FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Aplikace úvazkostroj - refaktoring a rozvoj
Student:	Vladimir Kulikov
Vedoucí:	Ing. Michal Valenta, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Cílem práce je rozvoj aplikace Úvazkostroj, jejíž novou verzi ve své diplomové práci navrhl a implementoval student Nguyen, Trong Chung Chau. Postupujte dle bodů uvedených níže:

1. Seznamte se se stávající verzí backend a frontend.
2. Dokončete úlohy (issues), které drobně doplňují funkcionalitu aplikace, a které nebyly dotaženy v předchozí práci.
3. Navrhněte a realizujte testování aplikace. Konkrétně: automatické testy pro provolání API backend, unit testy na autorizaci a uživatelské testování pro frontend.
4. Analyzujte, navrhněte a realizujte CI/CD infrastrukturu pro develop, stage a production prostředí.
5. Analyzujte přínosy upgrade ze stávající verze NextJS 12 na aktuální verzi 13. Budete-li čas, pak na pokyn vedoucího práce upgrade proveďte.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 30. října 2023 v Praze.

Bakalářská práce

APLIKACE ÚVAZKOSTROJ – REFAKTORING A ROZVOJ

Vladimir Kulikov

Fakulta informačních technologií
Katedra softwarové inženýrství
Vedoucí: Ing. Michal Valenta, Ph.D.
7. května 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych vyjádřit poděkování Ing. Michalu Valentovi, Ph.D., za vedení této práce a Ing. Trongovi Chung Chau Nguyen za pravidelné konzultace a cenné rady ohledně této práce. Děkuji své přítelkyni za její pomoc a rady při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel licenční smlouvu o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ust. § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů.

V Praze dne 7. května 2024

Abstrakt

Tato práce se zabývá implementací současného informačního systému pro hodnocení pedagogického výkonu, Úvazkostroj verze 4.0. Tento systém je v této práci podroben revizi s cílem identifikovat a opravit existující chyby a nedostatky, přidat chybějící funkcionality a modernizovat celkovou architekturu a technologie použité ve vývoji.

Práce systematicky prochází procesem od prvotního seznamení s aplikací, přes výběr vhodné metodiky vývoje softwaru, technologickou analýzu, návrh zlepšení, testování až po nasazení a přípravu CI/CD infrastruktury. Závěr této práce je věnován analýze upgradu frontendového frameworku Next.js.

Hlavním výstupem této práce je systém Úvazkostroj. Byly implementovány chybějící funkcionality, opraveny nedostatky předchozího řešení a provedena řada vylepšení z hlediska optimalizace, bezpečnosti a uživatelského rozhraní. Systém je nyní zcela připraven k nasazení do produkčního prostředí.

Klíčová slova Úvazkostroj, informační systém, frontend, autentizace, docker, Typescript, Postman, Next.js

Abstract

This thesis focuses on the implementation of the current information system for evaluating educational performance, Úvazkostroj version 4.0. This system is reviewed in this work with the goal of identifying and correcting existing errors and deficiencies, adding missing functionalities, and modernizing the overall architecture and technologies used in development.

The work systematically goes through the process from initial acquaintance with the application, through the selection of an appropriate software development methodology, technological analysis, design improvements, testing, to deployment and preparation of CI/CD infrastructure. The conclusion of this thesis is dedicated to the analysis of upgrade of the frontend framework Next.js.

The main output of this work is the system Úvazkostroj. Missing functionalities have been implemented, drawbacks of the previous solution have been corrected, and a number of improvements have been made in terms of optimization, security, and user interface. The system is now fully prepared for deployment in a production environment.

Keywords Úvazkostroj, information system, frontend, authentication, docker, Typescript, Postman, Next.js

Seznam zkratek

API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
CSR	Client-Side Rendering
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
REST	Representational State Transfer
SEO	Search Engine Optimization
SSG	Static Site Generation
SSR	Server-Side Rendering
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience

Úvod

Cílem této práce je implementace aplikace Úvazkostroj verze 4.0 – současného systému pro správu a hodnocení pedagogického výkonu, který byl původně navržen a implementován v diplomové práci studenta Nguyen, Trong Chung Chau na FIT ČVUT.

V první části této práce bude seznamení s aplikací — popíše se architektura systému, použité technologie, chyby a nedostatky, stejně tak chybějící funkcionality.

Ve druhé části této práce bude základní přehled metodik vývoje softwaru a volba metodiky, která se bude používat během této práce.

Třetí část této práce bude věnována technologické analýze nedostatků aplikace a návrhu jejich zlepšení. Nebude zde chybět ani teorie testování, analýza a výběr typu testů a testovacích nástrojů pro jednotlivé části systému. A na závěr této části bude provedena analýza a návrh metody nasazení a CI/CD infrastruktury.

V další části této práce budou rozebrány důležité části implementace systému, která bude přímo navazovat na analýzu a teorii z analytické části. Bude zde ukázána realizace API testů a provedení uživatelského testování.

V poslední části této práce bude provedena analýza upgradu frameworku Next.js.

Kapitola 1

Aplikace Úvazkostroj verze 4.0

Cílem této kapitoly je základní seznámení s aplikací, její architekturou a technologiemi. Dalším cílem je identifikovat chyby a nedostatky současného řešení a chybějící funkcionality.

1.1 O aplikaci Úvazkostroj

Aplikace *Úvazkostroj*[1] verze 4.0 je současný systém[2] pro hodnocení pedagogického výkonu, který původně vznikl pro *úvazkáře*[3]. fakulty.

Úvazkostroj eviduje a spravuje vyučující a předměty v rámci ČVUT FIT. Systém na základě přiřazení předmětů a dalších parametrů vypočítává, podle známé metodiky, míru plnění pedagogického výkonu. Mimo jiné poskytuje informace ohledně pokrytí paralelek jednotlivých předmětů a vypočítává náročnost předmětů a jejich cenu.

Pro kompletní historii vývoje systému se odkazuje na diplomovou práci [1] Ing. Nguyen Trong Chung Chau.

1.2 Architektura a technologie

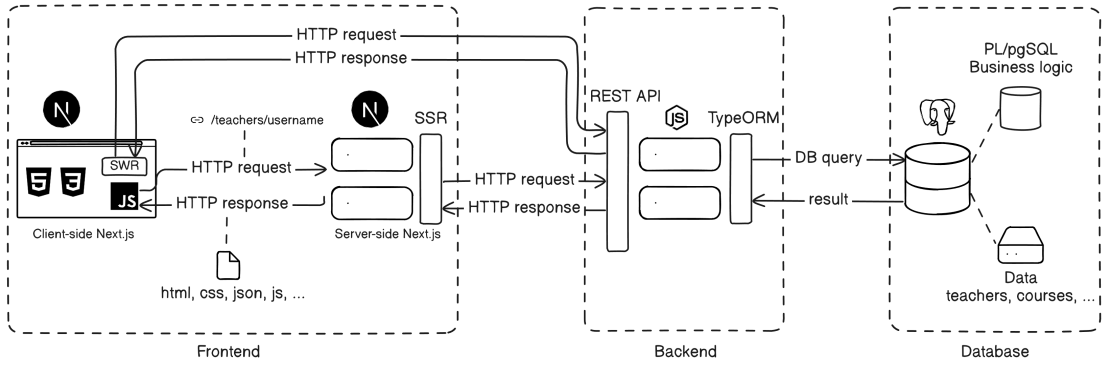
V této části bude popsána architektura současného systému spolu s použitými technologiemi v jednotlivých subsystémech. Detailnější dokumentace k architektuře, dále analýza volby jednotlivých technologií a detaily k implementaci jsou zdokumentovány v diplomové práci [1] Ing. Nguyen Trong Chung Chau.

Současný systém se v podstatě skládá z frontendu, backendu a databáze. Níže je uveden podrobnější popis každého z nich, a obrázek 1.1 zachycuje high-level pohled na celkovou architekturu systému, která se v průběhu této práce nezmění.

1.2.1 Frontend

Frontend je napsán v jazyce *JavaScript* a je postaven na moderním frameworku *Next.js* verze 12. Uživatelské rozhraní systému je realizováno pomocí komponentové knihovny *Material-UI*. Proces autentizace a autorizace je řízen hlavně na klientské straně, tedy v prohlížeči uživatele. Frontend komunikuje s backendem přes REST API a pro načítání dat na klientské straně využívá

[1] podle vedoucího této práce Ing. Michala Valenty, PhD.
[2] v rámci této práce bude termín *systém* používán jako synonymum pro *aplikaci*
[3] speciální pozice, jejíž povinností je hlídat plnění úvazků učitelů dané katedry



Obrázek 1.1 Architektura systému Úvazkostroj

technologii a nástroj *SWR*. Pro správu globálního stavu, jakožto například vybraný kód semestru, se používá knihovna *Redux*.

1.2.2 Backend

Backend je také napsán v jazyce *JavaScript* a běží na platformě *Node.js*. Backend má vystavené REST API, což znamená, že komunikuje s frontendem a dalšími službami pomocí HTTP požadavků a poskytuje data ve formátu JSON. Samotné REST API je implementováno pomocí frameworku *Koa.js*. Navíc na něm byly napsány různé *middleware*, například pro autentizaci, validaci požadavků a zpracování chyb. Pro komunikaci s databází používá ORM (Object-Relational Mapping) knihovnu *TypeORM*.

1.2.3 Databáze

Databáze je relační DBMS *PostgreSQL*. V této databázi jsou uložena jak data, tak i obchodní logika. Příkladem obchodní logiky jsou funkce v procedurálním jazyce *PL/pgSQL* pro výpočet plnění úvazků a mnoho užitečných pohledů na data.

1.3 Chyby a nedostatky

V systému se vyskytují chyby především v uživatelském rozhraní, avšak existují také nedostatky v implementaci frontendové části. Tyto nedostatky nejsou spojeny s globální architekturou, ale spíše s drobnými lokálními nedokonalostmi.

1.3.1 Chyby uživatelského rozhraní

Uživatelské rozhraní systému je z hlediska UI a UX dobře navrženo. Je intuitivní pro uživatele, a rozvržení je logicky strukturováno. Nicméně obsahuje několik méně závažných chyb, které obvykle nemají významný vliv na základní funkcionalitu.

Jak je vidět na obrázku 1.2, při zapnutí editačního módu na některých formulářích dochází k nepředvídatelnému posunu textových polí od jejich původního umístění v čtecím módu. Jedná se v zásadě o *layout shift*.

Podobný problém s posunem rozvržení se vyskytuje pokud uživatel zadá neplatné údaje do textového pole. Následně se pod polem zobrazí chybová hláška, která posune ostatní prvky dolů,

což vede k zmatení uživatele v celém formuláři a z pohledu UI a UX působí neesteticky, viz obrázek 1.3.

V mnoha tabulkách nejsou šířky sloupců dobře nastaveny. Například, šířka sloupce pro uživatelské jméno by měla být pevně stanovena, a šířka sloupce pro jméno by měla zabírat co nejvíce dostupného prostoru. Další problém vzniká při načítání následující stránky v tabulkách. Na následující stránce se šířky sloupců změní, což vede k nepříjemnému efektu posunování layoutu, o čemž bylo již zmíněno výše. Šířky sloupců by měly být konzistentní napříč všemi stránkami v tabulkách.

Při načítání stránky je prvním, co uživatel vidí, text „loading“ v levém horním rohu obrazovky, a až poté se stránka vykreslí. Z pohledu UX by bylo ideální, kdyby uživatel viděl alespoň část stránky okamžitě, i když nejsou všechna data načtena.

Podobný problém nastává, když se načítají data z backendu a v místech, kde se data očekávají, nic není. To může uživatele zmást, protože nemá jasno, zda je místo prázdné, nebo se data teprve načítají. Ideálně by, pokud se data načítají, měl být zobrazen nějaký indikátor, například React komponenta `Skeleton`, nebo textová zpráva informující uživatele, že data jsou v procesu načítání.

Pokud uživatel není přiřazen k žádné roli, ale je autentizován, zobrazí se mu chybová stránka 403 - Forbidden. Z hlediska UX by bylo vhodné poskytnout takovému uživateli informace, že nemá přiřazenou žádnou roli, a nabídnout mu kontaktování administrátora.

Dalším nedostatkem je nemožnost odeslat URL odkaz na konkrétní semestr, což komplikuje sdílení informací o konkrétních vyučujících v konkrétním semestru. Stejně tak nelze jazyk aplikace nastavit přímo v URL, což omezuje uživatelskou přívětivost při sdílení.

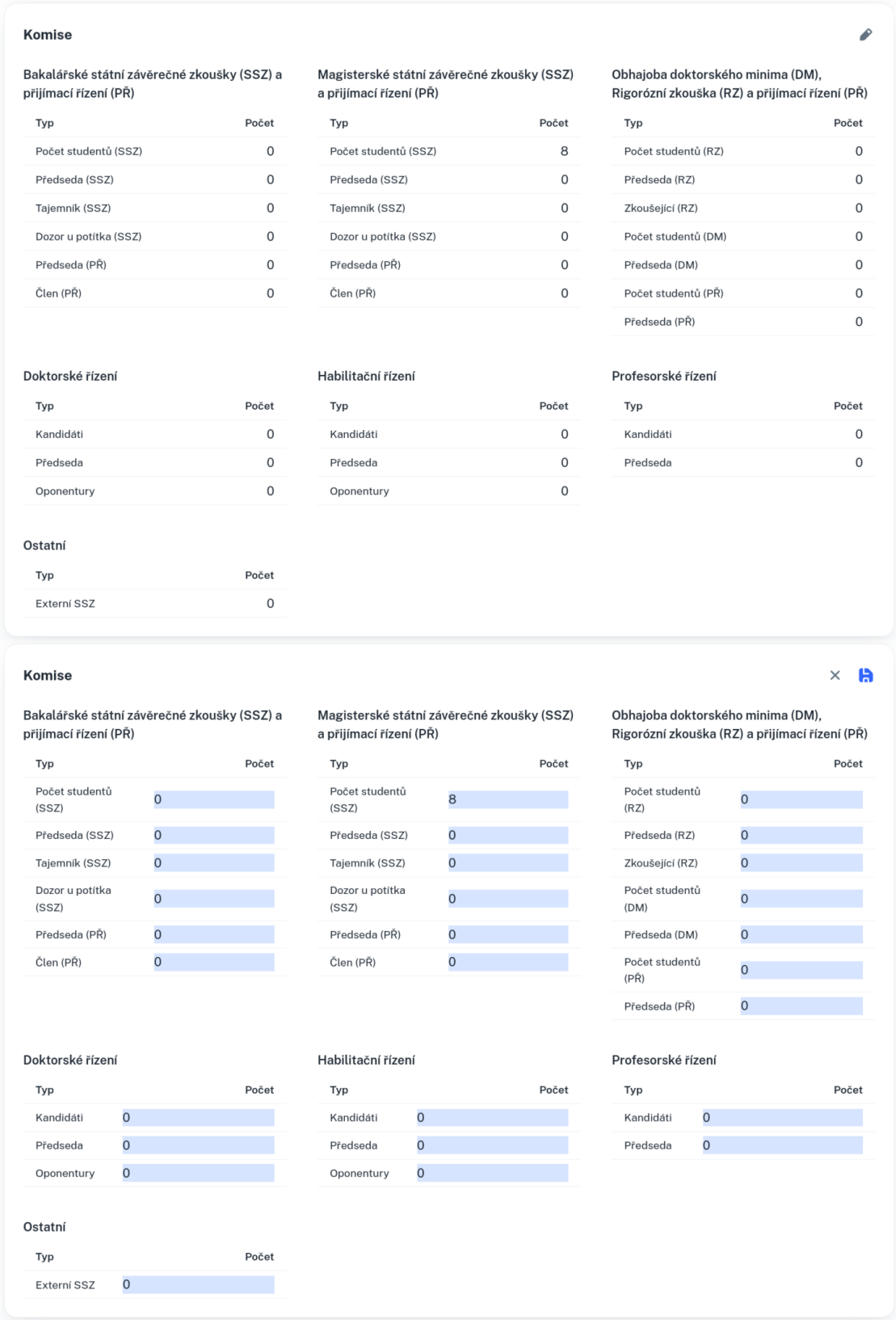
1.3.2 Chyby v implementaci

Přestože je frontend formálně napsán v TypeScript, nastavení TypeScriptu není striktní, což v praxi znamená, že kód byl napsán téměř jako čistý JavaScript bez využití striktního typování. V důsledku toho se v mnoha komponentách objevují typické typové chyby, které by byly jinak detekovatelné již při kompilaci. Často byly referencovány neexistující (undefined) atributy objektů, nebo byl předán jiný typ, než jaký funkce očekávala.

