bude potřeba na vystavení zdrojů, které budou proudit mezi klientem a serverem. Spring Security poskytne základní, avšak dostatečné nastavení bezpečnostních faktorů

v aplikaci. Spring Data JPA je balíček knihoven, které budou využity pro komunikaci s databází. Jedna z knihoven v Spring Data JPA je právě Hibernate.

4.4 Docker a databáze

V rámci této diplomové práce jsem se rozhodl využít technologii Docker k izolaci databázového prostředí aplikace. Docker poskytuje platformu pro kontejnerizaci, která umožňuje vývojářům distribuovat aplikace spolu s veškerými potřebnými závislostmi a knihovnami ve standardizovaných jednotkách zvaných kontejnery.[25] Jednou z hlavních výhod použití Dockeru je izolace prostředí, která zajišťuje, že aplikace bude fungovat stejně bez ohledu na lokální nastavení vývojového prostředí jednotlivých vývojářů. Toto je obzvláště důležité pro aplikace vyvíjené v týmech, kde může docházet k rozdílům v konfiguraci prostředí jednotlivých členů týmu.

Docker umožňuje jednoduché nasazení databázových služeb pomocí souborů `docker-compose.yml`, které definují veškeré parametry potřebné pro běh databáze. Tímto způsobem je možné automatizovat a standardizovat nasazení databází. Pokud vývojáři například potřebují spustit databázový server, mohou tak učinit spuštěním jednoduchého příkazu, který zaručí spuštění identické instance databáze jak na lokálním stroji, tak v produkci.

4.4.1 Integrace se Spring Boot aplikací

Pro integraci databáze běžící v Dockeru s aplikací vytvořenou pomocí frameworku Spring Boot je třeba provést několik kroků.

Vytvoření docker-compose.yml souboru Pro spuštění databáze v Dockeru je vhodné využít `docker-compose.yml`, který umožňuje definovat služby, případnou síť, ve které mezi sebou služby komunikují, a pamět na disku, kam si jednotlivé služby ukládají svoje data.

Následně je jednoduché takto přednastavenou databázi spustit pomocí příkazu `docker-compose up` v příkazovém řádku ve složce, ve které se tento soubor nachází. Další možnost, jak spustit nový kontejner, je použít aplikaci Docker Desktop nebo například vestavěné rozhraní v IntelliJ Idea (pouze ve verzi Ultimate).

Konfigurace v souboru application-psql.yaml Dalším krokem pro zprovoznění databáze je přidání několika vlastností do konfiguračního souboru `application.yml`. Přestože výchozí konfigurační soubor ve Spring Boot aplikaci je `application.properties`, tak framework podporuje i konfigurační YAML soubory. YAML na rozdíl od properties souboru umožňuje formátovat konfigurační soubory do čitelnější podoby.

Obr. 4.2 Definice docker-compose.yml

```
version: '3.8'

services:
  db:
    image: postgres:latest
    container_name: postgres_db
    environment:
      POSTGRES_DB: thesis_db
      POSTGRES_USER: patrik
      POSTGRES_PASSWORD: P@ssw0rd
    ports:
      - '5432:5432'
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

Kromě definování vlastností týkajících se `datasource` se zde nachází i nastavení pro Java Persistence API (JPA). Například pro předejití inkonzistenci dat v databázi během znovusestavení projektu je zde nastavena vlastnost `jpa.hibernate.ddl-auto` na hodnotu `create-drop`, což zapříčiní, že při každém spuštění programu se databáze i její tabulky znovu vytvoří a vymažou se všechna data. Pro produkční prostředí bude potřeba nastavit jinou strategii a změny ve schématu databáze provádět migračními skripty.

Obr. 4.3 Definice application-psql.yml

```
spring:
  datasource:
    url: ${SPRING_DATASOURCE_URL}
    username: ${SPRING_DATASOURCE_USERNAME}
    password: ${SPRING_DATASOURCE_PASSWORD}
    driver-class-name: org.postgresql.Driver
  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: true
    properties:
      hibernate:
        format_sql: false
    database: postgresql
    database-platform: org.hibernate.dialect.PostgreSQLDialect
```

Konfigurace v souboru proměnných prostředí Ve výše zmíněném konfiguračním souboru application-psql.yml si lze všimnout speciálního zápisu u některých vlastností. Tento zápis odkazuje na proměnné prostředí, které je možné programu pře-

dat při spouštění. Zatímco `${SPRING_DATASOURCE_URL}` představuje takzvaný *connection string*, který program používá k připojení k databázi, `${SPRING_DATASOURCE_USERNAME}` a `${SPRING_DATASOURCE_PASSWORD}` představují přihlašovací údaje do vytvořené databáze.

Přidání závislosti pro podporu PostgreSQL Poslední krokem v procesu napojení databáze bežící v Docker kontejneru do Spring Boot aplikace je přidání správného ovladače databáze, který na pozadí řídí veškerou komunikaci mezi programem a databází. V případě použití PostgreSQL databáze je zapotřebí přidat závislost `postgresql`.

4.5 Databázové entity

V této kapitole se zaměřím na architekturu databázových entit v aplikaci, která využívá Spring JPA a Hibernate pro mapování objektů na relační databázi, což umožňuje aplikaci efektivně komunikovat s databází bez nutnosti psát rozsáhlé množství SQL kódu. Tyto nástroje nabízí možnost pracovat s databází na vyšší úrovni abstrakce, kde lze manipulovat s databázovými entitami jako s běžnými Java objekty.

Databázové entity jsou základní stavební bloky aplikace, které definují strukturu dat a jejich vzájemné vztahy. Pro maximální efektivitu a čistotu kódu jsou tyto entity anotovány pomocí knihovny Lombok, která zjednodušuje definici třídy a redukuje boilerplate kód tím, že automaticky generuje potřebné gettery, settery a další standardní metody.

Kapitola poskytne pohled na to, jak jsou v praxi implementovány databázové entity v Javě pomocí Spring JPA a Hibernate, demonstrujíc přitom sílu a flexibilitu těchto technologií v moderním softwarovém vývoji.

4.5.1 Použité Lombok anotace

Napříč aplikací je možné nalézt použité Lombok anotace. Tyto se vztahují přímo k databázovým entitám:

- **@Data:** Anotace, která generuje automaticky gettery a settery, stejně jako metody `hashCode()`, `equals()` a `toString()`.
- **@Builder:** Umožňuje použití návrhového vzoru *builder* pro tvorbu instance dané třídy.
- **@NoArgsConstructor:** Generuje konstruktor bez argumentů.
- **@AllArgsConstructor:** Generuje kontruktory s argumenty pro všechny atributy třídy.

4.5.2 Entita User

Entita `User` je v aplikaci zásadní pro správu uživatelských účtů a autentizačních procesů. Protože tato databázová entita obsahuje citlivá data, bylo potřeba některé atributy anotovat pomocí `@JsonIgnore`, které zapříčiní, že při serializaci objektu do JSON výstupu, například na úrovni REST API, jsou tyto atributy vynechány. Jako primární klíč je zvolen atribut `email`, který už z principu musí být unikátní.

Entita `User` implementuje rozhraní `UserDetails`, které poskytuje základní informace potřebné pro autentizaci a autorizaci ve Spring Security, viz Obrázek 4.4. Tyto principy jsou popsány v kapitole **4.5**.

4.5.3 Entita Todo

Entita `Todo` je v aplikaci hlavní reprezentace úkolů s několika vazbami na jiné entity (TODO odkaz na ERD). Stejně jako `User` entita je implementována s využitím Lombok a JPA anotací. Zvláště užitečná je zde anotace `@Builder`, která zjednodušuje práci s entitou, která má většinu polí nepovinnou, bez nutnosti vytvářet mnoho různých konstruktorů.

Stejně jako Lombok anotace pomáhají snižovat boilerplate kód, JPA anotace umožňují to stejné pro tvorbu vazeb mezi entitami. Využité anotace na vytvoření vztahu mezi entitami:

- **@ManyToOne:** Pomocí této anotace lze vytvořit 1:N vazbu mezi entitami. Lze kombinovat s anotací `@JoinColumn`.
- **@ManyToMany:** Pomocí této anotace lze vytvořit M:N vazbu. V kombinaci s touto anotací je potřeba nastavit i vazební tabulku přes anotaci `@JoinTable`, viz Obrázek 4.5.
- **@JoinColumn:** Definuje sloupec tabulky, který je následně použit v generovaném SQL dotazu v JOIN klauzuli. Například při čtení Todo entity, která má vazbu u atributu `assignee` na sloupec `assignee_email`:
  `left join users a1_0 on a1_0.email=t1_0.assignee_email`,
  kdy `a1_0` je alias pro tabulku napojenou na sloupec `assignee` a alias `t1_0` reprezentuje Todo entitu.

4.5.4 Entity Category a Tag

Entity `Category` a `Tag` jsou navrženy pro kategorizaci a označování obsahu v aplikaci, což usnadňuje organizaci a vyhledávání informací. Obě entity opět využívají Lombok a JPA anotace.

Obr. 4.4 Definice entity User implementující rozhraní UserDetails

```
@Builder
@Data
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "users")
public class User implements UserDetails {