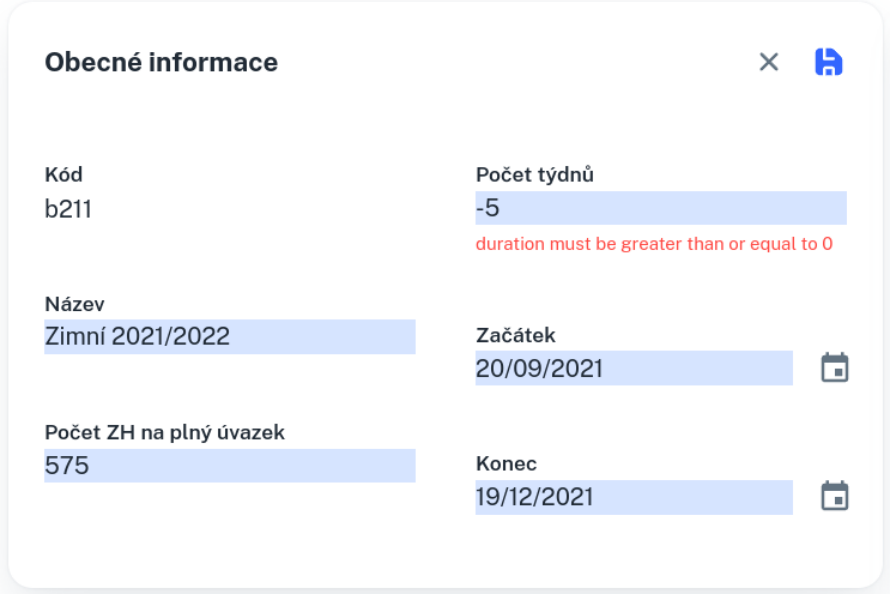
V některých částech systému chybí základní optimalizace. To se projevuje například na stránce učitele, kde se komponenta s grafem může vykreslovat víckrát, než je to ve skutečnosti potřeba.

1.4 Chybějící funkcionality

Chybějící funkcionality jsou převážně v uživatelském rozhraní. Tabulky postrádají možnost filtrování podle různých parametrů, stejně jako možnost řazení sloupců. Na stránce věnované semestru je zobrazena pouze malá část všech dostupných parametrů semestru. V backendové části chybí endpointy pro aktualizaci parametrů semestru a pro vytvoření nového semestru.



Obrázek 1.2 Ukázka současného uživatelského rozhraní: Formulář v čtecím módu nahoře a v editačním módu dole



Obrázek 1.3 Ukázka současného uživatelského rozhraní: Formulář s chybovou hláškou

Kapitola 2
Metodika vývoje

Cílem této kapitoly je popsat zakladní běžně používané metodiky vývoje software a v zavěru zvolit metodiku pro vývoj v rámci této práce.

2.1 O metodikách vývoje v softwarovém inženýrství

Metodika vývoje softwaru je sada doporučených postupů, technik a principů vedoucích k vytvoření požadovaného softwaru. Metodika specifikuje model životního cyklu softwaru, který popisuje jednotlivé fáze, jimiž aplikace prochází od počátečního nápadu až po její ukončení nebo vyřazení.

2.2 Tradiční metodiky

Tradiční metodiky představují přístupy k vývoji softwaru, které jsou strukturované a lineární. Často jsou velmi propracované, podrobné a formální. Kladou velký důraz na tvorbu dokumentace. Tyto metodiky jsou založené na předpokladu, že vývoj softwaru lze dopředu přesně popsat a naplánovat. Proto obvykle požadují dokončení jedné fáze projektu předtím, než se může začít s další.

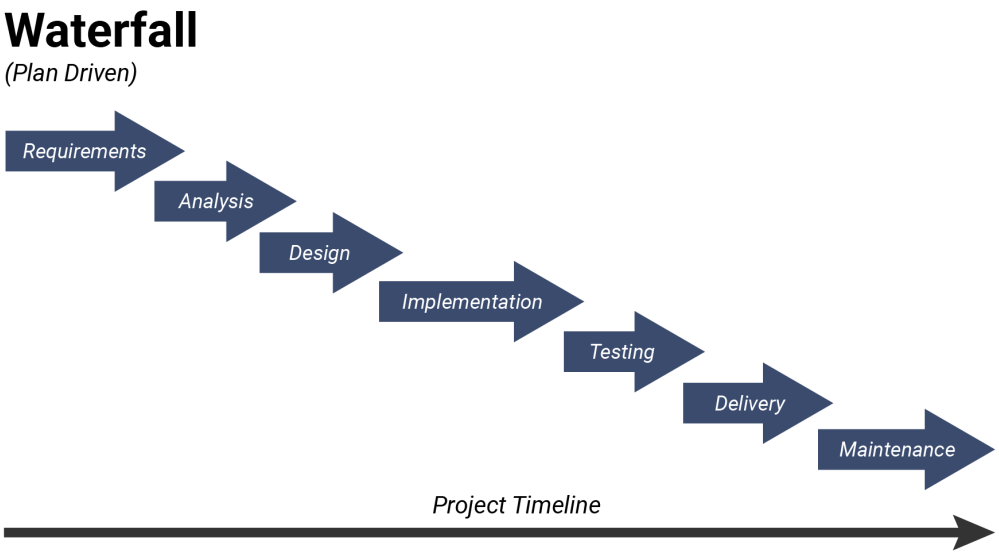
Tradiční metodiky mají model životního cyklu aplikace, který je nejčastěji založen na vodopádovém modelu[1]. Vodopádový model [2] dělí vývoj softwaru do postupně prováděných fází, a to: analýza a sběr požadavků, návrh architektury a design, implementace a testování, dodání a údržba. Viz obrázek 2.1. Výhodou tohoto modelu je, že přináší velmi dobré výsledky v případě, že jsou požadavky na software dopředu známé a neměnné. Logicky vyplývá hlavní nevýhoda tohoto modelu, a to nemožnost vracet se ve fázích zpět (zpravidla lze jen o jeden krok). To znamená, že je nutné chápat, co má software plnit již na začátku.

Z toho vyplývá, že tradiční metodiky jsou vhodné pro projekty s jasně definovanými a neměnnými požadavky, avšak v praxi se často upřednostňují agilní metodiky pro jejich flexibilitu a schopnost rychle reagovat na změny.

2.3 Agilní metodiky

Agilní metodiky jsou v jistém smyslu protikladem tradičním metodikám. Jsou založeny na myšlence, že vývoj softwaru je dynamický proces, a v mnoha případech ho nelze přesně dopředu

[1] Vodopádový model získal svůj název kvůli své sekvenční a lineární struktuře, která připomíná tok vody přes sérii vodopádů.



Obrázek 2.1 Vodopádový model [3]

popsat a je tedy nezbytné se pružně přizpůsobovat změnám, které se v průběhu vývoje objeví.

Základní principy agilních metodik byly popsány v Agilním manifestu [4] z roku 2001 a jedná se v podstatě o následující čtyři hlavní principy:

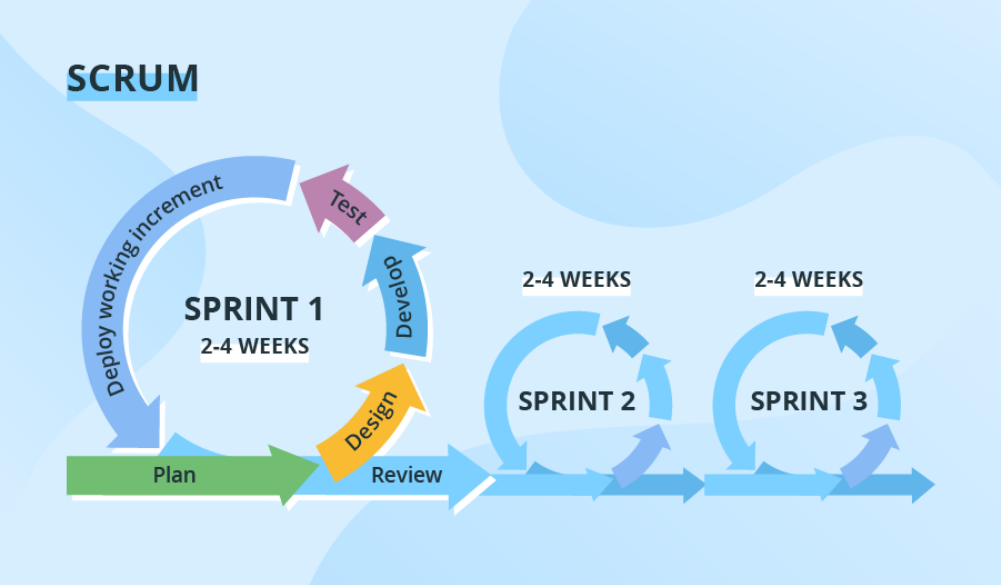
- Jednotlivci a interakce před procesy a nástroji.
- Spolupráce se zákazníkem před vyjednáváním o smlouvě.
- Reagování na změny před dodržováním plánu.
- Funkční software před vyčerpávající dokumentací.

Z toho plyne hlavní výhoda agilních metodik oproti tradičním metodikám. Touto výhodou je schopnost rychleji reagovat na potřeby zákazníka a tedy rychleji dodat použitelný software. To je možné díky použití iterativního a inkrementálního vývoje, kdy je práce rozdělena do úseků, v rámci kterých je software rozšiřován o další funkcionality nebo je modifikována funkcionalita stávající. Software je tedy zákazníkovi dodáván postupně.

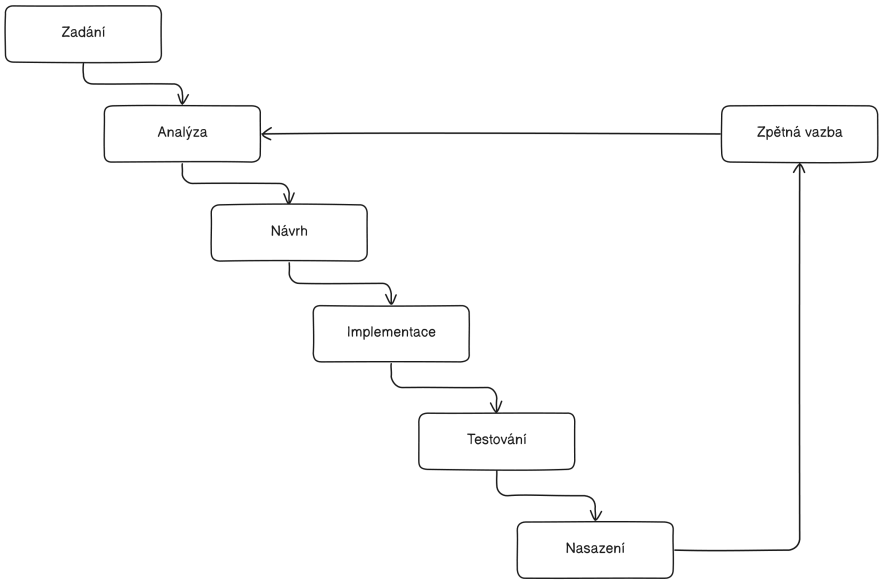
Typickým příkladem agilní metodiky je metodika Scrum [5], viz obrázek 2.2. Scrum rozděluje vývojový proces na krátké iterace nazývané sprinty, obvykle trvající 2-4 týdny, během nichž tým pracuje na prioritizovaném seznamu úkolů z backlogu. Na konci každého sprintu dochází k zhodnocení výstupu a může dojít i ke změnám požadavků zákazníka.

2.4 Shrnutí a volba metodiky

Jelikož specifikace (zadání bakalářské práce) a všechny požadované funkcionality systému jsou dopředu v této práci známé a měnit se nebudou, bude metodika založená na vodopádovém modelu coby nejvhodnější volba. V rámci této práce byl po dohodě s jejím vedoucím, Ing. Michalem Valentou, Ph.D., použit model iterativního vývoje. Tento model je zachycen na obrázku 2.3.



Obrázek 2.2 Scrum model [6]



Obrázek 2.3 Iterativní model používaný během této práce

Kapitola 3
Analýza

Cílem této kapitoly je analýza současného řešení hlavně frontendové části a návrh případného zlepšení. Dalším cílem je analýza a výběr typu testů a testovacích nástrojů pro jednotlivé části systému. Posledním cílem je analýza způsobu nasazení a navrch CI/CD infrastruktury.

3.1 Statické typování

Tato sekce se zabývá analýzou striktního statického typování, především pro frontend.

Je potřeba říci, že část frontendu je napsána v čistém JavaScriptu, další část je formálně sice napsána v TypeScriptu, ale nastavení typování není striktní. Toto způsobí, že se kód skoro neliší od čistého JavaScriptu, a tím přichází o většinu výhod statického typování, o čemž bude napsáno dále.

3.1.1 Kritika dynamického typování

Dynamické typování jazyka JavaScript je vlastnost, která umožňuje proměnným přijímat a měnit typy hodnot za běhu programu. Tato flexibilita může být pro některé vývojáře výhodou, protože umožňuje rychlejší a jednodušší vývoj aplikací. Nicméně současně přináší významné nevýhody a rizika.

Mezi hlavní nevýhody dynamického typování patří:

- **Náchylnost k chybám za běhu** – chyby typování se často objevují až za běhu aplikace, což může vést k neočekávanému chování nebo pádům programu.
- **Složitější údržba kódu** – bez explicitních deklarací typů je obtížnější pochopit účel nebo očekávané použití proměnných a funkcí, což komplikuje údržbu a rozvoj kódu.

3.1.2 O statickém typování

Statické typování znamená, že typy proměnných, parametrů, návratových hodnot funkcí a dalších struktur jsou pevně stanoveny již při kompilaci programu. Tento přístup umožňuje odhalit mnoho potenciálních chyb a problémů ještě před spuštěním programu.

Mezi hlavní výhody statického typování patří:

- **Chyby lze zachytit velmi brzy** – dokud nejsou chyby, zachycené při kompilaci, opraveny, program nemůže být zkompilován. To přináší významnou výhodu statického typování,

protože triviální chyby mohou být odhaleny velmi brzy, a to již během kompilace nebo před ní. To napomáhá vývoji stabilnějšího a lepšího programu.

- **Zlepšení udržitelnosti a čitelnosti kódu** – statické typování může zlepšit udržitelnost a čitelnost kódu tím, že usnadňuje jeho pochopení a úpravy. Typové anotace poskytují dodatečné informace o očekávaných vstupech a výstupech funkcí nebo metod, což usnadňuje ostatním vývojářům pochopení fungování kódu a identifikaci bezpečných možností pro jeho modifikaci. Kromě toho statické typování podporuje vývojáře ve psaní kódu, který se do jisté míry sám dokumentuje, což může snížit potřebu dodatečné dokumentace a usnadnit údržbu kódu v čase.
- **Lepší podpora nástrojů** – statické typování zlepšuje předvídatelnost kódu, což má přímý dopad na efektivitu a užitečnost nástrojů a integrovaných vývojových prostředí (IDE). Tyto nástroje mohou díky statickému typování poskytovat pokročilé funkce, jako jsou přesnější automatické doplňování kódu, lepší navigace v kódu, refaktoring a statická analýza.

Přestože statické typování přináší řadu výhod v oblasti bezpečnosti a přehlednosti kódu, může rovněž představovat určitá omezení a komplikovat vývojový proces.

Mezi nevýhody statického typování patří:

- **Složité chybové hlášky** – statické typovací systémy, zejména v jazycích s pokročilými typovými systémy, mohou produkovat velmi složité chybové hlášky, které mohou být obtížně srozumitelné.
- **Těžkosti s vyjádřením komplexních typů** – v některých případech může být obtížné vyjádřit komplexní typy pomocí statického typování. Například heterogenní kolekce, které mohou obsahovat mnoho různých datových typů. To může vést k potřebě kompromisů nebo použití návrhových vzorů, které mohou kód zkomplikovat.

3.1.3 Shrnutí

I když dynamické typování může v některých situacích zjednodušit vývoj, pro větší nebo komplexnější projekty může představovat významná rizika. V rámci systému zavádění striktní statické typové kontroly se určitě vyplatí, jelikož umožní odhalit mnoho chyb již ve fázi kompilace a poskytne dobře zdokumentovaný kód, což zlepší celkovou udržitelnost systému.

Pro zajištění statické typové kontroly se i nadále bude používat TypeScript. Je již součástí vývojového procesu, zaujímá vedoucí postavení v oblasti statického typování a v současné době neexistuje jiná alternativa, která by mu mohla konkurovat. Pro maximální efektivitu a odhalení potenciálních chyb je však klíčové používat TypeScript ve striktním režimu.

3.2 Redux

Tato sekce se zabývá analýzou knihovny *Redux* a zhodnocením možnosti jejího kompletního vyřazení ze systému.

Frontend využívá knihovnu Redux pro správu globálního stavu. Cílem Reduxu je poskytnout konzistentní prostředí pro správu stavu v rámci celé aplikace. Podrobné představení této knihovny spolu s patternem, který implementuje, bylo již uvedeno v diplomové práci [1] Ing. Nguyen Trong Chung Chau. Tato část se zabývá analýzou Reduxu a přínosem jeho odstranění z frontendu.

3.2.1 Kritika a alternativní přístupy

I když Redux přináší výhody ve správě složitých stavů ve velkých aplikacích, pro menší a středně velké projekty může být jeho využití zbytečně komplikované a může způsobovat nadměrné

zvyšování složitosti kódu. V současné verzi aplikace slouží Redux jako mechanismus pro konzistentní předávání dat mezi komponentami systému a udržuje stav učitelů, semestrů a předmětů, které jsou načteny z backendu. Také udržuje stav aktuálně vybraného kódu semestru a vybrané role v uživatelském rozhraní.

Použití knihovny *SWR* [7] pro načítání dat z backendu, která udržuje globální stav – cache s vrácenými daty po celé aplikaci, zpochybňuje nutnost Reduxu pro tento účel. Komponenty mohou načítat data přímo pomocí React hooku *useSWR*.

Pro správu globálního stavu v uživatelském rozhraní, konkrétně vybraného kódu semestru a uživatelské role, nebyla nalezena přímá existující alternativa. Nicméně existují různé metody správy tohoto globálního stavu, které se liší zejména podle toho, zda je stav sdílen se serverem, a podle úrovně bezpečnosti. Bezpečnost v kontextu vybraných prvků uživatelského rozhraní není prioritní, jelikož se nejedná o citlivé informace, ale pouze o stav uživatelského rozhraní. Pro přehled různých způsobů ukládání dat je uvedena následující tabulka:

Způsob uložení	Data viditelná serverové části?
Uložení dat v lokálním úložišti prohlížeče	Ne
Uložení dat v cookies	Ano
Použití React hooku *useContext*	Ne
Uložení v URL	Ano

Pro ukládání vybraného kódu semestru se ukazuje být nejvhodnější metoda ukládání v URL, což uživatelům umožňuje sdílení odkazů s konkrétním semestrem. Současná verze systému tuto možnost nenabízí, což může způsobovat uživatelům nepohodlí, jelikož musí explicitně uvádět, na který semestr se má druhá strana přepnout.

Vybranou roli uživatele je vhodné ukládat do cookies, jelikož je potřeba informace sdílet se serverem, což je důležité zejména pro procesy autentizace. Ukládání role uživatele v URL by nebylo vhodné z hlediska sdílení s ostatními uživateli, jelikož by mohlo vést k nežádoucímu zveřejnění informací o roli.

3.2.2 Shrnutí

V této části bylo analyzováno použití knihovny Redux, její nedostatky a přínosy po jejím odstranění z aplikace. Zjistilo se, že Redux je většinou redundantní a jeho odstraněním lze snížit komplexnost aplikace. Načítání dat je efektivně řešeno pomocí hooku useSWR, vybraný kód semestru může být uložen v URL a vybraná role uživatele v cookies, což přináší nové užitečné funkce a zjednodušuje strukturu aplikace.

3.3 Metody renderování webových stránek

Tato sekce se zabývá analýzou dostupných metod pro renderování [1] webových stránek a výběrem metod pro renderování stránek v rámci systému.

Next.js [8], používaný na frontendu, nabízí flexibilitu v metodách renderování stránek, což umožňuje optimalizovat výkon, zlepšit SEO [2], vylepšit uživatelskou zkušenost (UX) a, jak se ukáže v následující části, zvýšit i bezpečnost aplikace.

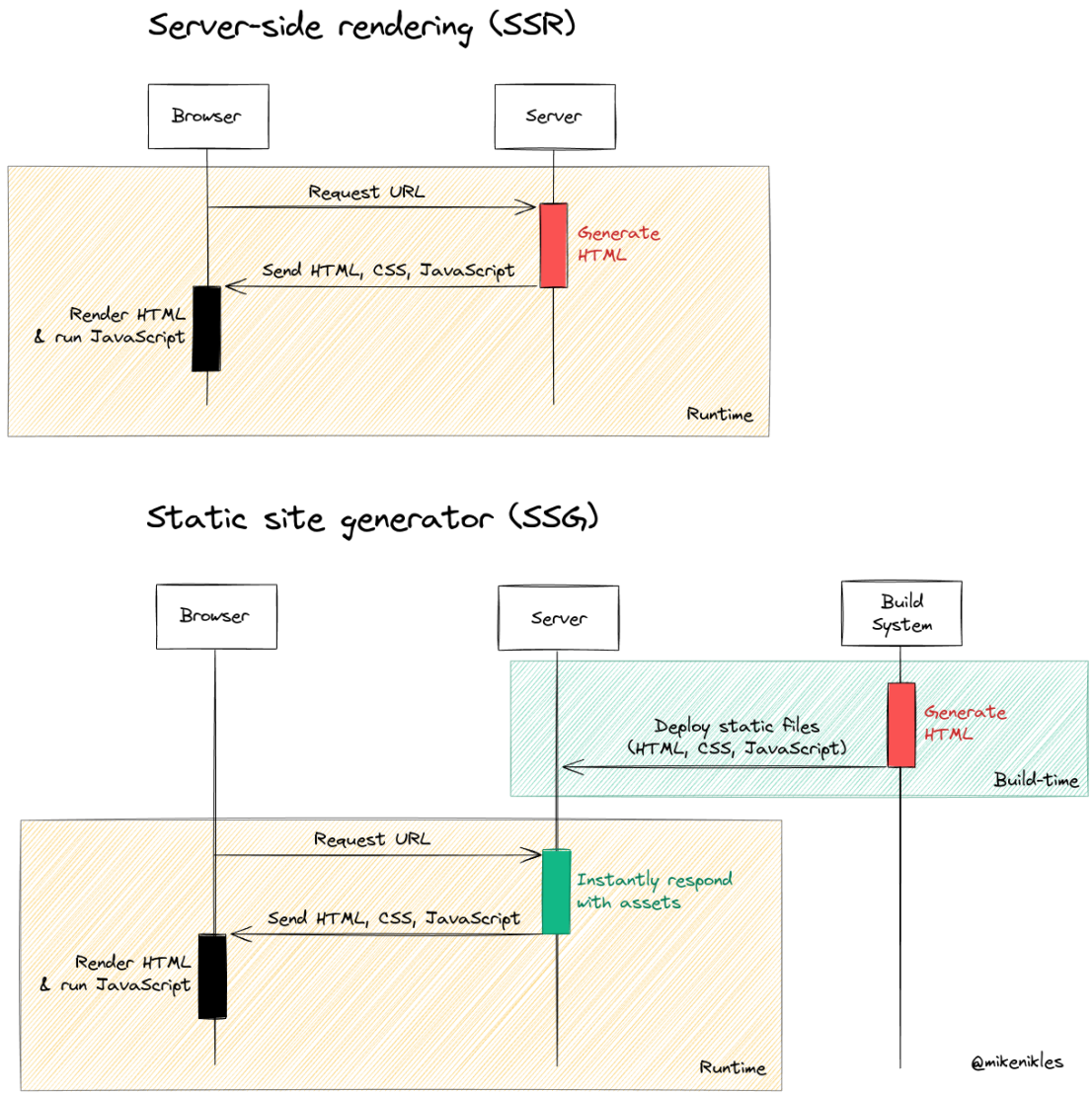
Ve výchozím nastavení Next.js pre-renderuje [9] každou stránku. To znamená, že Next.js generuje HTML pro každou stránku předem, namísto toho, aby bylo vše generováno klientem prostřednictvím JavaScriptu. Pre-rendering může vést k lepšímu výkonu a SEO.

[1] vykreslování

[2] SEO (Search Engine Optimization) je soubor technik a postupů určených k optimalizaci webových stránek pro vyhledávače s cílem zlepšit jejich pozici ve výsledcích vyhledávání a zvýšit tak organickou návštěvnost

Každé generované HTML je spojeno s minimálním JavaScriptovým kódem nezbytným pro danou stránku. Když prohlížeč načte stránku, její JavaScriptový kód se spustí a stránku učiní plně interaktivní (tento proces se v Reactu nazývá *hydratace*).

Klíčové metody zahrnují Static-Site Generation (SSG), Server-Side Rendering (SSR) a Client-Side Rendering (CSR), z nichž SSG a SSR představují pre-rendering stránky. Viz obrázek 3.1.



Obrázek 3.1 Pre-rendering stránky [10]

3.3.1 Static Site Generation (SSG)

Generování statických stránek znamená, že tyto stránky jsou celé zkompilovány do HTML již při sestavení aplikace. Když uživatel o takovou stránku požádá, je mu odeslána již zcela vygenerovaná statická HTML stránka. Navíc statické stránky lze efektivně uchovávat v mezipaměti CDN (Content Delivery Network), což umožňuje jejich opětovné využití při dalších požadavcích na stejnou stránku. To zajišťuje skoro okamžité načtení webové stránky. SSG se typicky používá pro statické webové obsahy, jako jsou blogy nebo dokumentace.

3.3.2 Server-Side Rendering (SSR)

Serverové renderování stránek znamená, že tyto stránky jsou pre-renderovány na serveru při každém požadavku. Na rozdíl od SSG je SSR vhodný pro dynamické aplikace, kde obsah závisí na uživatelských datech. Taková uživatelská data lze načíst na serveru před pre-renderingem stránky pomocí speciální funkce `getServerSideProps`.

3.3.3 Client-Side Rendering (CSR)

Klientské renderování stránek znamená, že tyto stránky jsou zcela renderovány na klientské straně, tedy v prohlížeči uživatele. Jedná se o tradiční metodu Reactu, při které server posílá prázdnou HTML stránku a veškerý obsah se pak dynamicky generuje pomocí JavaScriptu na klientské straně. Klientské renderování má velmi negativní dopad na SEO a dobu prvního načtení stránky. Přesto je stále velmi populární, hlavně kvůli své jednoduchosti a flexibilitě.

3.3.4 Shrnutí

Next.js umožňuje flexibilní kombinaci různých metod generování stránek v rámci jedné aplikace podle specifických potřeb jednotlivých stránek. Statické generování je ideální pro vytváření chybových stránek, jelikož jejich obsah se po sestavení aplikace nemění, a je tak statický. Na druhou stranu, pro část uživatelského rozhraní, která vyžaduje aktuální data, je vhodné zvážit použití SSR nebo CSR.

Přestože současné řešení formálně využívá SSR, všechny takto generované stránky se zobrazují s hlaškou „loading“ a vyžadují autentizaci na klientské straně. Poté je obsah stránek dynamicky generován pomocí JavaScriptu. To znamená, že většina úloh spojených s renderováním a autentizací probíhá v prohlížeči uživatele, což efektivně neguje všechny výhody SSR oproti CSR. Pro zlepšení této situace a efektivní využití SSR – lepší rychlosti prvního načtení a lepší vyhledatelnosti stránek — je nutné jasně identifikovat a oddělit ty části aplikace, které probíhají na klientské straně, včetně autentizace, a přesunout je na server. S tím souvisí další část, ve které se budou analyzovat další výhody provádění autentizace na serveru.