    @Id
    @Column(name = "email", nullable = false)
    @JsonIgnore
    private String email;

    @JsonIgnore
    @Column(name = "password", nullable = false)
    private String password;

    @Enumerated(EnumType.STRING)
    private UserRole role;

    @JsonIgnore
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role.name()));
    }

    @Override
    public String getUsername() { return email; }

    @JsonIgnore
    @Override
    public boolean isAccountNonExpired() { return true; }

    @JsonIgnore
    @Override
    public boolean isAccountNonLocked() { return true; }

    @JsonIgnore
    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @JsonIgnore
    @Override
    public boolean isEnabled() { return true; }
}
```

Pro potřeby ukázkové aplikace jsou jednoduché a mají podobnou strukturu a funkcionalitu. Daly by se ovšem rozšířit. Například entita `Tag` by mohla mít další atribut `color` typu `String`, který by v sobě ukládal nastavení barvy tagu, kterou si uživatel nastavil, v hexadecimální reprezentaci.

Obr. 4.5 Definice M:N vazby mezi entitami Todo a User

```
@ManyToMany(fetch = FetchType.LAZY)
@JoinTable(name = "todo_collaborators",
        joinColumns = @JoinColumn(name = "todo_id"),
        inverseJoinColumns = @JoinColumn(name = "user_email"))
private List<User> collaborators;
```

4.5.5 Výčtové typy

Výčtové typy jako `TodoStatus` a `TodoPriority` přinášejí významné výhody pro správu a integritu dat v aplikacích. Díky jejich použití se zvyšuje bezpečnost typů, protože jsou hodnoty omezeny na pevně stanovenou množinu, což minimalizuje chyby z nesprávného zadání dat. Tato omezení zároveň zlepšují čitelnost kódu, neboť všechny možné hodnoty jsou jasně definovány a centralizovaně spravovány. Při potřebě refaktoringu kódu umožňují výčtové typy jednoduché a bezpečné aktualizace bez rizika narušení datové integrity, což je zásadní při změnách ve struktuře dat. Integrace těchto typů s databázemi je navíc efektivně zjednodušena pomocí anotace `@Enumerated` v JPA a Hibernate, což usnadňuje mapování výčtových hodnot na databázové sloupce a zlepšuje správu databázových operací souvisejících s těmito atributy.

4.6 Autorizace a autentizace uživatele

V současném digitálním světě, kde bezpečnost aplikací hraje klíčovou roli, je nezbytné použít robustní autentizační mechanismy, které ochrání citlivá data a zároveň poskytují flexibilní a efektivní přístup k uživatelským službám. Tato kapitola se zaměřuje na vysvětlení implementace JSON Web Token (JWT) autentizace v moderní webové aplikaci a na to, jak lze tyto technologie integrálně využít pro zajištění bezpečné i škálovatelné autentizace a autorizace uživatelů.

Spring Boot poskytuje vynikající základ pro rychlý vývoj s minimálním nastavením. V kombinaci se Spring Security, který nabízí komplexní bezpečnostní infrastrukturu, vzniká robustní základna pro moderní webové aplikace. Avšak pro integraci JWT autentizace je potřeba vytvořit několik tříd, které rozšiřují nebo implementují novou logiku, která je vložena do bezpečnostních procesů frameworku.

TODO - Diagram jak funguje jednotlivé komponenty JWT ověření.

4.6.1 Třída ApplicationConfiguration

Třída `ApplicationConfiguration` je základním stavebním kamenem pro autentizační infrastrukturu v aplikaci. Tato třída využívá anotaci `@Configuration`, která indikuje, že se jedná o zdroj definice beanů pro Spring IoC kontejner. Spolupracuje s několika

základními službami Spring Security, Spring Data JPA a je klíčovou součástí pro správnou funkci autentizace v aplikaci. Následující prvky musí být definovány v rámci této konfigurační třídy pro správné fungování zpracovávání tokenů.

UserDetailsService Metoda `userDetailsService` definuje bean výše zmíněného typu, který je základem pro načítání dat o uživateli. Tato služba je nezbytná pro autentizaci, neboť poskytuje mechanismus pro získání uživatelských dat z databáze na základě uživatelského jména (v případě této aplikace se jedná o e-mailovou adresu).

AuthenticationProvider Metoda `authenticationProvider` vytváří bean specifického typu `DaoAuthenticationProvider`. Tento bean je konfigurován tak, aby používal mnou vytvořenou `userDetailsService` pro získávání informací o uživatelích a `passwordEncoder` pro ověření hesel, což přispívá k robustnosti autentizačního procesu.

AuthenticationManager Metoda `authenticationManager` vytváří bean typu `AuthenticationManager`. Tento bean je vytvořen na základě `AuthenticationConfiguration`. Jedná se o předdefinovanou konfiguraci, kterou poskytuje Spring Security. Tato metoda zjednodušuje konfiguraci autentizačního procesu a poskytuje standardizovaný způsob pro zpracování autentizačních požadavků v rámci aplikace.

PasswordEncoder Bean `passwordEncoder` definovaný v metodě `passwordEncoder` je instancí `BCryptPasswordEncoder`, což je moderní a bezpečný způsob šifrování hesel. Použití tohoto kodéru zajišťuje, že hesla uložená v databázi nebudou v čitelné formě, což zvyšuje bezpečnost uživatelských účtů proti útokům, jako je například únik dat.

Třída `ApplicationConfiguration` tak poskytuje základní konfiguraci nezbytnou pro bezpečnou a efektivní autentizaci uživatelů v aplikaci. Každý z těchto *beanů* hraje specifickou roli v autentizačním procesu, od získávání a ověřování uživatelských dat až po zpracování autentizačních požadavků, což zajišťuje, že aplikace je schopna správně a bezpečně identifikovat a autorizovat své uživatele.

4.6.2 Třída WebSecurityConfiguration

Tato třída je taktéž klíčovou součástí zabezpečení webové aplikace využívající Spring Security. Anotacemi `@Configuration` a `@EnableWebSecurity` upravuje a rozšiřuje základní bezpečnostní konfigurace poskytované Spring Security, aby vyhovovala specifickým potřebám aplikace pro využití JWT. Ve třídě jsou definovány a konfigurovány různé bezpečnostní komponenty, které společně zajišťují integrovanou ochranu proti běžným hrozbám.

Třída SecurityFilterChain Metoda `securityFilterChain` poskytuje bean typu `SecurityFilterChain`, který je základem pro definici bezpečnostních pravidel pro HTTP komunikaci v aplikaci. Pomocí objektu `HttpSecurity`, který je parametrem této metody, se konfiguruje celý řetězec bezpečnostních nastavení a filtrů aplikace.

- **Deaktivace CSRF ochrany:** Toto je běžná praxe u aplikace, která používá tokeny (jako JWT) pro ověření uživatelů, protože tyto tokeny samy o sobě poskytují ochranu proti útokům CSRF.
- **Konfigurace přístupů:** Definice pravidel pro různé URL. Běžná praxe je, že URL, které slouží pro autentizaci uživatelů, jsou volně přístupné. Dále mohou být volně přístupné URL, které poskytují metadata o API. Ostatní URL vyžadují autentizaci.
- **Session management:** Konfigurace také určuje, že uživatelská session bude bezstavová, což je základní předpoklad pro RESTful API[26], kde se stav uživatele neukládá mezi požadavky.
- **Přidání vlastního filtru:** Vložení `JwtAuthenticationFilter` před standardní filtr pro zpracování uživatelských jmen a hesel zapříčiní, že každý příchozí požadavek je nejprve zkontrolován na přítomnost a platnost JWT, což umožňuje efektivní a bezpečné zpracování autentizace. Zároveň je tento filtr zásadní pro zpracování tokenu v rámci celého bezpečnostního řetězce.
- **Integrace vlastní implementace AuthenticationProvider:** Zapojení beanu `authenticationProvider`, který byl konfigurován v `ApplicationConfiguration`, do kontextu Spring Security zajišťuje, že proces ověřování uživatelských údajů je prováděn podle specifikovaných pravidel a použití vlastní logiky.

Třída `WebSecurityConfiguration` efektivně zajišťuje, že aplikace je chráněna proti neoprávněnému přístupu, přičemž poskytuje potřebnou flexibilitu pro konfiguraci a rozšíření bezpečnostních pravidel. Integrace těchto komponent do jednotného bezpečnostního modelu podporuje vysoký standard zabezpečení, který je nezbytný pro moderní webové aplikace.

4.6.3 Třída JwtAuthenticationFilter

`JwtAuthenticationFilter` je specifický bezpečnostní filtr, který zajišťuje zpracování JWT pro každý příchozí HTTP požadavek. Jde o prvek pro ověřování uživatelských identit v aplikacích, které využívají bezstavové autentizační mechanismy. Tento filtr je anotován jako `@Component`, což umožňuje jeho automatické detekování a správu Spring

IoC kontejnerem. Implementace OncePerRequestFilter zaručuje, že tento filtr bude aplikován právě jednou na každý příchozí požadavek.

Ve funkci `doFilterInternal` filtr nejprve získá JWT z HTTP hlavičky `Authorization`. Pokud hlavička existuje a začíná prefixem `Bearer`, filtr extrahuje token, který následuje po tomto prefixu. Pokud hlavička chybí nebo není správně formátovaná, filtr umožní, aby požadavek pokračoval bez dalšího zásahu, což v praxi znamená, že požadavek nebude mít přiřazenou žádnou uživatelskou identitu, a pokud URL, na kterou požadavek míří, vyžaduje identitu, skončí požadavek HTTP statusem `403 Forbidden`.

Autentizace uživatele se provádí ve dvou krocích za pomoci definované logiky ve třídě `JwtService`. Prvně se z tokenu extrahuje informace o uživateli, respektive e-mail uživatele, poté se ověří, zda není uživatel již autentizován. Pokud není, pokračuje se v načítání dat o uživateli pomocí beany `userDetailsService`. Následně filtr ověří platnost tokenu. Kontroluje se, zda token nevypršel a zda odpovídá údajům o uživateli. Pokud je token platný, vytvoří se nový `UsernamePasswordAuthenticationToken`, který je následně nastaven jako autentizace v bezpečnostním kontextu aplikace pomocí třídy `SecurityContextHolder`.

`JwtAuthenticationFilter` tedy hraje zásadní roli v zabezpečení aplikace, kde funguje jako první obranná linie pro autentizaci uživatelů na základě JWT. Správná implementace a konfigurace takového filtru jsou klíčové pro ochranu citlivých částí aplikace a pro zajištění, že uživatelé jsou řádně ověřeni před přístupem k chráněným zdrojům. Tento filtr je příkladem efektivního spojení bezpečnostních praxí s technologickými řešeními Spring Security.

4.6.4 Rozhraní JwtService

Rozhraní `JwtService` je dalším prvkem v architektuře JWT autentizace, poskytující klíčové metody pro manipulaci s JWT. Toto rozhraní definuje soubor operací, které jsou nezbytné pro generování, extrakci informací a validaci JWT, viz Obrázek 4.6. Implementace tohoto rozhraní zajišťuje, že tokeny jsou správně vytvářeny, ověřovány a zpracovávány v kontextu zabezpečení aplikace.

Metody dle jejich účelu lze seskupit do tří kategorií:

- **Generování JWT:** Za pomoci přetížení metody `generateToken` třída poskytuje dvě možnosti generování JWT. Metoda umožňuje generovat nový JWT s možností přidání dodatečných tvrzení (claims). Tvrzení mohou obsahovat další informace o uživateli nebo o session, které nejsou přímo součástí standardního `UserDetails` objektu. Generovaný token obsahuje uživatelské jméno jako subjekt a je podepsán pomocí bezpečného klíče. Následně pak zjednodušená verze metody

Obr. 4.6 Definice rozhraní JwtService