3.4 Autentizace a autorizace

Tato sekce se zabývá analýzou různých metod autentizace a autorizace v rámci aplikací Next.js a výběrem metody pro budoucí implementaci.

V současném systému je autentizace spolu s autorizací realizována na straně klienta, avšak existují alternativní přístupy k autentizaci a autorizaci v aplikacích založených na Next.js, které spočívají ve zpracování těchto procesů na serveru.

3.4.1 Autentizace na straně klienta

Autentizace na straně klienta znamená, že tento proces je prováděn klientem, tedy prostřednictvím JavaScriptu v prohlížeči uživatele. To znamená, že klient řídí autentizační a autorizační procesy – rozhoduje, které stránky a data je uživatel oprávněn číst a kam má být v případě potřeby přesměrován.

Postup autentizace na straně klienta (v prohlížeči) má následující kroky:

1. Prohlížeč odešle požadavek na Next.js server.
2. Server pre-vyrenderuje tuto stránku (tato pre-renderovaná stránka je ve skutečnosti prázdná, jelikož v tomto okamžiku není ještě známo, zda je uživatel autentizován či autorizován k prohlížení obsahu) a odešle ji klientovi.

3. Klient poté odešle požadavek na autentizaci serveru.
4. Na základě odpovědi se klient rozhodne, jak postupovat dále. Nejčastěji dojde k renderování obsahu původní stránky, pokud má uživatel příslušná oprávnění. V opačném případě je uživatel přesměrován na stránku s chybou 403 nebo na přihlašovací stránku.

Hlavními výhodami jsou snadná a rychlá implementace. Nicméně, tato metoda může představovat bezpečnostní rizika, jelikož uživatel má přímý přístup k JavaScriptovému souboru, který řídí proces autentizace a zobrazení chráněných informací.

3.4.2 Autentizace na straně serveru

Autentizace na straně serveru znamená, že tento proces je prováděn serverem – server rozhoduje, zda se daná stránka má odeslat, nebo co dělat v případě, že uživatel nemá příslušná práva. Tento postup se obvykle realizuje prostřednictvím funkce *getServerSideProps*, která se vykonává na serveru při každém požadavku na načtení nějaké stránky. Tato metoda přináší zvýšenou bezpečnost tím, že klíčové autentizační procesy jsou řízeny na serveru, což snižuje riziko úniku citlivých dat. Navíc tato metoda zvyšuje celkový výkon aplikace tím, že v případě úspěšné autorizace uživatele umožňuje pre-renderování obsahu původní stránky, a tak přispívá k výhodám SSR, které byly rozebrány v předchozí sekci.

3.4.3 Shrnutí

Autentizace na straně klienta představuje v současném systému značné bezpečnostní riziko. V případě, že se uživatel pokusí otevřít stránky, ke kterým nemá přístup, přesto může nahlédnout do citlivých informací prostřednictvím staženého JavaScriptu. Naopak autentizace na straně serveru tento problém řeší tím, že uživatelé, kteří nemají přístupová práva, jsou okamžitě přesměrováni na chybovou stránku 403, čímž se jim znemožní získat přístup k citlivým informacím.

Je patrné, že přechod na serverovou autentizaci a autorizaci nabízí bezpečnější a efektivnější řešení v porovnání s autentizací na straně klienta. Tento krok by významně zlepšil bezpečnost a celkovou výkonnost aplikace, a zároveň by položil robustní základ pro její budoucí rozvoj.

3.5 Optimalizace

Tato sekce se věnuje analýze vybraných optimalizačních technik pro frontend a návrhu jejich aplikace na pomalé a neoptimalizované části systému.

Vzhledem k tomu, že frontendový framework Next.js již provádí řadu optimalizací [11], jako je rozdělení kódu (code splitting), komprese *gzip* vyrenderovaných stránek nebo optimalizace obrázků, cílem budou jen ty vybrané technologie, které jsou pro systém relevantní. Dále budou identifikována problematická místa v systému, která by mohla profitovat z optimalizace, a bude provedena analýza aplikace vybraných optimalizačních technik.

3.5.1 Přehled optimalizačních technik a nástrojů

Níže jsou uvedeny hlavní optimalizační techniky pro frontend.

3.5.1.1 Memoizace

Memoizace v Reactu je technika používaná k optimalizaci výkonu funkčních komponent. Podstatou je ukládání výsledků náročných výpočtů nebo volání funkcí do mezipaměti. Je zvláště užitečná při práci s výpočetně náročnými nebo často volanými funkcemi se stejnými vstupními

hodnotami, protože pomáhá vyhnout se redundantním výpočtům a zlepšuje celkovou efektivitu aplikace.

V Reactu existují tři techniky memoizace: `React.memo` [12], `useMemo` a `useCallback`. Všechny tři konstrukce dělají v podstatě to samé – obalují komponenty, náročné výpočty nebo funkce, aby se zabránilo jejich opětovnému vykreslování či výpočtu, pokud zůstanou jejích závislosti (props) nezměněné. Například použitím `React.memo` je již vykreslená komponenta prohlížečem uložena do mezipaměti. Pokud se její závislosti od posledního vykreslování nezměnily, React použije předchozí již vykreslený výsledek místo toho, aby proces vykreslování opakoval. Tímto šetří čas a zdroje.

3.5.1.2 Lazy Loading komponent

Zatímco memoizace pomáhá vyhnout se opakovanému vykreslování komponent nebo náročnému výpočtu, technika *Lazy Loading* se zaměřuje na to, aby samotné vykreslování těžkých komponent nezpomalovalo celou aplikaci. React podporuje lazy loading komponent pomocí `React.lazy` [13] a `Suspense`. Tato technika umožňuje odložit načítání komponent až do okamžiku, kdy jsou skutečně potřeba. To může výrazně zlepšit dobu načítání stránky, zejména u rozsáhlých aplikací. Lazy Loading se ukazuje jako obzvláště účinný u komponent náročných na výpočet nebo vykreslení, jako jsou grafy nebo interaktivní mapy. Místo toho, aby uživatel čekal při prvotním načtení stránky na vykreslení těchto těžkých komponent, je výhodnější nejdříve načíst ostatní důležité komponenty, jako jsou tlačítka a layout. Teprve poté se načtou ty těžké komponenty, které uživatel aplikace nepotřebuje okamžitě. Je nežádoucí, aby vykreslování těchto komponent zpomalovalo nebo blokovalo start aplikace.

3.5.1.3 List Virtualization

Virtualizace seznamu [14] je technika, která významně omezuje množství operací s DOMem potřebných pro zobrazení dlouhých seznamů nebo tabulek v aplikaci. Tímto způsobem se zvyšuje výkon aplikace, protože do DOMu se v reálném čase načítají pouze ty položky seznamu, které jsou právě zobrazeny uživateli. Při skrolování jsou tyto položky dynamicky nahrazovány novými. Díky tomu dochází k výraznému snížení počtu elementů ve struktuře stránky, což zlepšuje celkovou rychlost a reakční schopnost aplikace.

Virtualizace seznamu funguje na principu vytváření „okna“ nebo „viewportu“. Okno pak zobrazuje jen omezenou část seznamu. Jak uživatel posouvá obsah, aplikace dynamicky načítá a zobrazuje položky určené k zobrazení v daném okamžiku, zatímco ty, které jsou mimo zorné pole, jsou z DOMu odstraňovány nebo nahrazovány. Tento přístup může dramaticky snížit množství zdrojů potřebných pro vykreslování dlouhých seznamů a zlepšit tak výkon aplikace.

3.5.1.4 Server-Side Rendering (SSR)

Jednou z hlavních výhod vykreslování na straně serveru je lepší uživatelský zážitek, protože uživatelé dostanou prohlížitelný obsah rychleji než u aplikací vykreslovaných na straně klienta. I když z hlediska optimalizace je vykreslování na straně serveru většinou preferovanou volbou, je důležité sledovat počet klientů a náklady spojené s výkonem serveru. Pokud server není dostatečně výkonný pro určitý počet uživatelů, může se stát, že vykreslování rozsáhlé aplikace s velkým množstvím dat na serveru bude pomalejší než vykreslování na straně klienta.

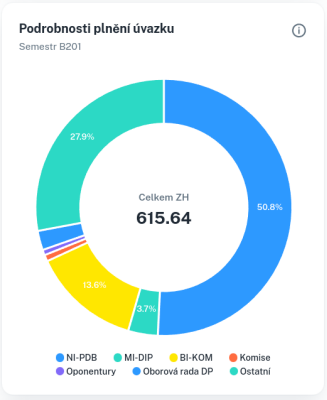
3.5.2 Analýza míst na frontendu vyžadujících optimalizaci

Dále budou identifikována hlavní problematická místa na frontendu z pohledu optimalizace v současném systému.

3.5.2.1 Vykreslovaní grafu

V současném systému na stránce s informacemi o učiteli se nachází několik komponent, které mají za úkol vykreslovat grafy ukazující naplnění úvazků nebo složení těchto úvazků. Vykreslování a s nim spojené výpočty probíhá na straně klienta, tedy přímo v prohlížeči. Aktuální implementace těchto grafů nevyužívá memoizaci, což vede k opakovanému a zbytečně náročnému vykreslování komponent. Tento problém je možné vyřešit použitím techniky `React.memo`, která umožní obalit komponentu s grafem do wrapperu. Tento wrapper pak řídí znovuvykreslení grafu pouze v případě, že dojde ke změně dat, která graf vyžaduje. Například graf zobrazující plnění úvazků, viz obrázek 3.2, by se měl aktualizovat pouze pokud dojde ke změně informací o úvazcích učitelů.

Nicméně, samotné vykreslení grafu při spuštění aplikace může být pomalé a prodloužit tak čas načítání stránky. Řešením může být využití techniky Lazy Loading, díky které se graf vykreslí až po načtení všech ostatních, hlavních částí systému. To umožní uživateli ihned po načtení stránky interagovat s ostatními prvky.



Obrázek 3.2 Graf plnění úvazků

3.5.2.2 Načítání tabulkových dat

Načítání tabulkových dat probíhá na straně klienta v prohlížeči. Tato data zahrnují například seznam všech učitelů nebo seznam všech předmětů v semestru. Přestože tabulka obsahuje funkci stránkování, data se vždy načítají celá. To může představovat velký optimalizační problém pro větší aplikace, které v tabulkách obsahují tisíce záznamů. Tento přístup výrazně zpomaluje počáteční načítání a uživatel tak musí dlouho čekat. Nejlepším řešením by bylo, aby se klient dotazoval backendu pouze na určitou část dat tabulky. Například, by se nejprve načetlo pouze prvních 10 řádků tabulky, a poté by se klient dotazoval serveru na dalších 10 řádků atd. Nicméně, v rámci systému Úvazkostroj by to nepřineslo tak výraznou výhodu, jelikož objem dat není tak velký a v budoucnu se očekává jen mírný nárůst. Jde pouze o desítky KB dat, což je v dnešní době s rychlostí internetového připojení zanedbatelně málo a načítání všech dat najednou nemá výrazný vliv na uživatelskou zkušenost.

3.5.2.3 Zobrazení tabulkových dat

I když velikost načítaných dat nemusí být kritickým problémem, jejich množství potřebné k vykreslení může představovat významný optimalizační problém. Pokud uživatel vybere zobrazení všech dat v tabulce najednou, může se výrazně zpomalit načítání stránky v prohlížeči. Uživatel pravděpodobně nepotřebuje vidět tisíce řádků dat najednou na jedné stránce, ale spíše by preferoval postupné procházení dat. Zde se nabízí použití techniky virtualizace seznamů, která umožňuje zobrazovat a vykreslovat v JavaScriptu pouze omezenou část seznamu. Když uživatel prochází

například seznam všech semestrů, aplikace dynamicky načítá a zobrazuje předměty, které mají být v daném okamžiku viditelné, zatímco ty mimo zorné pole jsou z DOMu odstraněny nebo nahrazeny.

3.5.2.4 Řazení a vyhledávání tabulkových dat

Řazení dle jednotlivých sloupců a vyhledávaní v tabulce mohou být v systému velmi užitečné. Tyto funkce představují důležitý aspekt z pohledu optimalizace, jelikož obě operace – řazení i vyhledávání – mohou být poměrně náročné. Nejjednodušší přístup by spočíval v řazení dat přímo v komponentě s tabulkou při každém jejím vykreslení. Avšak efektivnějším řešením je použití nástroje `useMemo` pro obalení procesu řazení a vyhledávání do wrapperu. Tento wrapper pak spustí proces pouze v případě, že došlo ke změně dat nebo parametrů řazení či vyhledávání.

3.6 Testování

V této části budou vybrány a analyzovány různé typy testů a nástrojů, spolu s jejich uplatněním a přínosy pro různé části systému Úvazkostroj.

3.6.1 Testování databázové části

Vzhledem k tomu, že veškerá obchodní logika je uložena v databázi, je důležité zvážit možnosti jejího testování a určit, zda se takové testování vyplatí.

3.6.1.1 Unit testy

Pro ověření správnosti funkcí, triggerů a procedur napsaných v PL/pgSQL, které tvoří jednotlivé komponenty obchodní logiky databáze, lze využít unit testy. Tyto testy umožňují izolovaně testovat správnost výstupů těchto komponent na základě definovaných vstupů. Existuje několik frameworků pro unit testy v PostgreSQL, například *pgTAP* [15], který poskytuje sadu nástrojů pro psaní a spouštění testů přímo v databázi. V rámci systému mohou unit testy pomoci ověřit funkčnost logiky pro výpočet započitatelných hodin. Nicméně, testování databázové části a obchodní logiky v PL/pgSQL může být poměrně složité a méně běžné. Vzhledem k tomu, že backend aplikace často slouží jako prostředník mezi databází a uživatelským rozhraním, může být efektivnější soustředit se na testování backendu. Tento přístup umožňuje komplexně ověřit funkčnost celého backendu, včetně interakcí s databází a správného provádění obchodní logiky, což vede k efektivnějšímu testovacímu procesu.

3.6.1.2 Výkonnostní testy

Testy výkonnosti databáze jsou nezbytné pro ověření, zda databáze a uložené procedury vyhovují požadavkům na rychlost a možnost škálování. Nástroje jako *pgBench* [16] umožňují simulovat zátěž na databázi a analyzovat její výkonnost v různých situacích. Výkonnostní testy databáze mohou identifikovat problémy spojené s rychlostí aplikace, pokud je doba čekání na data z databáze příliš dlouhá a jsou potřebné různé databázové optimalizace. Následně lze pomocí výkonnostních testů ověřit efektivitu těchto optimalizací.

3.6.2 Testování backendové části

Vzhledem k tomu, že obchodní logika je součástí databázové vrstvy, můžeme tuto logiku testovat i na backendu. Pro testování backendové části systému můžeme využít následující typy testů.