```
public interface JwtService {
    String extractUsername(String token);

    <T> T extractClaim(String token, Function<Claims, T>
       claimsResolver);

    String generateToken(Map<String, Object> extraClaims, UserDetails
       userDetails);

    String generateToken(UserDetails userDetails);

    boolean isValidToken(String token, UserDetails userDetails);

    boolean isTokenExpired(String token);
}
```

pro generování tokenů, která používá pouze základní informace z `UserDetails` bez dodatečných tvrzení (claims).

- **Extrakce informací z JWT:** Základní metodou s tímto účelem je metoda `extractUsername`, která získá uživatelské jméno z tokenu. V případě této aplikace se jedná o e-mail uživatele. Poté je tu generická metoda `extractClaim`, která umožňuje extrakci libovolného tvrzení z JWT pomocí předané callback funkce. Tato flexibilita je užitečná pro získání specifických dat z tokenu, která mohou být potřebná pro validaci nebo další zpracování.
- **Validace JWT:** `isTokenValid` ověřuje, zda je token platný v kontextu daného objektu `UserDetails`. To zahrnuje ověření, že uživatelské jméno v tokenu odpovídá uživatelskému jménu v `UserDetails` a že token nevypršel. Ke kontrole vypršení tokenu slouží metoda `isTokenExpired`, která kontroluje, zda má token nastavený čas expirace v minulosti.

V rámci implementace rozhraní `JwtService` je naprogramována logika generování klíče pro podepisování JWT, což je zásadní krok v zajištění integrity a autenticity tokenů. Využívá se HMAC s algoritmem SHA-256 (HS256), což je symetrický algoritmus, kde stejný klíč slouží jak pro generování, tak pro ověřování podpisu. Klíč je odvozen z tajného řetězce konfigurovaného v proměnných daného prostředí, což zajišťuje, že pouze držitelé tohoto klíče mohou validně podepisovat nebo ověřovat JWT. Tento přístup poskytuje vysokou úroveň bezpečnosti, protože SHA-256 je odolný proti kolizím, což znamená, že je prakticky nemožné najít dva různé vstupy, které by vedly k témuž hashi, a tedy manipulovat s tokenem bez znalosti tajného klíče.[27] Takový mechanismus podepisování je široce uznáván a doporučován pro svou bezpečnost a efektivitu v ochraně dat.

`JwtService` hraje důležitou roli v zabezpečení aplikace, poskytuje esenciální metody pro práci s JWT a zajišťuje, že tokeny jsou vytvářeny, ověřovány a správně zpracovány v rámci bezpečnostní architektury aplikace.

4.6.5 Rozhraní AuthenticationService

Rozhraní `AuthenticationService` je součástí vrstvy byznys logiky, která zajišťuje autentizační procesy v aplikaci. Zároveň je však také nedílnou součástí celého procesu autorizace a autentizace uživatele v aplikaci. Toto rozhraní definuje základní operace pro registraci a přihlášení uživatelů, čímž umožňuje správu uživatelských účtů v kontextu zabezpečení.

Obr. 4.7 Definice rozhraní AuthenticationService

```
public interface AuthenticationService {
    AuthenticationResponse register(AuthRequest authRequestDto);

    AuthenticationResponse login(AuthRequest authRequestDto);
}
```

Implementace rozhraní `AuthenticationService` zajišťuje jeho funkčnost a roli v byznys logice prostřednictvím několika komponent:

- **UserRepository:** Komponenta pro interakci s databází uživatelů.
- **PasswordEncoder:** Komponenta pro hashování hesel, což zvyšuje bezpečnost ukládání hesel.
- **JwtService:** Služba pro generování a validaci JWT, která slouží pro správu session po autentizaci.
- **AuthenticationManager:** Spring Security komponenta zodpovědná za ověření přihlašovacích údajů.

Použití rozhraní a služeb jako `PasswordEncoder` a `JwtService` podporuje principy bezpečného designu a umožňuje snadné rozšíření funkcionalit bez nutnosti měnit existující kód, což je v souladu s Open Closed Principle.[28] Celkově je tato implementace dobře integrována do Spring Security ekosystému, což přináší vysokou míru spolehlivosti a bezpečnosti.

4.7 Byznys logika aplikace

V aplikaci je vrstva byznys logiky klíčová pro oddělení byznys pravidel aplikace od uživatelského rozhraní a infrastruktury. Tato vrstva zahrnuje definici rozhraní, která

specifikují operace, jež mohou být provedeny v kontextu daného byznys scénáře. Rozhraní jako `AuthenticationService`, `JwtService` a `TodoService` jsou umístěna právě zde, protože poskytují abstrakci, která umožňuje formulaci byznys pravidel nezávisle na vnějších faktorech, jako jsou technologie pro persistenci dat nebo specifika uživatelského rozhraní.

Umístění těchto rozhraní v *use case* vrstvě podporuje principy SOLID[29], zvláště princip jednotné odpovědnosti a princip otevřenosti/zavřenosti, což zvyšuje možnost znovupoužití kódu a usnadňuje testování jednotlivých komponent izolovaně od zbytku systému.[28] Například `TodoService` definuje operace specifické pro manipulaci s úkoly, zatímco již popsané rozhraní `AuthenticationService` řeší autentizaci uživatelů, a taktéž již popsané rozhraní `JwtService` správu JWT. Každé z těchto rozhraní má jasně definovaný účel a společně tvoří základní byznys logiku aplikace.

4.7.1 Rozhraní TodoService

Rozhraní `TodoService` představuje součást vrstvy byznys logiky, která má na starosti správu úkolů. Toto rozhraní definuje sadu operací, které lze provádět s úkoly.

Obr. 4.8 Definice rozhraní TodoService

```
public interface TodoService {
    List<Todo> getTodos(String username);

    List<Todo> getSortedTodos(String username, String sortType,
       Boolean ascending);

    Todo createTodo(CreateTodoDTO todo, String username);

    Todo getTodo(Long id, String username);

    Todo updateTodo(UpdateTodoDto todo, String username);

    Todo markCompleted(Long id, String username);

    Todo deleteTodo(Long id, String username);
}
```

`TodoServiceImpl` je konkrétní implementací rozhraní `TodoService`, která používá Spring anotaci `@Service` k označení, že je to komponenta spravovaná Spring kontejnerem. Tato třída je zodpovědná za logiku spojenou s operacemi definovanými v rozhraní a pracuje přímo s datovou vrstvou a dalšími službami:

- **TodoRepository:** Představuje datovou vrstvu, která zprostředkovává operace s databází úkolů s využitím Spring JPA.

- **TagService, UserService, CategoryService:** Tyto služby zprostředkovávají operace, které zahrnují práci s dalšími entitami, jako jsou tagy, uživatelé a kategorie, které jsou asociovány s úkoly.

Třída `TodoServiceImpl` také obsahuje logiku pro ověření oprávnění uživatelů k provedení operací na úkolech, což zajišťuje, že úkoly mohou být zobrazovány nebo modifikovány pouze oprávněnými uživateli. Oprávněný uživatel v rámci této implementace je takový uživatel, který je buď autorem úkolu, je mu úkol přiřazen, anebo je jedním ze spolupracovníků.

Obr. 4.9 Definice privátní metody checkIfUserIsAuthorized

```
private void checkIfUserIsAuthorized(Todo todo, User user) {
    if (!(todo.getCreatedBy().equals(user)
            || todo.getCollaborators().contains(user)
            || todo.getAssignee().equals(user))) {
        throw new SecurityException("You are not authorized to perform
            this action");
    }
}
```

Rozhraní `TodoService` a jeho implementace prostřednictvím `TodoServiceImpl` představují zásadní komponenty byznys logiky aplikace pro správu úkolů. Implementační třída integrálně spolupracuje s dalšími komponentami systému, což značně zvyšuje koherenci a modulárnost aplikace.

Díky abstrakci, kterou rozhraní poskytuje, je možné systém snadno rozšiřovat a adaptovat na změněné požadavky uživatelů bez nutnosti zásahů do stabilního jádra aplikace, což je v souladu s principy čistého kódu a dobrých programátorských praktik.[30] Tímto způsobem `TodoService` a `TodoServiceImpl` nejen podporují agilní vývojové procesy, ale také přispívají k dlouhodobé udržitelnosti a rozšiřitelnosti celého softwarového řešení.

4.7.2 Rozhraní SortingStrategy

Rozhraní `SortingStrategy` a jeho implementace představují součást aplikace, která reflektuje definované požadavky a umožňuje flexibilní řazení úkolů podle různých kritérií. Tento modulární přístup zvyšuje znovupoužitelnost kódu a usnadňuje údržbu systému tím, že odděluje logiku řazení od ostatních částí aplikace. Je to praktické využití návrhového vzoru *Strategy*.

`SortingStrategy` je jednoduché rozhraní definující jedinou metodu, a to `sort`, viz Obrázek 4.10.

Následné implementace tohoto rozhraní přináší do aplikace logiku řazení. Všechny implementace jsou anotovány pomocí `@Component("nazev_strategie")`, což umožňuje

Obr. 4.10 Definice rozhraní SortingStrategy

```
public interface SortingStrategy {
    List<Todo> sort(List<Todo> todos, boolean ascending);
}
```

jejich snadné začlenění do Spring kontejneru a využití v rámci aplikace. Řazení je řešeno třídou `Comparator`, což je funkční rozhraní, které umožňuje definovat metodu porovnání dvou objektů typu `T`, aby bylo možné určit jejich uspořádání. Používá se hlavně pro řazení nebo porovnávání objektů, když přirozené uspořádání definované rozhraním `Comparable` a jeho metodou `compareTo` není dostatečné nebo vhodné.

Jednotlivé implementace rozhraní `SortingStrategy`:

- **ByCategorySortingStrategy:** Tato strategie řazení seřadí úkoly podle kategorie. Používá `Comparator`, který srovnává názvy kategorií úkolů.
- **ByDeadlineSortingStrategy:** Řadí úkoly podle termínu splnění. Komparátor porovnává objekty typu `LocalDataTime`.
- **ByPrioritySortingStrategy:** Řadí úkoly podle jejich priority, přičemž priorita je reprezentována výčtovým typem a každá hodnota priority má přiřazen ordinální index.

Rozhraní `SortingStrategy` a jeho specifické implementace jsou ideálním příkladem aplikace principů čistého kódu a efektivního designu softwaru. Každá strategie je uzavřená jednotka s jasně definovanou odpovědností, což usnadňuje testování, údržbu a rozšiřování aplikace. Řešení implementací opět plně odpovídá principům *Open/Closed* a *Single Responsibility* ze SOLID principů.[29]

4.8 REST API Controllery

REST API controllery slouží jako most mezi uživatelským rozhraním a business logikou. V rámci Clean Architecture patří controllery do prezentační vrstvy zvané "Interface Adapters"[3]. Jejich hlavní úlohou je komunikovat s klienty (např. webovými prohlížeči nebo mobilními aplikacemi). Controllery přijímají data z HTTP požadavků, často ve formě DTO (Data Transfer Objects), a předávají je do nižších vrstev aplikace, jako jsou služby, které implementují byznys logiku. Controllery zároveň zpracovávají odpovědi od služeb a formátují je zpět jako HTTP odpovědi.

V kontextu Spring Boot aplikací jsou controllery implementovány s anotacemi jako `@RestController`, které Spring Framework interpretuje k vytvoření HTTP endpointů. Tento způsob deklarace endpointů a závislostí je příkladem "Inversion of Control"(IoC), což je další princip, který se snaží Clean Architecture podporovat.

4.8.1 Role DTO v Clean Architecture

DTO jsou klíčovou součástí vrstvy *Interface Adapters*. Slouží ke zjednodušení a bezpečnému přenosu dat mezi různými částmi aplikace, zejména mezi klientem a serverem.[31] Využití DTO umožňuje oddělit interní reprezentaci dat v aplikaci od dat odesílaných klientovi nebo přijímaných od klienta. Tímto způsobem lze snadno změnit datový model bez dopadu na externí API. DTO také pomáhají v implementaci principu *Single Responsibility Principle* tím, že odstíní složitosti doménových modelů od klientů a soustředí se jen na data potřebná pro konkrétní operace. Navíc použití DTO napomáhá v čitelnosti a údržbě kódu, což je v souladu s principy Clean Code[30], protože každý objekt má jasně definovanou roli a odpovědnost.

V této aplikaci jsou data z HTTP požadavků převáděna na DTO s použitím anotace `@RequestBody`. Anotace `@RequestBody` je používána k označení parametru metody v controlleru, který by měl být naplněn daty z těla HTTP požadavku. Když server přijme HTTP požadavek (např. *POST* nebo *PUT*), tělo tohoto požadavku, obvykle ve formátu JSON, je automaticky deserializováno do odpovídajícího objektu, který je definován jako parametr metody. Tento proces využívá knihovny jako Jackson nebo Gson, které Spring integruje pro převod JSON dat do DTO. Tento mechanismus umožňuje snadné a bezpečné zpracování přijatých dat bez nutnosti manuálního parsování, což zjednodušuje práci s daty a zvyšuje bezpečnost aplikace.

4.8.2 Použité anotace

Anotace `@RestController` v rámci Spring frameworku označuje třídu jako speciální typ controlleru, který je připraven ke zpracování webových požadavků. Tato anotace je derivátem `@Controller`, s tím rozdílem, že všechny metody vracející hodnoty mají své odpovědi automaticky serializovány přímo do těla HTTP odpovědí, typicky ve formátu JSON. Toto chování je zvláště užitečné pro vývoj RESTful webových služeb, kde je potřeba jednoduše mapovat HTTP požadavky na metody Java tříd.

Anotace `@RequestMapping` a parametr uvnitř anotace (například "/api/v1/auth") definují základní URL cestu, která se aplikuje na všechny metody v daném controlleru. To umožňuje strukturovaný a přehledný přístup ke směrování požadavků.

4.8.3 Verzování API

Ve vývoji softwarových aplikací je důležité zabezpečit kompatibilitu mezi různými verzemi API, což zajišťuje hladkou integraci pro vývojáře i uživatele. Verzování API cest, například pomocí prefixu `api/v1/`, je klíčovou technikou, která umožňuje efektivně spravovat změny v API bez narušení funkčnosti existujících systémů. Tímto způsobem může být starší verze API nadále podporována, zatímco nové funkce a vylepšení jsou

zaváděny ve verzi novější.[32] To je obzvlášť užitečné ve velkých projektech, kde různé komponenty mohou být aktualizovány nezávisle na sobě.

Důležitost verzování API se také projevuje v možnosti provádět bezpečnostní opravy a změny v datových modelech bez přerušení služeb pro koncové uživatele. Pokud například objevíme kritickou chybu nebo potřebu změnit strukturu databáze, můžeme implementovat tyto změny v nové verzi API a nastavit prefix na `api/v2/`, zatímco starší verze `api/v1/` zůstává stabilní a dostupná pro klienty, kteří ještě nepřešli na aktualizovanou verzi. Verzování API tak poskytuje nezbytnou flexibilitu pro správu životního cyklu aplikací a minimalizuje rizika spojená se změnami v systému.[32]

4.8.4 Třída AuthenticationController

`AuthenticationController` slouží jako primární vstupní bod pro autentizační funkce aplikace, viz Obrázek 4.11. Jeho úkolem je zjednodušit proces autentizace přes REST API, což zahrnuje ověřování uživatelských údajů a generování JWT tokenů, které slouží pro předávání informací o session a autorizaci uživatelů v rámci aplikace. Tento controller je jedním z prvků, které za pomoci Spring Security zabezpečují aplikaci.

Metody, které `AuthenticationController` obsahuje, jsou:

- **login:** Tato metoda zpracovává POST požadavek na cestu `/login`. Přijímá údaje potřebné pro přihlášení uživatele, jako jsou e-mail a heslo, zabalené ve třídě `AuthRequestDTO`. Po úspěšném ověření uživatele službou `AuthenticationService` vrátí HTTP odpověď s tokenem, který je reprezentován objektem třídy `AuthenticationResponseDTO`.
- **register:** Tato metoda zpracovává POST požadavek na cestu `/register`. Umožňuje novým uživatelům zaregistrovat se do systému. Přijaté údaje (e-mail a heslo) reprezentované jako `AuthRequestDTO` jsou opět předány do služby `AuthenticationService` pro zpracování registrace. Po úspěšné registraci vrátí HTTP odpověď s tokenem, taktéž jako objekt třídy `AuthenticationResponseDTO`.

4.8.5 Přidání informací o uživateli do kontextu požadavků

Anotace `@ModelAttribute` se ve frameworku Spring používá k automatickému přidání atributů do modelu, který je dostupný pro metody v controllerech. Tato anotace může být použita na metodách ve třídách anotovaných `@Controller`, což umožňuje centralizované zpracování určitých dat předtím, než jsou zpracována v controllerech.

V kontextu třídy `UserIdInjector`, která je anotována s `@ControllerAdvice`, se anotace `@ModelAttribute` používá k přidání atributu `user`, který obsahuje uživatelské

Obr. 4.11 Definice třídy AuthenticationController