3.6.2.1 Unit testy

Unit testy backendové části by měly být zaměřeny na testování jednotlivých komponent, jejichž účelem je vracení výsledků výpočtů funkcí z databáze, například pro výpočet plnění úvazků. Nicméně, protože backend primárně zprostředkovává data z databáze a případně je modifikuje, je vhodné přímo testovat API.

3.6.2.2 API testy

API testy, speciální případ integračních testů, jsou vhodné pro testování celé backendové části prostřednictvím REST API rozhraní, které poskytuje. Tyto testy by měly ověřit funkčnost vybraných endpointů, konkrétně by měly kontrolovat validitu HTTP požadavků, statusy odpovědí, tělo odpovědí a samozřejmě autentizaci s autorizací a komunikaci s databází. K dispozici je mnoho nástrojů a frameworků pro testování API, z nichž každý nabízí specifické funkce a výhody.

- **Postman** je uživatelsky přívětivý nástroj určený pro testování API, který umožňuje snadno vytvářet, sdílet, testovat a dokumentovat API. Nabízí rozsáhlou sadu nástrojů pro práci s API, včetně možnosti vytváření složitých požadavků, testování autentizace, procházení historie požadavků a mnoho dalšího.
- **cURL** je příkazový řádek a knihovna pro přenos dat s podporou různých protokolů. cURL je oblíbený pro svou univerzálnost a schopnost být použit ve skriptech a automatizaci. Přestože nabízí méně uživatelsky přívětivé rozhraní než Postman, je to silný nástroj pro pokročilé uživatele, kteří preferují práci přímo z příkazového řádku.
- **Insomnia** je další moderní open-source nástroj pro testování API, který se zaměřuje na jednoduchost a efektivitu. Je velmi podobný Postmanu, ale je lehčí a jednodušší; nemá však automatizovanou dokumentaci API.

Pro testování API v systému se hodí zejména Postman, hlavně kvůli jeho popularitě a rozsáhlým možnostem automatizovaného testování a integrace v CI/CD procesech. [17] [18] [19]

3.6.3 Testování frontendové části

Při testování frontendové části je vhodné zvážit jak unit testy jednotlivých komponent, tak i end-to-end testování.

3.6.3.1 Unit testy

Unit testy se používají pro ověření funkčnosti jednotlivých React komponent uživatelského rozhraní. Například testování komponenty reprezentující tabulku by mohlo spočívat v ověření, zda počet vykreslených řádků odpovídá počtu prvků vstupního pole. Psaní těchto testů může být časově náročné, jelikož uživatelské rozhraní se mezi všemi komponentami systému mění nejčastěji.

3.6.3.2 E2E testy

End-to-end (E2E) testy jsou užitečné pro automatizovanou simulaci uživatelského chování na uživatelském rozhraní. K tomu slouží různé nástroje a frameworky, například *Cypress* [20]. Psaní E2E testů je obecně mnohem složitější než psaní unit testů pro React komponenty.

3.6.3.3 Uživatelské testování

Uživatelské testování se zaměřuje především na použitelnost uživatelského rozhraní a funkčnost aplikace. Provádějí ho skuteční uživatelé, tj. potenciální nebo stávající uživatelé dané aplikace, protože vývojáři nebo testeři nejsou schopni věrně napodobit chování nového reálného návštěvníka webu.

Uživatelské testování obvykle probíhá tak, že koncový uživatel dostane úkol nebo sadu úkolů (scénář), které se snaží splnit. K uživateli je přiřazen pozorovatel, který zaznamenává chování a postup uživatele včetně úvah. Po skončení testu uživatel odpovídá na otázky týkající se průběhu testování a jeho dojmů z aplikace.

3.6.4 Shrnutí a výběr testů

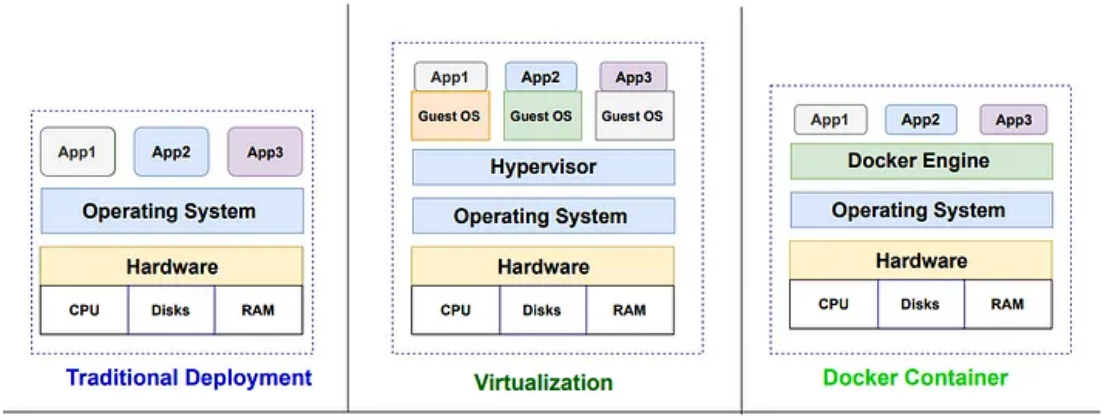
Při testování backendové části systému se jako nejvhodnější ukázaly API testy. Důvodem je jejich schopnost pokrývat nejen funkčnost obchodní logiky z databáze, ale také autentizaci a autorizaci API a validaci HTTP požadavků.

Pro testování frontendové části bylo po dohodě s vedoucím práce, Ing. Michalem Valentou, Ph.D., zvoleno uživatelské testování. K tomuto účelu budou připraveny scénáře, které uživatelé budou následovat.

3.7 Nasazení

Tato sekce se věnuje analýze různých metod nasazení aplikací a návrhu nejvhodnější metody a CI/CD infrastruktury.

Nasazení představuje klíčovou fázi vývojového cyklu softwaru, jelikož zahrnuje všechny kroky a procesy nezbytné pro zpřístupnění softwaru uživatelům. Existují různé přístupy k nasazení aplikací, z nichž každý má své specifické výhody a nevýhody. Všechny dostupné metody nasazení jsou zachyceny na obrázku 3.3.



Obrázek 3.3 Metody nasazení

3.7.1 Tradiční servery

Nasazení aplikace přímo na fyzickém serveru představuje tradiční způsob, který se používá i v současnosti. Na jednom fyzickém serveru může být nasazeno několik aplikací, které sdílejí jedno prostředí a operační systém.

Proces nasazení na fyzický server lze zjednodušeně popsat takto:

1. Nahrání aplikace na server prostřednictvím FTP nebo SSH.
2. Konfigurace serveru včetně instalace nástrojů a závislostí potřebných pro aplikaci.
3. Spuštění aplikace na serveru.

Hlavní nevýhodou přístupu s tradičními servery je absence izolace zdrojů mezi aplikacemi. Aplikace sdílejí stejné prostředí a může docházet ke konfliktům závislostí. Další nevýhodou je špatná škálovatelnost – není možné efektivně rozdělovat serverové zdroje, jako jsou CPU a paměť, mezi jednotlivé aplikace. Kromě toho absence možnosti nastavit zdroje pro konkrétní aplikace může vést k situaci, kdy nadměrné využití zdrojů jednou aplikací může způsobit pád celého fyzického serveru.

3.7.2 Virtualizace

Virtualizace je technika umožňující provozovat několik virtuálních strojů na jednom fyzickém serveru. Každý virtuální stroj obsahuje plnou kopii operačního systému, aplikaci, nezbytné binární soubory a knihovny. Tím řeší problém tradičních serverů týkající se izolace prostředí jednotlivých aplikací. O správu virtuálních počítačů se stará hypervizor, který umožňuje konfigurovat zdroje pro jednotlivé virtuální počítače. Tak řeší i další problém tradičních serverů tím, že umožňuje nastavit, kolik zdrojů fyzického stroje bude využívat každý virtuální počítač s aplikací. Při pádu aplikace na nějakém virtuálním stroji to nijak neovlivní ostatní virtuální stroje.

Avšak virtualizace má i své nevýhody. Hlavní nevýhodou je, že operační systém každého virtuálního stroje vyžaduje značné množství paměti, často desítky GB. Je to také velice pomalé řešení, protože je třeba operační systém virtuálního počítače spustit, což může trvat několik minut.

3.7.3 Kontejnerizace

Kontejnerizace představuje proces virtualizace na úrovni operačního systému, při kterém dochází k vytváření kontejnerů. Kontejner je specifická abstrakce na úrovni operačního systému, která umožňuje seskupení aplikace spolu s jejími závislostmi do jednotného balíčku. Tento přístup má za následek, že kontejnery vyžadují méně prostoru než tradiční virtuální stroje a spouštějí se velmi rychle, obvykle během několika sekund či milisekund.

Díky efektivnímu využití podkladového hardwaru kontejnery eliminují plýtvání zdroji, což je běžný problém při používání tradiční virtualizace. Kontejnery typu *Docker* [21] jsou izolovány na úrovni procesů a pro svůj běh nevyžadují hardwarový hypervizor, což je činí vysoce přenositelnými. Díky těmto vlastnostem představuje kontejnerizace efektivní a flexibilní řešení pro nasazování aplikací ve více prostředích s minimálními požadavky na infrastrukturu.

3.7.4 Výběr způsobu nasazení

Současným řešením nasazení je tradiční způsob nasazení na server s operačním systémem *Linux Alpine*. Na základě požadavku vedoucího této práce, Ing. Michala Valenty, Ph.D., bylo rozhodnuto přejít k provozování aplikace na serveru s operačním systémem *Ubuntu*. Jako nejvhodnější se ukázala metoda kontejnerizace s využitím nástroje Docker.

3.7.5 Návrh CI/CD infrastruktury

CI/CD je automatizovaný proces skládající se z kontinuální integrace (CI) a kontinuálního nasazení (CD). Kontinuální integrace zahrnuje automatické sestavování a testování kódu aplikace po každé změně, zatímco kontinuální nasazení umožňuje automatické nasazování změn do produkčního prostředí po úspěšném běhu testů. Tento přístup zajišťuje rychlejší vývoj a nasazení aplikací, vyšší kvalitu a menší riziko.

Aktuálně podporované CI/CD infrastruktury se omezují pouze na jedno vývojové prostředí (DEV). Pro efektivní vývoj a provoz aplikace je však nezbytné provozovat více prostředí, z nichž každé plní specifický účel:

- **TEST** — prostředí určené pro prvotní fázi testování. Vývojáři a testeři zde provádějí unit testy, integrační testy a další automatizované testy k ověření funkcionalit nově vyvinutého kódu.
- **STAGE** — předprodukční prostředí, které co nejvíce napodobuje produkční prostředí. Provádějí se zde uživatelské testy, bezpečnostní testy a zátěžové testy, aby bylo zajištěno, že aplikace je připravena pro nasazení do produkce.
- **PROD** — produkční prostředí, ve kterém běží finální verze aplikace dostupná koncovým uživatelům.

Dalším krokem ve vývoji stávající CI/CD infrastruktury bude zavedení Dockeru pro kontejnerizaci. Pro backend i frontend bude potřeba vytvořit *Dockerfile*, což je soubor určený ke generování *Docker image*. Tyto image představují způsob efektivního přenosu zabalené aplikace spolu s jejími závislostmi. Je třeba zřídit *Docker repozitář*, kam se v rámci CI/CD pipeline na GitLabu budou tyto image ukládat. Při fázi nasazení by server určený pro konkrétní prostředí měl načíst daný image a spustit na jeho základě kontejner. Ideálně by jeden image měl sloužit pro nasazení ve všech prostředích, což zvyšuje efektivitu a usnadňuje správu verzí aplikace.

Kapitola 4
Implementace

Cílem této kapitoly je popsat implementaci vybraných částí systému.

V backendové části byla vytvořena kompletní *Swagger* [22] dokumentace k API a přidán endpoint pro aktualizaci parametrů semestru. Původně bylo plánováno také implementovat endpoint pro vytvoření nového semestru, ale po konzultaci s vedoucím práce, Ing. Michalem Valentou, Ph.D., bylo od tohoto záměru upuštěno. Vytváření nového semestru bude realizováno pouze pomocí SQL skriptu v databázi. Následující sekce budou věnovány popisu implementace frontendové části.

4.1 Integrace TypeScriptu do vývoje

Prvním krokem bylo nastavení striktní typové kontroly v konfiguračním souboru TypeScriptu. Tento krok zpřísnil typovou bezpečnost a pomohl identifikovat a opravit běžné chyby, jako jsou:

- **Implicitní používání typu `any`** – bez explicitního určení typu se předpokládá typ `any`, a tím se vypíná typová kontrola pro danou funkci nebo objekt. Striktní režim vynucuje explicitní definování typu každé proměnné, funkce nebo objektu, pokud sám nemůže jednoznačně odvodit typ. Byla provedena analýza, jaké typy mohou jednotlivé komponenty systému nabývat, a byly vytvořeny příslušné typy.
- **Neošetřování `undefined` a `null` hodnot** – v běžném režimu TypeScript dovoluje přiřadit `null` nebo `undefined` k jakémukoliv typu. Striktní režim vynucuje, aby se explicitně ošetřovaly tyto hodnoty. Proto byla provedena analýza výskytu možných nedefinovaných hodnot a přidány různé kontrolní mechanismy, aby takové hodnoty byly očekávány a ošetřovány správně.

V TypeScriptu existují dva konstrukty pro definování typů: `interface` a `type`. Jsou velmi podobné, ale `type` nabízí větší flexibilitu. Jako osvědčený postup se doporučuje v kódové bázi používat pouze jeden z těchto dvou pro konzistenci. Pro použití v systému byl zvolen `interface` kvůli jeho jednoduchosti a přehlednosti.

Dalším krokem bylo rozhodnout, kde v kódu definovat typy, aby byly přehledné a konzistentní. Obecně neexistuje jediný nejlepší způsob. Z principů softwarového inženýrství a doporučení týmu TypeScript by typy měly být v souborech `types.ts`, kde budou všechny typy nějakého modulu v systému organizovány. Například, modul, reprezentující tabulku v UI, byl organizován následovně:

```
table
├─ Table.tsx
├─ TableBody.tsx
├─ TableHead.tsx
├─ TableRow.tsx
├─ ...
└─ types.ts
```

Soubor `types.ts`, který definuje znovupoužitelné typy pro modul tabulky, je uveden v ukázce kódu 4.1.

```
export type Align = "left" | "right" | "center"
export type Order = "asc" | "desc"
export type CellValue = string | number

export interface Row {
    [key: string]: CellValue
}

export interface Column<RowType extends Row,
                        K extends keyof RowType = keyof RowType> {
    id: K
    label: string
    isSortable: boolean
    align?: Align
    minWidth?: string
    width?: string
    renderCell?: (cellRow: RowType, cellValue: RowType[K]) => React.ReactNode
}

...
```

Výpis kódu 4.1 Definice souboru `types.ts` pro modul reprezentující tabulku v uživatelském rozhraní

Jak je patrné, tento přístup umožňuje jasně oddělit část s definicemi typů od části s funkcionalitou.