```
@RestController
@RequestMapping("/api/v1/auth")
@RequiredArgsConstructor
public class AuthenticationController {

    private final AuthenticationService authenticationService;

    @PostMapping("/login")
    public ResponseEntity<AuthenticationResponseDTO> login(
        @RequestBody AuthRequestDTO request) {
         return ResponseEntity.ok(
             authenticationService.login(request)
         );
    }

    @PostMapping("/register")
    public ResponseEntity<AuthenticationResponseDTO> register(
        @RequestBody AuthRequestDTO request) {
         return ResponseEntity.ok(
             authenticationService.register(request)
         );
    }
}
```

jméno získané z JWT, do modelu každého požadavku. Tento atribut je pak dostupný pro všechny metody controlleru, které mohou tuto hodnotu potřebovat.

Proces funguje tak, že metoda `injectUser` je automaticky volána při každém HTTP požadavku, který je zpracováván aplikací. Metoda nejprve zkontroluje, zda existuje hlavička `'Authorization'` s JWT. Pokud ano, extrahuje token, ověří, zda nevypršel, a pomocí implementace rozhraní `JwtService` získá uživatelské jméno z tokenu. Toto uživatelské jméno je poté přidáno do modelu HTTP requestu pod klíčem `user`, viz Obrázek 4.12.

Využití `@ModelAttribute` v kombinaci s `@ControllerAdvice` poskytuje efektivní způsob, jak zajistit, že určité klíčové informace jsou dostupné napříč různými částmi aplikace bez potřeby duplicity kódu v každém controlleru. To značně zjednodušuje údržbu kódu a zvyšuje jeho přehlednost, zatímco zůstává bezpečným způsobem, jak manipulovat s uživatelskými daty v kontextu autentizace a autorizace.

4.8.6 Třída TodoController

`TodoController`, obdobně jako předchozí controller, slouží jako vstupní bod aplikace pro práci s úkoly. Controller zajišťuje, že operace jako vytváření, aktualizace, mazání a načítání úkolů jsou prováděny bezpečně a efektivně. Metody jsou navrženy tak, aby reflektovaly běžné CRUD (Create, Read, Update, Delete) operace, což usnadňuje uži-

Obr. 4.12 Definice třídy UserIdInjector

```
@ControllerAdvice
@RequiredArgsConstructor
public class UserIdInjector {

    private final JwtService jwtService;

    @ModelAttribute("user")
    public String injectUser(HttpServletRequest request) {
        String authHeader = request.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String token = authHeader.substring(7);
            if (!jwtService.isTokenExpired(token)) {
                return jwtService.extractUsername(token);
            }
        }
        return null;
    }
}
```

vatelům správu jejich úkolů. Zmíněné operace jsou rozděleny do metod controlleru, přičemž je do každé metody vložen uživatel získaný z JWT tokenu. Metody controlleru jsou tyto:

- **getTodos:** Načítá seznam všech úkolů pro daného uživatele.
- **getSortedTodos:** Vrací seznam úkolů seřazený podle zadaného kritéria a způsobu řazení. Rozhodnutí, jaká řadící strategie se použije, závisí na parametrech z URL adresy, kontrétně `sortBy` a `asc`.
- **getTodoById:** Získává data pro jeden úkol, pokud je k úkolu uživatel autorizován. Identifikátor požadavku je získán z URL adresy za pomoci anotace `@PathVariable("id")`.
- **createTodo:** Vytváří nový úkol podle dat poskytnutých v těle požadavku a přiřazuje jej uživateli, který poslal požadavek. Data z těla požadavku jsou převedena do objektu `CreateTodoDTO` pro další zpracování.
- **updateTodo:** Aktualizuje údaje existujícího úkolu podle dat zaslaných v požadavku. Protože se operace týká konkrétního objektu, je zde opět použit parametr v cestě s názvem `id`. Data z těla požadavku jsou převedena do objektu `UpdateTodoDTO` pro další zpracování.
- **completeTodo:** Označuje úkol s předaným identifikátorem z URL adresy jako dokončený.

- **deleteTodo:** Označuje úkol s předaným identifikátorem z URL adresy jako smazaný.

Kompletní implementaci třídy `TodoController` je možné si prohlédnout v přiložených zdrojových kódech v Příloze 1.

4.8.7 Třída CategoryController

`CategoryController` slouží jako vstupní bod pro práci s kategoriemi. Controller poskytuje operace pro čtení kategorií, na které má uživatel právo, vytváření nové kategorie a mazání kategorie. Stejně jako `TodoController` využívá v `CategoryController` logiku na získávání uživatele z JWT tokenu. Definované metody controlleru jsou tyto:

- **getAllCategoriesForUser:** Tato metoda získává všechny kategorie spojené s daným uživatelem.
- **createCategory:** Tato metoda zpracovává požadavky na vytvoření nové kategorie. Používá `@RequestBody` pro deserializaci těla požadavku do objektu `CreateCategoryDTO`, který obsahuje data potřebná pro vytvoření nové kategorie.
- **deleteCategory:** Tato metoda zpracovává požadavky na odstranění kategorie na základě jejího identifikátoru.

4.9 Rozhraní pro čtení a zápis dat z databáze

Repositáře hrají klíčovou roli jako adaptéry mezi doménovou vrstvou a databází. V případě aplikací využívajících Spring JPA, rozhraní `JpaRepository` poskytuje elegantní a výkonný způsob pro práci s daty, který je přizpůsobený jak potřebám ORM, tak i principům Spring ekosystému.

`JpaRepository` je rozhraní ve Spring Data, které dědí od `PagingAndSortingRepository` a nakonec od `CrudRepository`. To poskytuje bohatou sadu metod pro CRUD operace a také pro stránkování a řazení, což je zásadní pro efektivní manipulaci s velkými objemy dat. Implementace tohoto rozhraní umožňuje aplikacím snadno a efektivně interagovat s databází bez potřeby psaní velkého množství SQL kódu. V této aplikaci je napsán přesně jeden SQL dotaz.

Rozhraní automaticky poskytuje základní operace jako `save`, `findOne`, `findAll`, `delete` atd., což při vývoji ušetří čas, který by se jinak musel věnovat implementaci těchto běžných databázových operací. Dále rozhraní umožňuje jednoduché stránkování a řazení výsledků, což je nezbytné pro dobrý výkon a uživatelskou přívětivost ve webových aplikacích zobrazujících velké množství dat.

Jednou z nejvýznamnějších funkcí Spring Data JPA je možnost definovat query metody přímo v rozhraní repositáře. Tyto metody jsou definovány pouze názvem metody, který Spring Data interpretuje a automaticky generuje odpovídající SQL. Kromě query metod lze v `JpaRepository` definovat vlastní SQL nebo JPQL dotazy, což umožňuje flexibilitu při implementaci složitějších dotazů.

Celkově využití `JpaRepository` poskytuje robustní, flexibilní, bezpečný a efektivní způsob pro manipulaci s daty, což umožňuje soustředit se na implementaci byznys logiky, zatímco databázové operace a datová persistence jsou řešeny efektivně a s minimální námahou.

4.9.1 Základní JPA repositář

Jednoduché rozhraní, které neposkytuje žádné další metody nad rámec těch, které jsou poskytovány rozhraním `JpaRepository`, se dá naimplementovat na jednom řádku kódu. Díky takovému rozhraní může aplikace:

- Vytvářet nové záznamy (metoda `save`).
- Načítat záznamy podle jejich ID (metoda `findById`).
- Vypisovat všechny záznamy (metoda `findAll`).
- Mazat záznamy (metoda `delete`).

Pokud je ale potřeba specifická logika pro čtení dat, jako například v rozhraní `TagRepository`, je možné ji definovat pouze a jen názvem metody. Díky frameworku není třeba explicitně definovat implementaci SQL nebo JPQL dotazu. Framework automaticky interpretuje název metody a generuje odpovídající dotaz.

Obr. 4.13 Rozšíření základního JPA repositáře

```
public interface TagRepository extends JpaRepository<Tag, Long> {
    Optional<Tag> findByName(String name);
}
```

4.9.2 JPA repositář s využitím JPQL

Rozhraní `TodoRepository`, které dědí od `JpaRepository`, poskytuje logiku pro manipulaci s entitami `Todo` v databázi. Specifické pro toto rozhraní je definování vlastní query metody, která rozšiřuje základní funkcionalitu poskytovanou frameworkem.

Metoda `findByUserInvolved` je příkladem použití JPQL pro definici vlastního dotazu nad databází. Tato metoda slouží k získání seznamu úkolů, ve kterých je uživatel

Obr. 4.14 Definice JPA repositáře s vlastním JPQL dotazem

```
public interface TodoRepository extends JpaRepository<Todo, Long> {
    @Query("SELECT t FROM Todo t WHERE t.createdBy = :user OR t.
       assignee = :user OR :user MEMBER OF t.collaborators")
    List<Todo> findByUserInvolved(@Param("user") User user);
}
```

buď tvůrcem, přiřazenou osobou, nebo členem skupiny spolupracovníků. Použití anotace `@Query` umožňuje přímé zadání JPQL dotazu, což poskytuje velkou flexibilitu ve způsobu, jakým jsou data získávána.

Metoda používá anotaci `@Param("user")` pro mapování parametru metody `user` do JPQL dotazu. Tím je zajištěna správná substituce proměnné v dotazu a zároveň je tento přístup bezpečný proti útokům typu *SQL injection*, jelikož framework bezpečně zpracovává parametry dotazů.

5 Implementace frontendové aplikace

V předchozí kapitole jsem podrobně popsal implementaci serverové části aplikace, včetně souladu s principy Clean Architecture. Důležitým aspektem celého systému je však také frontendová část, jejímž úkolem je poskytnout uživatelům přívětivé, intuitivní a efektivní prostředí pro interakci se serverovou částí aplikace, což vychází z definovaných požadavků. Cílem této kapitoly je popsat, jak byla navržena a implementována.

V kontextu Clean Architecture hraje frontendová aplikace zásadní roli v prezentaci dat získaných z entit definovaných v doménové vrstvě a byznys logiky z use case vrstvy, které jsou zprostředkovány skrze RESTful controllery definované na backendu. Tímto způsobem frontend slouží jako spojnice mezi uživatelem a serverem, zajišťující, že interakce uživatele jsou efektivní, příjemné a bezpečné.

5.1 Architektura frontendové aplikace

Pro zrychlení a zpříjemnění vývoje frontendové aplikace jsem zvolil moderní framework Next.js, který slouží pro vývoj webových aplikací a využívá React. Je navržen tak, aby podporoval rendering na straně serveru (SSR), generování statických stránek (SSG), anebo kombinaci obojího (ISR). Next.js s sebou přináší několik vymožeností, které by jinak museli vývojáři v Reactu implementovat sami. Poskytuje například automatické rozdělení kódu, kdy na webového klienta jde pouze tolik javascriptového kódu, kolik je pro danou stránku potřeba. Dále také nabízí optimalizované načítání obrázků za pomoci vlastní komponenty `Image` a vestavěnou podporu pro API routy, což z tohoto framewroku dělá nástroj, ve kterém je možné vyvinout i serverovou část aplikace. Díky své modulárnosti a integraci s různými technologiemi a službami je vysoce oblíbený pro vývoj škálovatelných a výkonných webových projektů.

5.1.1 Next.js a App router

App Router v Next.js, zavedený ve verzi 13, přináší inovace do systému směrování tím, že podporuje sdílené rozložení (layouty), vnořené směrování, stavy načítání a zpracování chyb.[33] Tento router je umístěn v novém adresáři `app`, což umožňuje jeho postupnou adaptaci vedle tradičního přístupu s adresářem `pages`. Pokud tomu tak v projektu je, tato struktura dává přednost App Routeru před Pages Routerem, aby se předešlo konfliktům URL.

Výhody využití App routeru by se daly shrnout takto:

- **Sdílené layouty:** Umožňuje snadné sdílení designových prvků napříč různými stránkami.

- **Vnořený routing:** Podporuje složitější architekturu uživatelského rozhraní.
- **Stavy načítání a zpracování chyb:** Zlepšuje vývojářskou zkušenost tím, že umožňuje jednoduchou implementaci indikace probíhajícího načítání dat nebo zobrazování chybových hlášek.

Na druhou stranu s sebou tento přístup nese i nevýhody:

- **Složitost:** Může zvýšit náročnost a komplexnost aplikace.
- **Adopce:** Vyžaduje adaptaci pro ty, kteří jsou zvyklí na tradiční strukturu adresáře `pages`.

5.1.2 Struktura aplikace

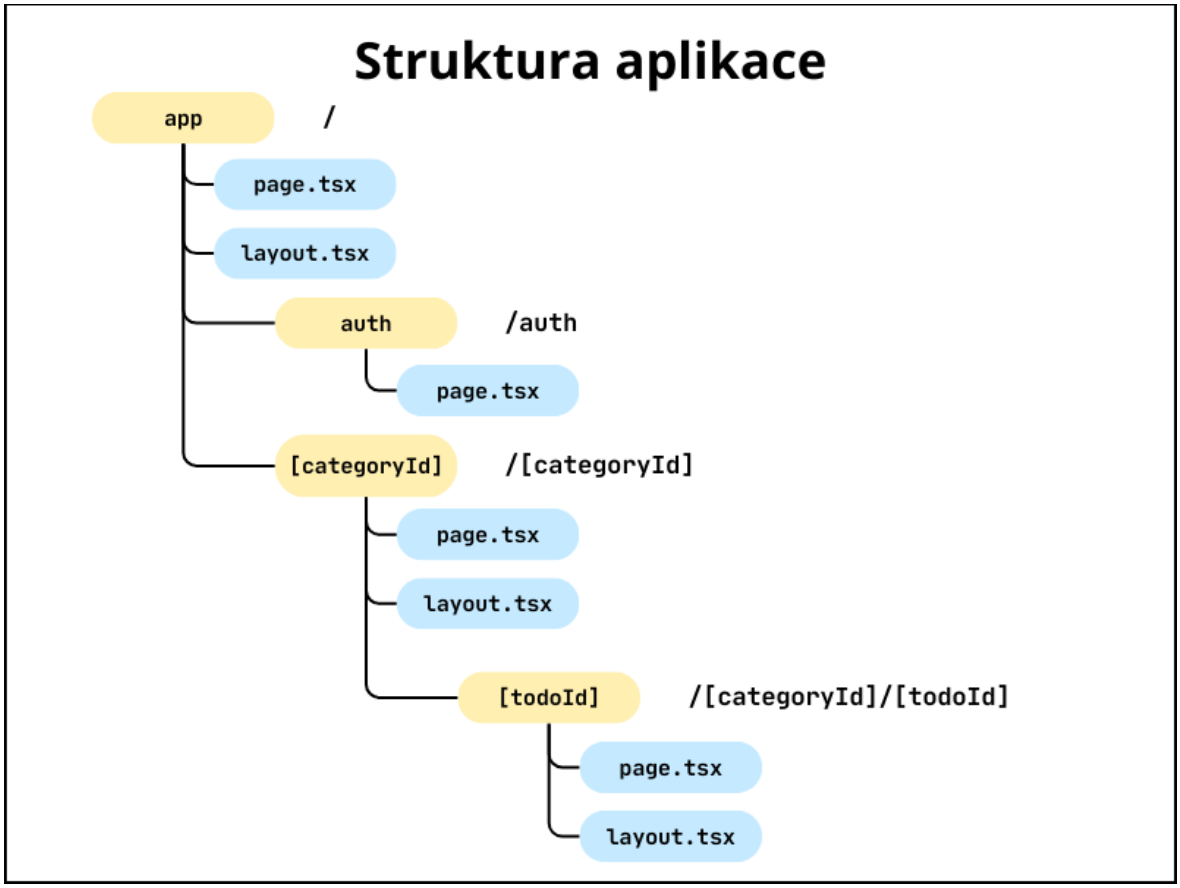
Použití App routeru v Next.js významně ovlivňuje strukturu projektů tím, že umožňuje složitější a flexibilnější organizaci komponent a stránek. V systému směrování, který App router přináší, lze vytvářet sdílené layouty a vnořené routy, což umožňuje lepší oddělení komponent podle jejich funkcí a závislostí. Toto uspořádání napomáhá k lepší modularitě a udržitelnosti kódu, protože komponenty, které sdílejí stejný layout nebo jsou logicky propojeny, mohou být organizovány pospolu. To vede k čistší a intuitivnější struktuře projektu, která podporuje efektivnější vývoj a snadnější údržbu.

Vnořené cesty a layouty Framework Next.js generuje cesty v aplikaci z adresářové struktury ve složce `app`, viz Obrázek 5.1, kde každá podsložka může představovat jednu cestu ve webové aplikaci.[34] Vnořené cesty umožňují, že child komponenty dědí layouty a stavy od svých rodičovských komponent, což zjednodušuje správu stavů a kontextů napříč aplikací. Tento přístup také podporuje lepší organizaci kódu a usnadňuje jeho údržbu, protože změny v designu nebo funkcionalitě na jednom místě mohou být snadno propagovány skrze celou strukturu vnořených cest.

Next.js umožňuje definovat globální layout přímo ve složce `app` a specifikovat další layouty pro jednotlivé sekce nebo stránky aplikace. Tyto layouty se automaticky aplikují na všechny podstránky nebo komponenty vnořené v dané struktuře adresářů, viz Obrázek 5.2. Uspořádání layoutů v aplikaci je znázorněno na následujícím obrázku.

5.2 Komponenty a implementační detaily

Komponenta v Reactu je základní stavební jednotka uživatelského rozhraní, která umožňuje izolaci a znovupoužití kódu.[35] Komponenty přijímají data formou `props` (properties), které mohou ovlivňovat chování a konfiguraci komponent. Komponenty



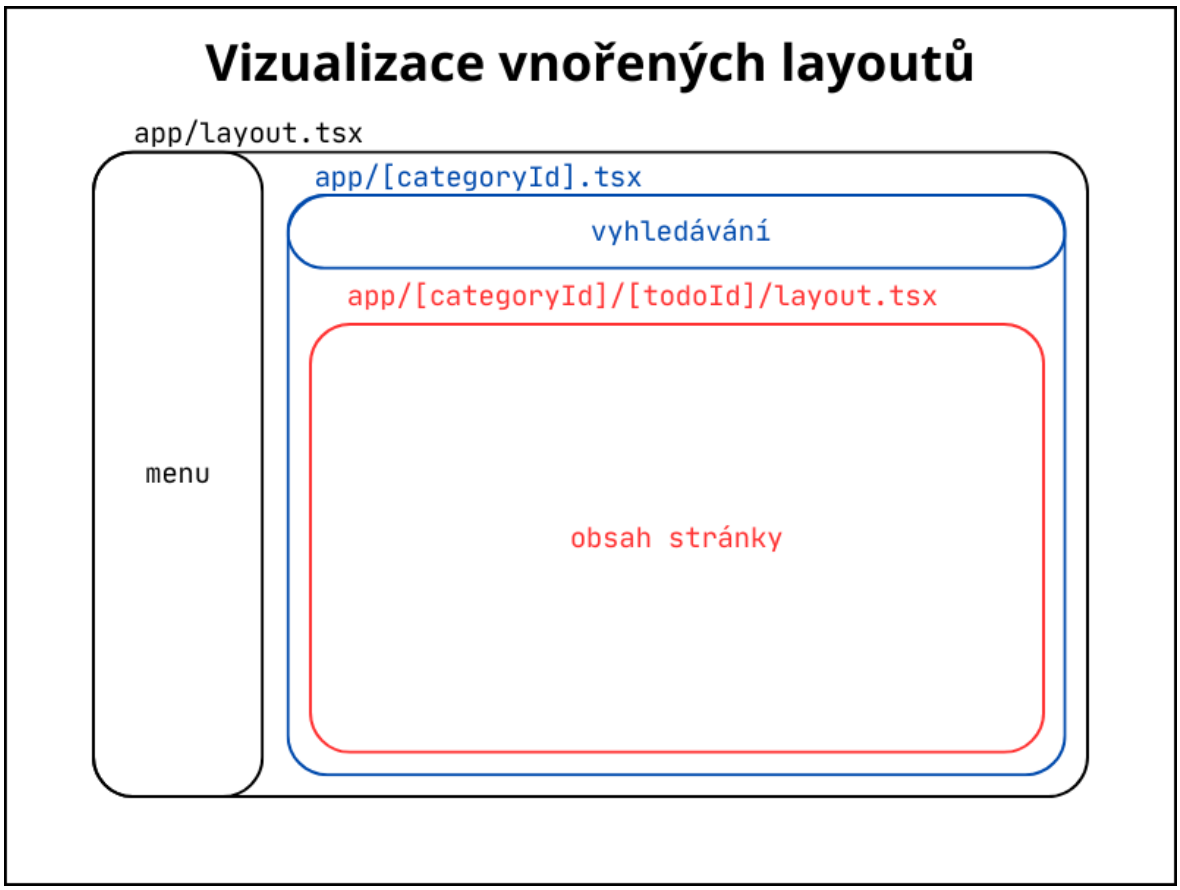
Obr. 5.1 Grafická reprezentace frontendové aplikace, autor: Patrik Procházka

vytvářejí hierarchii, kde mohou být data předávána z rodičovských komponent do potomků.

React definuje základní pravidla pro vytváření komponent, které je silně doporučené dodržovat:[35]

- **Single Responsibility Principle:** Každá komponenta by měla mít jednu zodpovědnost a být relativně malá.
- **Pure Components:** Komponenty by měly být "čisté", což znamená, že by neměly měnit vstupní data (props).
- **Znovupoužitelnost:** Komponenty by měly být navrženy s možností znovupoužití v různých částech aplikace.

Data putují mezi komponentami prostřednictvím `props`. Rodičovské komponenty mohou posílat data a funkce (logiku) do svých potomků, což jim umožňuje reagovat na změny a interakce uživatelů. Tento přístup podporuje jasné a udržitelné propojení komponent, které zajišťuje jejich snadnější správu.



Obr. 5.2 Grafická reprezentace vnořených layoutů, autor: Patrik Procházka

5.2.1 Ukázková definice komponenty

Jako příklad lze uvést komponentu `Input`, která je použita napříč aplikací.

Komponenta `Input`, viz Obrázek 5.3, má několik vstupních vlastností, některé z nich mají nastavené výchozí hodnoty. Protože je frontendová aplikace programovaná v TypeScriptu, tak existuje definované rozhraní `InputProps`, viz Obrázek 5.6, které definuje typy jednotlivých vlastností.

Každá z těchto poskytnutých vlastností má svůj vlastní význam. Vlastnosti jako `small`, `hasError` slouží ke stylování obsahu, zatímco vlastnosti jako `value`, `defaultValue`, `onChange`, `required` a `type` ovlivňují logiky a chování komponenty.

Následné použití komponenty v praxi vypadá tak, že komponentu v JSX části komponenty zmíníme jako specifickou HTML značku, která se jmenuje jako nadefinovaná komponenta, v tomto případě `<Input />`, a předáme jí vlastnosti, které bude potřebovat ke správnému fungování.

Obr. 5.3 Definice Input komponenty