Avšak pokud se jedná o typ specifický pro danou komponentu, je vhodnější jej mít umístěný blíže k samotné komponentě. Například u komponenty reprezentující tlačítko pro smazání v uživatelském rozhraní může být typ, který reprezentuje její vstupní hodnoty, definován přímo před ní. Důvodem je, že se jedná o velmi specifický typ, který není znovupoužitelný na více místech v kódu. Viz výpis kódu 4.2.

```
interface DeleteIconButtonProps extends IconButtonProps {
    visible: boolean;
}

const DeleteIconButton: React.FC<DeleteIconButtonProps> = ({
    visible,
    ...props
}) => {
    ...
}
```

Výpis kódu 4.2 Definice typu pro komponentu tlačítka Smazat

Tento způsob umožňuje udržet kód komponenty a její typovou definici v těsné blízkosti, což zjednodušuje orientaci v kódu a jeho údržbu.

4.2 Implementace řazení, filtrů a stylizace tabulek

Po konzultaci s vedoucím práce, Ing. Michalem Valentou, Ph.D., byla provedena řada vylepšení tabulek z hlediska uživatelského rozhraní (UI) a uživatelské zkušenosti (UX). Byla implementována funkce řazení tabulek podle jednotlivých sloupců, ve kterých má řazení smysl. Dále byly do tabulek přidány filtry. Například tabulka s přehledem pokrytí předmětů nabízí filtry pro zobrazení předmětů specifických kateder a pro zobrazení všech nepokrytých předmětů. Stylisticky byly tabulky také vylepšeny, konkrétně byly upraveny vhodné šířky sloupců pro lepší přehlednost a konzistenci napříč všemi stránkami s tabulkami. Navíc byl první sloupec v tabulce, který často plní roli primárního klíče, nastaven jako `sticky`, což uživatelům pomáhá neztratit se v tabulkách s mnoha sloupci.

4.3 Implementace SSR

Jak již bylo zmíněno v analytické části této práce, použití Server-Side Rendering (SSR) přináší mnoho výhod, a proto bylo nezbytné jej implementovat pro různé funkcionality aplikace, jako jsou autentizace a autorizace, validace HTTP požadavků atd. Vzhledem k tomu, že v Next.js 12 je SSR realizována pomocí funkce `getServerSideProps` v souboru stránky, pro kterou je potřeba serverovou logiku implementovat, představovalo toto velkou výzvu, jak efektivně implementovat různé aspekty SSR pro konkrétní stránky. Bylo třeba najít řešení, které by umožňovalo na každé stránce definovat, jaká serverová logika je potřebná. Například na stránce s informací o semestru je nutná serverová logika pro validaci HTTP požadavků na neexistující semestr, logika pro autorizaci – pouze administrátor může tuto stránku prohlížet, a tak dále.

Cílem je sestavit složitou funkci `getServerSideProps`, která obsahuje a kombinuje různé části serverové logiky, jako je validace HTTP požadavků nebo autentizace. Jako nejvhodnější řešení se jeví použití návrhového vzoru *Builder* [23], který umožňuje postupně konstruovat složitý objekt. Byl implementován Builder pro konstrukci výsledného `props` objektu, který reprezentuje celkový výsledek všech serverových logik. Tento `props` se pak předává komponentě stránky. Viz ukázka kódu 4.3.

```
export class GetServerSidePropsBuilder {
    private gsspFunctions: GetServerSideProps[] = []

    add(gsspFunc: GetServerSideProps) {
        this.gsspFunctions.push(gsspFunc)
        return this
    }

    build(): GetServerSideProps {
        return async (context) => {
            let accProps: any = {}

            for (const gsspFunc of this.gsspFunctions) {
                const retProps = await gsspFunc(context)
                accProps = _.merge(accProps, retProps)
            }

            return accProps
        }
    }
}
```

Výpis kódu 4.3 Builder třída pro sestavení výsledku serverové logiky

Pro každý aspekt serverové logiky bylo nutné implementovat funkci, která vrací funkci typu `GetServerSideProps`. Byly vytvořeny funkce pro přidání autentizace – `withAuth`, pro přidání validace HTTP požadavků – `withUrlValidation` a další. Implementace těchto funkcí bude podrobně rozebrána v následujících sekcích. Sestavení funkce `getServerSideProps` pro stránku s učitelem je ukázáno v následujícím kódu 4.4.

```
export const getServerSideProps = new GetServerSidePropsBuilder()
    .add(withAuth([roleTypes.Admin, roleTypes.DepartmentAdmin]))
    .add(
        withTranslations([
            "common",
            "departments",
            "employmentStatuses",
            "positions",
        ])
    )
    .add(withUrlValidation())
    .build()
```

Výpis kódu 4.4 Sestavení funkce `getServerSideProps` pro stránku s učitelem

4.4 Implementace autentizace a autorizace na serverové straně

Pro zajištění autentizace a autorizace na serverové straně byla vytvořena funkce `withAuth`, viz výpis kódu 4.5. Tato funkce je navržena tak, aby byla použita jako middleware v rámci pro-

cesu serverového zpracování požadavku. Funkce `withAuth` umožňuje definovat, které uživatelské role mají přístup k dané stránce, a zajišťuje, že pouze autentizovaní a autorizovaní (na základě vybrané role) uživatelé mohou přistupovat k obsahu této stránky.

```
export default function withAuth(allowedRoles?: string[]): GetServerSideProps {
    return async (context) => {
        const session = await getServerSession(
            context.req,
            context.res,
            authOptions
        )

        if (!session) {
            // code to redirect user to login page
            // ...
        }

        const cookies = parse(context.req.headers.cookie || "")
        let selectedRole = cookies.selectedRole
        const userRoles = getRoles(session)
        const pageRequiresRole = allowedRoles !== undefined

        if (userRoles.length === 0) {
            if (!pageRequiresRole) {
                return { props: { session } }
            }

            return {
                // code to redirect user to 403 page
                // ...
            }
        }

        // code to set default selectedRole if not selected
        // ...

        if (pageRequiresRole && !allowedRoles.some((r) => r === selectedRole)) {
            return {
                // code to redirect user to 403 page
                // ...
            }
        }

        return { props: { session, selectedRole } }
    }
}
```

Výpis kódu 4.5 Funkce `withAuth` pro zajištění autentizace a autorizace

Kompletní implementaci funkce `withAuth` lze nalézt v příloze, konkrétně ve zdrojovém kódu frontendové části.

4.5 Odstranění Redux

Po odstranění Reduxu, jak bylo popsáno v analytické části této práce, bylo potřeba implementovat ukládání vybraného kódu semestru do URL a vybranou roli uživatele do cookies.

Pro uložení vybraného kódu semestru do URL byla vytvořena složka `[semesterCode]`, což v Next.js označuje *dynamic route* [24]. Do této složky bylo následně potřeba přenést strukturu ostatních stránek, aby bylo pak možné z jakékoliv vnořené stránky získat kód semestru pomocí hooku `useRouter`.

Ukládání vybrané role uživatele do cookies bylo provedeno triviálně pomocí knihovny *cookie* [25] pro serverovou stranu ve funkci `withAuth`, a pomocí knihovny *js-cookie* [26] pro klientskou stranu v hooku `useSelectedRole`.

4.6 Provedení optimalizace

Implementace různých optimalizačních technik vychází z dříve identifikovaných neoptimalizovaných míst ve frontendové části aplikace, jak bylo uvedeno v analytické kapitole.

4.6.1 Optimalizace vykreslování grafu

Pro optimalizaci vykreslování grafu bylo nutné implementovat dvě techniky: Lazy Loading a memoizaci. Lazy Loading již byl implementován pomocí speciálního konstruktu Next.js `dynamic`, který je alternativou k `React.lazy`. Ukázka kódu 4.6 ilustruje toto použití.

```
const Chart = dynamic(() => import("react-apexcharts"), { ssr: false })
```

Výpis kódu 4.6 Použití `dynamic` pro Lazy Loading grafu

Vzhledem k tomu, že žádná komponenta s grafem nezávisí na vstupních parametrech, byly tyto komponenty obaleny konstrukcí `React.memo`, což zabránilo zbytečnému překreslování. Následující ukázka kódu 4.7 pro komponentu `WorkloadBySemester` demonstruje použití memoizace.

```
export default React.memo(WorkloadBySemester)
```

Výpis kódu 4.7 Použití `React.memo` pro optimalizaci komponenty `WorkloadBySemester`

Tímto způsobem bylo dosaženo efektivnějšího vykreslování grafů, snížení zátěže na prohlížeči uživatele a zlepšení celkové uživatelské zkušenosti.

4.6.2 Optimalizace řazení a vyhledávání tabulkových dat

Vzhledem k tomu, že řazení a vyhledávání jsou náročné operace, zejména pokud jde o velký objem dat, je důležité tyto operace provádět pouze pokud je to opravdu nezbytné. K tomu byla využita technika memoizace a speciální React konstrukt `useMemo`. Jak ukazuje následující příklad kódu 4.8 pro řazení řádků, byly jako závislosti určeny proměnné, jejichž změna vyvolá znovu provedení řazení. Těmito závislostmi jsou samotné řádky, pořadí řazení a sloupec, podle kterého se má řadit.

```
const sortedRows = useMemo(() => {
    if (!orderBy) return rows

    return rows.slice().sort((a, b) => {
        const valueA = a[orderBy]
        const valueB = b[orderBy]
        if (valueA < valueB) return order === "asc" ? -1 : 1
        if (valueA > valueB) return order === "asc" ? 1 : -1
        return 0
    })
}, [rows, order, orderBy])
```

Výpis kódu 4.8 Optimalizace řazení řádků tabulky

Pro optimalizaci vyhledávání tabulkových dat byla použita kombinace React hooků `useState` a `useEffect`, které společně poskytují stejnou úroveň optimalizace jako `useMemo`. Toto řešení bylo zvoleno, protože filtrované řádky se v komponentě používají a mění na mnoha dalších místech. Následující ukázka kódu 4.9 demonstruje optimalizaci vyhledávání v tabulce s učiteli.

```
const [searchQuery, setSearchQuery] = useState("")
const [filteredTeacherRows, setFilteredTeacherRows] =
    useState<TeacherRow[]>([])

...

useEffect(() => {
    // teacherRows are unfiltered rows that came from the API
    setFilteredTeacherRows(filterByQuery(teacherRows, searchQuery))
}, [teacherRows, searchQuery])
```

Výpis kódu 4.9 Optimalizace vyhledávání v tabulce s učiteli

4.6.3 Ostatní optimalizace

Další optimalizace spočívaly v memoizaci dalších výpočtů nebo zabránění zbytečné re-inicializaci velkých objektů. Jako příklad lze uvést memoizaci výpočtu subpole všech řádků v tabulce, které označuje pouze tu část řádků, které uživatel vidí na vybrané stránce tabulky. Bylo nastaveno, aby se výpočet spouštěl znovu pouze v případě změny dat, vybrané stránky v paginaci tabulky nebo parametru určujícího počet řádků na stránku, viz kód 4.10.

```
const visibleRows = useMemo(() => {
    return rows.slice(page * rowsPerPage, page * rowsPerPage + rowsPerPage)
}, [rows, page, rowsPerPage])
```

Výpis kódu 4.10 Optimalizace výpočtu viditelných řádků v tabulce

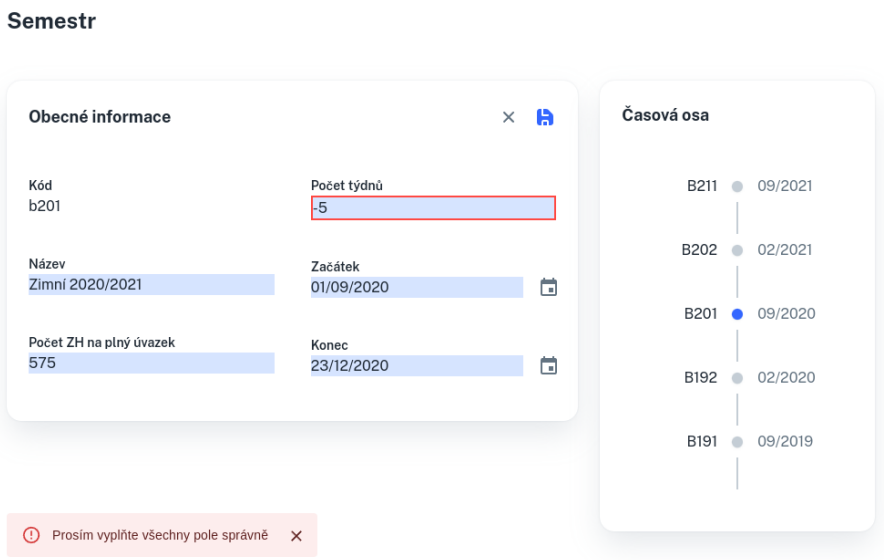
4.7 Zpracování chyb

Dalším úkolem bylo identifikovat místa v aplikaci, kde mohou potenciálně vzniknout chyby, ať už z důvodu uživatelské chyby nebo selhání jiných částí systému, jako je backend nebo databáze.

Důležité je správně na tyto chyby reagovat, obvykle zobrazením srozumitelné chybové hlášky uživateli, aby bylo jasné, co se stalo a co je třeba změnit.

4.7.1 Chyby ve formuláři

Chyby ve formuláři se stávají, pokud uživatel zadá do nějakého pole ve formuláři nevalidní údaj. Je nutné uživatele o této situaci informovat a v uživatelském rozhraní to jasně označit. Jelikož formuláře mohou být často rozsáhlé a rozdělené do několika stránek, může se stát, že si uživatel chyby nevšimne nebo dané pole s nevalidním údajem není na obrazovce vidět. V takovém případě, pokud uživatel stiskne tlačítko ”uložit”, systém znovu zkontroluje validnost všech polí ve formuláři. Pokud některé z nich jsou nevalidní, zobrazí se uživateli chybová hláška. Implementace těchto hlášek v uživatelském rozhraní je zobrazena na obrázku 4.1.



Obrázek 4.1 Příklad zobrazení chybové hlášky ve formuláři

4.7.2 Chyby 404

Kontrolu chyb typu 404 je efektivní provádět na serverové straně při každém požadavku na stránku. Pro tento účel byla vytvořena funkce `withUrlValidation`, která ověřuje, zda data odkazovaná z URL existují. Ukázka kódu 4.11 demonstruje, jak tato funkce funguje.

```
export default function withUrlValidation(): GetServerSideProps {
    return async (context) => {
        const urlSemesterCode = getUrlParam(context.params, "semesterCode")

        if (urlSemesterCode) {
            const found = await dataExists(`/semester/${urlSemesterCode}`)
            if (!found) {
                return {
                    notFound: true,
                }
            }
        } else {
            return { props: {} }
        }

        // here goes similar code for teacher username, course code
        // ...
    }
}
```

Výpis kódu 4.11 Funkce `withUrlValidation` pro kontrolu existence dat odkazovaných z URL

Tento přístup zajišťuje, že pokud uživatel zadá do prohlížeče URL adresu, která odkazuje na neexistující data (například neexistující kód semestru, učitele nebo kurz), server automaticky vrátí chybu 404, čímž informuje uživatele o problému a zlepšuje celkovou uživatelskou zkušenost při navigaci v aplikaci.

4.7.3 Chyby API

Často se může stát, že HTTP požadavek odeslaný klientem na backend selže. V současném systému se může stát, že klient bude neustále opakovat stejný dotaz v naději na úspěch. Tento přístup však není správný, a je třeba po několika pokusech přestat a uživateli zobrazit chybovou hlášku. Toto chování bylo nakonfigurováno pomocí globální konfigurace SWR, viz ukázka kódu 4.12. Klient tedy pošle na backend API požadavek pouze třikrát s intervalem 3 sekundy, a pokud zase selže, zobrazí se chybová hláška s informacemi o chybě.

```
const ApiConfig: React.FC = ({ children }) => {
    const { showError } = useErrorNotification()

    return (
        <SWRConfig
            value={{
                onError: (error, key) => {
                    if (
                        error.response.status !== 403 &&
                        error.response.status !== 404
                    ) {
                        showError(
                            `API Error on '${key}' endpoint:
                            ${error.response.status}
                            ${error.response.statusText}`
                        )
                    }
                },
                errorRetryCount: 3,
                errorRetryInterval: 3000,
            }}
        >
            {children}
        <SWRConfig>
    )
}
```

Výpis kódu 4.12 Konfigurace SWR pro zpracování chyb API

Tento přístup zlepšuje způsob, jakým aplikace reaguje na chyby API, a zajišťuje, že uživatelé jsou informováni o problémech, což jim umožňuje přijmout případné kroky pro jejich řešení.

4.8 Změna způsobu internacionalizace a lokalizace i18n

Současné řešení využívá internacionalizaci hlavně jako řešení na straně klienta, s možností cachování překladů v lokálním úložišti prohlížeče. Toto řešení je typické pro React aplikace, ale nevyužívá všech výhod Server-Side Rendering (SSR) Next.js, o kterých bylo podrobně zmíněno dříve. Bylo potřeba nahradit současný systém i18n z dvou důvodů:

1. Jelikož bylo implementováno SSR pro načítání stránek, aby klient již dostal HTML skoro zcela vygenerované stránky, tak spolu se současným překladem na straně klienta to vypadalo tak, že uživatel na chvíli na začátku viděl nepřeložené texty, a jen poté co se tento překlad dostal z backendu, tak viděl klient texty přeložené. Toto je špatná uživatelská zkušenost.
2. Současné řešení nepodporuje zadání jazyka v URL, a proto nelze sdílet odkaz na jinou verzi stránky webové aplikace, například na anglickou verzi.

Pro opravu daných problémů a změnu strategie načítání překladů – aby je posílal server spolu s vyrenderovanou stránkou, byla použita knihovna `next-i18next` [27]. Tato knihovna zapouzdřuje nezbytnou logiku pro SSR. Funkce pro načítání překladů na straně serveru vypadá následovně – viz výpis kódu 4.13. Přijímá parametry namespaces, které udávají, jaké soubory s překlady jsou potřeba pro překlad této stránky.

```
export default function withTranslations(
    namespacesRequired?: string[]
): GetServerSideProps {
    return async (context) => {
        const locale = context.locale ?? "cs"

        const translations = await serverSideTranslations(
            locale,
            namespacesRequired
        )

        return {
            props: {
                ...translations,
            },
        }
    }
}
```

Výpis kódu 4.13 Funkce pro načítání překladů na straně serveru s využitím knihovny `next-i18next`

Tímto způsobem bylo zajištěno, že všechny texty na stránce jsou přeloženy ještě před tím, než se stránka zobrazí uživateli, což vede k lepšímu uživatelskému zážitku a umožňuje sdílení odkazů na stránky ve specifických jazykových verzích.

Kapitola 5
Testování

5.1 Implementace API testů

Struktura testů byla založena na rozdělení všech testů do čtyř hlavních kategorií:

1. Testy ověřující celkovou autentizaci a autorizaci API.
2. Testy ověřující správnost endpointů pro předměty.
3. Testy ověřující správnost endpointů pro semestry.
4. Testy ověřující správnost endpointů pro učitele.

Každá kategorie poté obsahovala testovací scénáře, kde každý scénář odpovídal jednomu endpointu. Pro každý scénář bylo navrženo a vytvořeno několik testovacích případů, zaměřených na testování různých aspektů daného scénáře (endpointu). Například jeden testovací případ pro pozitivní případ, kdy je očekáván stavový kód 200, a druhý testovací případ pro negativní případ, tedy kdy je očekávána nějaká chyba v odpovědi.

Pro testování autentizace a autorizace API byl na backendu vytvořen *Koa* [28] middleware `authMockMiddleware`, viz ukázka kódu 5.1. Jeho účelem je simulovat autentizační FIT ČVUT server a taky KOS server na základě předem definovaných testovacích údajů, jako jsou falešní uživatelé s různými rolemi. Pro mockování těchto serverů byla použita knihovna *nock* [29].

```
const authMockMiddleware = async (
    ctx: Context,
    next: Next
): Promise<void> => {
    const token = ctx.headers.authorization?.replace("Bearer ", "") || ""

    if (!tokenToMockUser.has(token)) {
        await next()
        return
    }

    const mockUser = tokenToMockUser.get(token)

    mockTokenValidation(token, { user_name: mockUser.username })
    mockKosApiUserFetch(mockUser.username, token, mockUser)

    await next()
    nock.cleanAll()
}
```

Výpis kódu 5.1 Middleware `authMockMiddleware` pro mockování autentizace a autorizace

Tento přístup umožnil podrobně testovat chování API v různých scénářích autentizace a autorizace. Testovací případy zaměřené na autentizaci se soustředily na ověření správné reakce serveru v situacích, kdy uživatel nebyl autentizován, kdy měl neplatný token, nebo kdy mu chyběla potřebná role. Hlavně se testovaly statusové kódy odpovědí, které API vracelo, tedy 200, 401, 403.

Další kategorie testů, týkající se endpointů pro předměty, semestry a učitele, byly prováděny s platnými autentizačními údaji. Byly implementovány jak jednoduché testy, které kontrolovaly základní funkčnost endpointů, tak i složitější testy zahrnující více kroků a kombinací různých endpointů.

Příklad takového jednoduchého testovacího případu je ověření správnosti endpointu, který vrací všechny semestry. Bylo nutné otestovat, že odpověď má statusový kód 200 a vrací neprázdné pole. Níže je ukázka kódu 5.2 automatizovaného Postman testu.