```
const Input: FC<InputProps> = ({
    id,
    name,
    value,
    defaultValue,
    placeholder,
    required = false,
    onChange,
    hasError,
    small = false,
    type = "text"
}) => {
    return (
        <input
            type={type}
            id={id}
            name={name}
            defaultValue={defaultValue}
            value={value}
            onChange={onChange}
            className={`w-full ${small ? "px-1 py-1" : "p-2 md:px-4 md
                :py-3.5 h-full"} border rounded focus:outline-none
                focus:border-blue-500 ${hasError ? "border-red-700" :
                ""} text-xs md:text-base`}
            placeholder={placeholder}
            required={required}
        />
    )
}

export default Input;
```

Obr. 5.4 Definice rozhraní InputProps

```
interface InputProps {
    id?: string;
    name?: string;
    value?: string;
    defaultValue?: string;
    onChange?: (event: ChangeEvent<HTMLInputElement>) => void;
    placeholder: string;
    required?: boolean;
    hasError?: boolean;
    small?: boolean;
    type?: string;
}
```

5.3 Komunikace se serverem

Pro komunikaci se serverem využívá aplikace nativní javascriptovou funkci `fetch`. Pro standardizování všech volání v aplikaci je implementována funkce `fetchJson`, která

Obr. 5.5 Ukázka použití komponenty Input

```
const NewCategoryForm: FC<NewCategoryFormProps> = ({
    onSuccessfullyCreate}) => {

    // Logika komponenty NewCategoryFrom

    return (
        {/* Definice elementu v komponente */}
            <div className="w-4/5 mb-2">
                <Input
                    placeholder="Nazev kategorie"
                    id="categoryName"
                    name="categoryName"
                    small
                    hasError={!!error}
                />
            </div>
        {/* Definice dalsich elementu v komponente */}
    )
}
```

rozšiřuje standardní možnosti `fetch` o manipulaci s HTTP hlavičkami pro zahrnutí autorizačního tokenu. Tato funkce je základem pro dvě další metody: `fetchWithNoAuth` pro neautentizované požadavky a `fetchWithAuth pro požadavky` vyžadující autorizaci pomocí JWT. Přístup k tokenu je realizován přes `localStorage`, což umožňuje jednoduché a bezpečné spravování stavu autentizace uživatele. Tento způsob je vhodný pro vývojovou fázi aplikace, ale pro produkční nasazení by se místo `localStorage` měly využít cookies.

Komentář Celá frontendová aplikace se skládá z většího počtu komponent a není v zájmu této práce, aby byla každá jedna z nich popsána. Tato kapitola nastínila, jak bylo k celému vývoji frontendové aplikace přistoupeno, přičemž kompletní zdrojové kódy lze najít v Příloze 1.

Obr. 5.6 Definice funkcí fetchJson, fetchWithNoAuth, fetchWithAuth

```
async function fetchJson(url: string, options?: FetchOptions): Promise
    <any> {
     const headers = new Headers(options?.headers);
     if (options?.token) {
         headers.set('Authorization', `Bearer ${options.token}`);
     }

     const response = await fetch(`${API_URL}${url}`, { ...options,
        headers });

     if (!response.ok) {
         const errorInfo = await response.json();
         throw new Error(errorInfo.message || 'API request failed');
     }

     return response.json();
}

export function fetchWithNoAuth(url: string, options?: RequestInit):
    Promise<any> {
     return fetchJson(url, options);
}

export function fetchWithAuth(url: string, token: string, options?:
    RequestInit): Promise<any> {
     return fetchJson(url, { ...options, token });
}
```

ZÁVĚR

Jedním z cílů práce bylo prozkoumat, jak může Clean Architecture zlepšit proces vývoje softwaru a zvýšit kvalitu výsledného produktu.

První část práce poskytla teoretický základ, zdůraznila význam dobře navržené softwarové architektury a popisovala principy i vrstvy Clean Architecture. Diskuze o jejích výhodách a omezeních byla podpořena porovnáním s dalšími populárními architektonickými styly, což poskytlo hlubší porozumění její unikátní hodnotě v kontextu moderního softwarového vývoje.

V praktické části jsem prezentoval implementaci aplikace, která demonstruje výhody použití Clean Architecture ve skutečném vývojovém projektu. Důraz byl kladen na modulární design a flexibilitu při integraci různých technologií a nástrojů, což jsou klíčové vlastnosti, které Clean Architecture podporuje.

Během práce jsem narazil na několik omezení. Specificky, implementace Clean Architecture vyžaduje pečlivé plánování a detailní znalost jak návrhových vzorů, tak doménového modelu aplikace, což může být v počátečních fázích projektu časově náročné. V některých případech může tato předběžná náročnost zpomalit rychlost vývoje, zejména ve fázích, kdy je potřeba rychle prototypovat.

Výsledky této práce ukázaly, že Clean Architecture může významně přispět k efektivitě vývojových procesů, udržitelnosti softwaru a jeho schopnosti přizpůsobit se měnícím se požadavkům bez nutnosti rozsáhlých úprav stávajícího kódu (z dlouhodobého hlediska). Podařilo se mi navrhnout a implementovat aplikaci, která je nejen funkční a vyhovuje stanoveným požadavkům, ale také je dobře připravená na budoucí rozšíření a údržbu.

Na základě výsledků práce doporučuji další výzkum zaměřit na optimalizaci procesu implementace Clean Architecture, specificky v kontextu agilních metodik vývoje, kde rychlá iterace a flexibilita jsou klíčové.

Závěrem lze konstatovat, že Clean Architecture nabízí robustní řešení pro vývoj softwaru, které může pomoci lépe reagovat na dynamický vývoj trhu a technologické inovace. Tato práce představuje důležitý krok k mému profesnímu růstu a poskytuje cenné lekce, které mohu aplikovat v budoucích projektech.

SEZNAM POUŽITÉ LITERATURY

[1] Richards, M.; Ford, N.: *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Inc., 2020, ISBN 978-1-492-04345-4.

[2] Bass, L.; Clements, P.; Kazman, R.: *Software Architecture in Practice: An Engineering Approach*. Addison-Wesley Professional, 2012, ISBN 978-0-321-81573-6.

[3] Martin, R. C.: *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017, ISBN 978-0134494166.

[4] Buczyński, S.: *Implementing the Clean Architecture*. Self-published, 2020. URL `https://cleanarchitecture.io`

[5] Vernon, V.: *Implementing Domain-Driven Design*. Boston, MA: Addison-Wesley, 2003, ISBN 0-321-83457-7.

[6] Evans, E.: *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003, ISBN 0-321-12521-5.

[7] Rocha, H. F. O.: *Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices*. Apress, 2021, ISBN 978-1-4842-7468-2.

[8] Richards, M.: *Software Architecture Patterns*. O'Reilly Media, Inc., 2024.

[9] Vieira, D.: *Designing Hexagonal Architecture with Java - Second Edition*. Packt Publishing, 2023, ISBN 978-1-83763-511-5.

[10] Ciolli, G.; Mejías, B.; Angelakos, J.; aj.: *PostgreSQL 16 Administration Cookbook*. Birmingham, UK: Packt Publishing, 2023, ISBN 9781835460580.

[11] Heckler, M.: *Spring Boot: Up and Running: Building Cloud Native Java and Kotlin Applications*. O'Reilly Media, 2021, ISBN 9781492076988.

[12] Spilca, L.: *Spring Security in Action: Building Cloud Native Java and Kotlin Applications*. Manning Publications, 2021, ISBN 9781617297731.

[13] Spring Framework Documentation :: Spring Framework - Testing. 2024. URL `https://docs.spring.io/spring-framework/reference/testing.html`

[14] TypeScript: The Basics. `https://www.typescriptlang.org/docs/handbook/2/basic-types.html`, 2024, cit: 2024-02-27.

[15] TypeScript: Why does TypeScript exist? `https://www.typescriptlang.org/why-create-typescript`, 2024, cit: 2024-02-27.

[16] React: Thinking in React. `https://react.dev/learn/thinking-in-react`, 2024, cit: 2024-02-27.

[17] Google Trends. `https://trends.google.com/trends/explore?cat=31&date=2022-03-31%202024-03-31&q=%2Fm%2F012l1vxv,%2Fg%2F11c0vmgx5d,%2Fg%2F11c6w0ddw9`, cit: 2024-02-27.

[18] Robertson, S.; Robertson, J.: *Mastering the Requirements Process: Getting Requirements Right*. Boston, MA: Addison-Wesley Professional, 2012, ISBN 978-0321815743.

[19] Wiegers, K.; Beatty, J.: *Software Requirements (Developer Best Practices)*. Boston, MA: Microsoft Press, 2013, ISBN 978-0735679665.

[20] Adlin, T.; Pruitt, J.: *The Essential Persona Lifecycle: Your Guide to Building and Using Personas*. Amsterdam: BIS Publishers, 2010, ISBN 978-0-12-381418-0.

[21] Cohn, M.: *User Stories Applied: For Agile Software Development*. Boston, MA: Addison-Wesley Professional, 2004, ISBN 0-321-20568-5.

[22] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Prentice Hall, 1995, ISBN 9780201633610.

[23] Walls, C.: *Spring in Action, Sixth Edition*. Manning Publications, 2022, ISBN 9781617297571.

[24] Vitale, T.: *Cloud Native Spring in Action: With Spring Boot and Kubernetes*. Manning Publications, 2022, ISBN 9781617298424.

[25] Turnbull, J.: *The Docker Book*. Turnbull Press, 2014, ISBN 978-0-98-882020-3.
URL `https://www.oreilly.com/library/view/the-docker-book/9780988820203/`

[26] IBM: IBM Topics: REST APIs. `https://www.ibm.com/topics/rest-apis`, 2024, cit: 2024-05-10.

[27] Menezes, A.; van Oorschot, P.; Vanstone, S.: *Handbook of Applied Cryptography*. CRC Press, 1996, ISBN 978-0-84-938523-0.
URL `http://www.cacr.math.uwaterloo.ca/hac/`

[28] Martin, R. C.: Design Principles and Design Patterns. https://wnmurphy.com/assets/pdf/Robert_C._Martin_-_2000_-_Principles_and_Patterns.pdf, 2000, cit: 2024-05-11.

[29] Martin, R. C.: Getting a SOLID start. https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start, cit: 2024-05-11.

[30] Martin, R. C.: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008, ISBN 978-0132350884.

[31] Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series, Boston, MA: Addison-Wesley, 2003, ISBN 0-321-12742-0.

[32] Microsoft: Web API design best practices. https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design, cit: 2024-05-12.

[33] Building Your Application: Routing | Next.js. https://nextjs.org/docs/app/building-your-application/routing, cit: 2024-05-12.

[34] Routing: Defining Routes | Next.js. https://nextjs.org/docs/app/building-your-application/routing/defining-routes, cit: 2024-05-12.

[35] React: Your First Component. https://react.dev/learn/your-first-component, 2024, cit: 2024-05-12.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface
CRUD	Create, Read, Update, Delete operace
DDD	Domain-Driven Design (architektonický přístup)
DTO	Data Transfer Object
EDA	Event-Driven Architecture (architektonický přístup)
HMAC	Hash-Based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IoC	Inversion of Control (základní princip Spring Boot frameworku)
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVCC	Multi-Version Concurrency Control
NPM	Node Package Manager
ORM	Object-Relational Mapping
REST	Representational state transfer
SPA	Single Page Application
URL	Uniform Resource Locator

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

PŘÍLOHA P I. CD S ARCHIVEM OBSAHUJÍCÍ ZDROJOVÉ KÓDY APLIKACE

Příloha obsahuje zdrojové kódy, konfigurační soubory a instrukce ke spuštění aplikace.