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response should be a non-empty array", function () {
    const body = pm.response.json();
    pm.expect(Array.isArray(body)).to.be.true;
    pm.expect(body.length).to.be.above(0);
});
```

Výpis kódu 5.2 Automatizovaný Postman test pro ověření, že endpoint pro semestry vrací statusový kód 200 a neprázdné pole

Příklad komplexního testovacího případu zahrnuje testování endpointu pro přiřazení učitele k předmětu. Tento testovací případ lze popsat následujícím způsobem:

1. Zavolat `PUT /semester/:semesterCode/course/:courseCode/teacher` s prázdným tělem

pro odstranění všech učitelů předmětu.

2. Zavolat `PUT /semester/:semesterCode/course/:courseCode/lecture` pro nastavení jedné paralelky přednášky pro tento předmět.
3. Zavolat `GET /semester/:semesterCode/course/:courseCode` pro ověření, že předmět skutečně nemá žádné učitele a má jednu paralelku přednášky.
4. Zavolat `POST /semester/:semesterCode/course/:courseCode/teacher/:username` pro samotné přidání učitele k předmětu. V těle požadavku specifikovat, jak velký učitel bude mít úvazek v dané paralelce v hodinách.
5. Zavolat `GET /semester/:semesterCode/course/:courseCode` pro ověření, že nyní má předmět přiřazeného jednoho učitele, který byl přidán v předchozím požadavku. Dále zkontrolovat, že se u předmětu snížil počet zbývajících hodin pro pokrytí.

Celkem bylo vytvořeno 35 automatizovaných testovacích případů, které pokrývají veškeré aspekty REST API backendu, od autentizace přes jednoduchou kontrolu správnosti statusových kódů až po složité případy zahrnující práci s několika požadavky. Po spuštění všech testů bylo detekováno několik chyb, a byl vytvořen *bug report*. Celkově se chyby primárně týkaly jen statusových kódů a nebyly kritické. Celý proces testování volání API backendu byl zahrnut do CI pipeline na GitLabu, což se spouští při každém commitu.

5.2 Provedení uživatelského testováni

Pro provedení uživatelského testování byly vytvořeny testovací scénáře vedoucím této práce, Ing. Michalem Valentou, Ph.D. Tyto scénáře se zaměřovaly na ověření hlavních případů užití aplikace. Následně byla aplikace zpřístupněna úvazkářům, kteří měli za úkol vyplnit tyto scénáře a zaznamenat své dojmy. Celkem se testování zúčastnilo sedm úvazkářů. Někteří z nich již měli zkušenosti se starou aplikací. Seznam jednotlivých scénářů spolu s jejich průměrným hodnocením těžkosti lze najít v příloze této práce.

Mezi nejčastější připomínky patří:

- Chybějící odkazy na jednotlivé předměty a učitele v jejích názvech v tabulkách, což nutí k ručnímu vyhledávání.
- Některé tabulky, které obsahují mnoho sloupců, jsou v UI úzké, což komplikuje práci a nutí používat horizontální posuvník.

Tyto připomínky byly po uživatelském testování implementovány. Byly však další důležité připomínky, které se ukázaly jako složitější k implementaci. Tyto změny by vyžadovaly rozsáhlé úpravy mnoha částí aplikace. Patří mezi ně:

- Filtrování tabulek by nemělo být jen jednorázové, ale trvalé, ukládané v rámci session nebo v lokálním úložišti prohlížeče.
- Implementace pohledu na úvazky učitelů za dva semestry.

Kapitola 6

Analýza upgradu Next.js

Hlavním cílem této kapitoly je provést analýzu aktualizace frontendového frameworku Next.js, definovat pojem technického dluhu a vysvětlit, jak souvisí s aktualizací technologií.

6.1 Technický dluh

Technický dluh je koncept v oblasti softwarového vývoje, který odkazuje na dodatečné náklady spojené s pozdějšími úpravami kódu, který nebyl původně napsán s ohledem na nejlepší možná řešení, ale byl implementován rychle a jednoduše s cílem dosáhnout krátkodobých cílů. Podobně jako finanční dluh, i technický dluh generuje „úroky“, což znamená, že čím déle odložíme jeho splacení, tím náročnější a nákladnější se stává údržba a rozvoj kódové báze.

Aktualizace frameworku je jedním ze způsobů, jak řešit technický dluh. Frameworky a knihovny, na kterých jsou aplikace založené, se neustále vyvíjejí, a každá nová verze přináší lepší výkon, nové funkce a opravy chyb, které mohou přispět ke snížení technického dluhu. Ignorování aktualizací může vést k tomu, že aplikace zůstane závislá na zastaralých technologiích, což zvyšuje technický dluh a může vést k bezpečnostním rizikům.

Jak technologie na straně backendu, tak technologie frontendu podléhají postupnému zastarávání. Obecně se však uznává, že technologie na straně frontendu zastarávají rychleji než technologie backendu, což je dáno rychlým vývojem nových standardů, frameworků a nástrojů v oblasti webového inženýrství a uživatelských rozhraní.

V následující části bude provedena analýza upgradu frontendového frameworku Next.js a zhodnocení, jaký dopad by měl upgrade na aplikaci Úvazkostroj, jaké přináší výhody a zda tento upgrade pomůže snížit celkový technický dluh systému.

6.2 Analýza upgradu Next.js na verzi 13

Dne 25. října 2022 byla na konferenci Next.js společností Vercel představena [30] nová verze jejich frameworku – Next.js 13. Hlavním cílem nové verze bylo urychlit a zjednodušit vývojový proces a zvýšit výkon aplikací prostřednictvím lepšího rozdělení kódu, ukládání do mezipaměti a optimalizace načítání zdrojů.

V této sekci bude provedena analýza migrace frontendového frameworku Next.js z předchozí verze 12, která byla používána v aplikaci, na novější verzi Next.js 13, která se od ní výrazně liší.

6.2.1 Serverové a klientské komponenty

Pro pochopení této části je nezbytný dobrý přehled o rozdílech mezi SSG, SSR a CSR, které byly probírány v teoretické části této práce v sekci o technologiích renderování webových stránek.

6.2.1.1 O serverových a klientských komponentách

Serverová komponenta je komponenta, která se načítá a vykresluje na serveru, zatímco klientská komponenta se načítá a vykresluje na straně klienta (v prohlížeči). Ve verzi Next.js 12 jsou všechny komponenty považovány za klientské a mohou tak interagovat s API prohlížeče. Ačkoli je možné rozšířit tyto klientské komponenty o serverovou logiku pomocí funkce `getServerSideProps`, problémem ale je , že primární renderování stránky ve verzi 12 probíhá na straně klienta, bez ohledu na serverový pre-rendering. Toto je způsobeno tím, že všechny komponenty jsou klientské, a proto nelze určit, zda již byly zcela vyrenderovány serverem.

Next.js 13 řeší tento problém zavedením serverových komponent. Serverová komponenta je načtena a kompletně vyrenderována na serveru, což značně snižuje velikost odesílaného JS balíčku a zlepšuje celkový výkon. Serverové komponenty nepodporují stav, hooky, API prohlížeče a ostatní klientské nástroje, protože vše je prováděno na serveru.

Hlavní charakteristiky a rozdíly serverových a klientských komponent jsou následující:

- **Serverové komponenty** – jsou vhodné pro načítání dat z API nebo databází. Mají tu výhodu, že jsou obvykle rychlejší než načítání na straně klienta, protože server je zpravidla blíže ke zdroji dat než prohlížeč. Na rozdíl od klientské komponenty, která může být obohacena o `getServerSideProps`, serverová komponenta umožňuje přímé využívání asynchronních funkcí JavaScriptu, jako jsou `await` a `async`. Serverová komponenta se také dobře cachuje, což nabízí mnoho možností pro optimalizaci, a je hlavním místem pro manipulaci s citlivými daty, jako jsou tokeny a API klíče. Díky funkci streamování je možné rozdělit renderování na menší části a postupně je odesílat klientovi, jakmile jsou připraveny. Uživatelé tak mohou vidět části stránky dříve, aniž by museli čekat na kompletní vyrenderování stránky serverem.
- **Klientské komponenty** – jsou vhodné pro situace, kdy se očekává interakce s uživatelem nebo když je potřeba využívat API prohlížeče. Jak ve verzi 12, tak i v nové verzi 13, jsou tyto komponenty pre-renderovany na serveru, což přispívá k lepšímu výkonu aplikací a zlepšuje uživatelskou zkušenost tím, že minimalizuje čas potřebný k zobrazení obsahu na straně klienta.

6.2.1.2 Shrnuti

Celkově by zavedení hlavně serverových komponent systému prospělo, jelikož by se tím jasně oddělila serverová logika od klientské. Serverové komponenty by definitivně zvýšily celkovou rychlost aplikace, jelikož prvotní načítání dat a celkový rendering stránky by probíhal na serveru, a uživatel by tak mohl vidět vykreslené stránky rychleji. Například serverová komponenta by mohla být stránka reprezentující seznam učitelů. Prvotní načítání učitelů by probíhalo na serveru, a uživatel by tak mohl vidět vyrenderovanou stránku s učiteli dříve. Naopak komponenta jako tabulka by byla klientská komponenta, protože s ní uživatel aktivně interaguje.

6.2.2 Routing

Routing je klíčovým aspektem webových aplikací, neboť řídí způsob navigace v aplikaci a zpracování různých cest (URL adres) uživatelem. *Router* je obvykle modul v rámci webové aplikace, který implementuje logiku specifického mechanismu routování [1]. Funguje jako „řídící středisko“,

[1] směrování

které určuje, jaká stránka nebo komponenta se uživateli zobrazí, když navštíví určité URL. Next.js 13 přinesl inovativní mechanismus routování nazývaný „App Routing“, který poskytuje větší flexibilitu a kontrolu oproti tradičnímu routování prostřednictvím „Pages Routing“. V této části jsou rozebrány obě technologie, jejich rozdíly, výhody a nedostatky. Nakonec je vyhodnoceno, zda by přechod na „App Routing“ představoval výhodu pro současný systém.

6.2.2.1 Pages routing

Next.js 12 využívá systém routování známý jako „Pages routing“, který se opírá o princip směrování podle souborového systému. Tento přístup výrazně usnadňuje vytváření webových stránek – stačí přidat soubor do adresáře pojmenovaného `pages/`. Page router následně automaticky generuje URL adresy na základě názvů těchto souborů a jejich umístění v adresářové struktuře projektu. Tento způsob směrování je vysoce intuitivní a přímý. Všechny komponenty umístěné v `pages/` jsou klientské komponenty. Mezi hlavní výhody page router patří jeho jednoduchost, upřednostnění konvencí před konfigurací a možnost definování statických i dynamických cest. Současný systém využívá následující adresářovou strukturu, která je založena právě na „Pages routing“, je vidět zleva na obrázku 6.1.

6.2.2.2 App routing

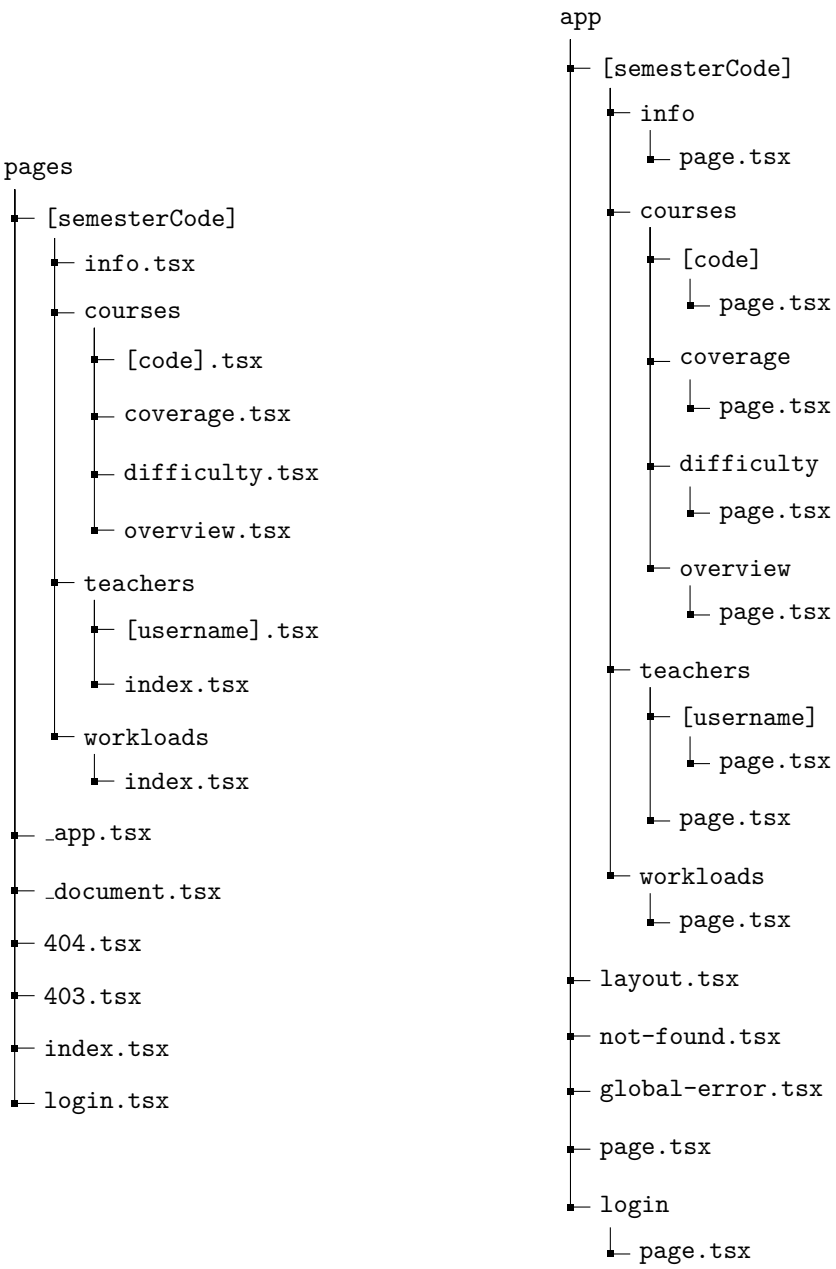
App router byl představen jako alternativa k Pages routeru, jež je zaměřena především na složitější případy routování, kde je flexibilita klíčová. V jistém smyslu App router rozšiřuje základní funkce Pages routeru, přičemž zachovává směrování podle souborového systému a vyžaduje umístění webových stránkových komponent do adresáře `app/`. Prvním podstatným rozdílem oproti Pages routeru je, že všechny stránkové komponenty jsou serverové komponenty, čímž využívají všechny výhody popsané v předchozí sekci. Druhým rozdílem je, že App router používá zcela odlišné pojmenování souborů pro směrování. V App routeru bylo definováno velké množství názvů souborů se speciálním významem, jako jsou `loading.tsx`, `error.tsx`, `layout.tsx`, `template.tsx`, které právě přinášejí flexibilitu v definování a konfiguraci uživatelského rozhraní. Pokud by systém používal App router místo Pages routeru, adresářová struktura stránek by mohla vypadat jako na obrázku 6.1 vpravo. Je dobře vidět, že tato struktura stránek je velice komplikovaná a méně přehledná, než Pages router.

6.2.3 Shrnutí

App router je stále poměrně nová technologie, která zatím není široce podporována komunitou. Mnohé knihovny nejsou plně adaptovány, chybí podrobná dokumentace a stabilita, kterou má Pages Router. Ačkoliv App router přináší velkou flexibilitu pro pokročilé potřeby směrování, pro systém by toto zavedení nemuselo být přínosné. Naopak, implementace App routeru by mohla zbytečně zkomplikovat relativně jednoduchou aplikaci s intuitivním a přehledným současným Pages routerem.

Přesto bude dříve či později nutné přejít na App router, jelikož Pages Router se stane zastaralým. Jak doporučuje tým Next.js, nejlepší je postupovat podle principu *incremental adoption* [31], což znamená, že je možné postupně převádět části aplikace z Pages Routeru do nového App routeru při zachování funkčnosti Pages Routeru.

Celkově lze říci, že přechod na Next.js 13 by přinesl několik výhod, jako je intenzivnější využití serverové logiky díky serverovým komponentám a efektivnější načítání dat. Na druhou stranu by zavedení nového App Routeru projekt zbytečně zkomplikovalo. V závěru lze konstatovat, že přechod na Next.js 13 by nebyl příliš výhodný, a proto je vhodnější počkat na budoucí verze (14 a další) a posoudit jejich potenciální přínosy.



Obrázek 6.1 Porovnání adresářové struktury Page router (vlevo) a App router (vpravo)

Kapitola 7
Závěr

Tato práce se zabývala návrhem a implementací informačního systému pro hodnocení pedagogického výkonu *Úvazkostroj*. Byly pokryty jednotlivé fáze vývojového cyklu softwaru, které byly započaté volbou vhodné metodiky a analýzou stávajícího stavu. Dále byla provedena analýza nedostatků současného řešení a jejich odstranění, stejně tak návrh chybějících funkcionalit. Backend byl otestován pomoci API testů, a zároveň proběhlo uživatelské testování frontendové části.

Výsledná implementace obsahuje všechny požadované funkcionality a navíc poskytuje uživatelům přívětivější uživatelské rozhraní. Aplikace byla vylepšena jak z hlediska optimalizace, tak i z hlediska bezpečnosti. Všechny cíle byly splněny.

Na závěr byla provedena analýza možnosti upgradu frontendového frameworku Next.js na novější verzi, což pomůže při údržbě tohoto systému.

Příloha A

Výsledky uživatelského testování

Scenář	**Ohodnocení**[1]
Zjistěte přehled výuky vyučujícího Valenta v semestru B201.	1.8
P. Tvrdíkovi v semestru B191 přidejte jeden oponentský posudek dizertační práce a zjistěte o kolik se mu zvýšil počet ZH za příslušný semestr.	2
P. Tvrdíkovi přidejte výuku 13 hodin předměu BI-DBS v semestru B192, včetně odpovídajícího podílu na klasifikaci během semestru, protože nyní je pokryto více hodin cvičení než bylo vypsáno, odeberte přebývající hodiny vyučujícímu M. Valenta.	2.8
Zjistěte, které předměty katedry KTI v semestru B191 nejsou pokryté (mají nějaké ZH, které nejsou nikomu přidělené).	1.6
Zjistěte, kdo v semestru B202 dostal ZH za zkoušení v předmětu BI-AG2.	1.5

[1] V průměru, kde 1 = velmi lehké, 5 = velmi těžké

Bibliografie

1. CHAU, Nguyen Trong Chung. *Aplikace Úvazkostroj verze 4*. Praha, 2022. Diplomová práce. Fakulta informačních technologií ČVUT v Praze. Vedoucí práce Ing. Michal Valenta, PhD.
2. ROYCE, Winston W. Managing the Development of Large Software Systems. In: *Proceedings of IEEE WESCON*. Los Angeles, CA, 1970, s. 1–9. Presented in August 1970.
3. *Visualization of the Waterfall Model* [online]. 2021. [cit. 2024-03-24]. Dostupné z: `https://www.actitime.com/wp-content/uploads/2021/02/Waterfall-Model-01.png`. Image depicting the Waterfall model of software development.
4. BECK, Kent; BEEDLE, Mike; BENNEKUM, Arie van; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. *Manifesto for Agile Software Development* [online]. 2001. [cit. 2024-03-24]. Dostupné z: `http://agilemanifesto.org`.
5. SCHWABER, Ken; SUTHERLAND, Jeff. *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*. scrumguides.org, 2020. Dostupné také z: `https://scrumguides.org`.
6. *Illustration of the Scrum Framework* [online]. 2023. [cit. 2023-03-24]. Dostupné z: `https://www.scnsoft.com/blog-pictures/custom-software-development/6-scrum-.png`. Image illustrating the Scrum model for agile software development.
7. VERCEL, Inc. *SWR: React Hooks for Data Fetching* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://swr.vercel.app/`. Official SWR documentation.
8. VERCEL, Inc. *Next.js: The React Framework* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://nextjs.org`. Official Next.js website.
9. VERCEL, Inc. *Understanding Rendering in Next.js* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://nextjs.org/docs/pages/building-your-application/rendering`. Next.js Documentation.
10. *Illustration of Pre-rendering Techniques* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://excalidraw.com/#json=5470294399516672,DIV0EF7egE0KUDph8bQZUw`. Excalidraw diagram.
11. VERCEL, Inc. *Optimizing Performance in Next.js Applications* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://nextjs.org/docs/pages/building-your-application/optimizing`. Next.js Documentation.

12. REACT, Inc. *React.memo: Optimizing Functional Component Re-rendering* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://react.dev/reference/react/memo`. Official React documentation.
13. REACT, Inc. *React.lazy: Code Splitting with Suspense* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://react.dev/reference/react/lazy`. Official React documentation.
14. *List Virtualization: Enhancing Performance in Large Lists* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://www.patterns.dev/vanilla/virtual-lists/`. Patterns.dev tutorial.
15. *pgTAP: Unit Testing for PostgreSQL* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://pgtap.org/`. Official pgTAP website.
16. *pgBench: Benchmarking Tool for PostgreSQL* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://www.postgresql.org/docs/current/pgbench.html`. PostgreSQL Documentation.
17. *Postman: The Collaboration Platform for API Development* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://www.postman.com/`. Official Postman website.
18. *cURL: Command Line Tool and Library for Transferring Data with URLs* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://curl.se/`. Official cURL website.
19. *Insomnia: Simplify the API Design Process, From Design to Deployment* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://insomnia.rest/`. Official Insomnia website.
20. *Cypress: Fast, Easy and Reliable Testing for Anything That Runs in a Browser* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://www.cypress.io/`. Official Cypress website.
21. *Docker: Empowering App Development for Developers* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://www.docker.com/`. Official Docker website.
22. *Swagger: Simplify API Development for Users, Teams, and Enterprises* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://swagger.io/`. Official Swagger website.
23. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
24. *Dynamic Routes in Next.js* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://nextjs.org/docs/pages/building-your-application/routing/dynamic-routes`. Next.js Documentation.
25. *cookie: Basic HTTP cookie parser and serializer for HTTP servers* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://www.npmjs.com/package/cookie`. NPM package.
26. *js-cookie: A simple, lightweight JavaScript API for handling browser cookies* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://www.npmjs.com/package/js-cookie`. NPM package.
27. *next-i18next: Easy internationalization of your Next.js apps* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://github.com/i18next/next-i18next`. GitHub repository.
28. *Koa: Next generation web framework for Node.js* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://github.com/koajs/koa`. GitHub repository.
29. *Nock: HTTP Server Mocking and Expectations Library for Node.js* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://github.com/nock/nock`. GitHub repository.
30. *Introducing Next.js 13* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://nextjs.org/blog/next-13`. Official Next.js blog post.
31. *Incremental Adoption of Next.js' App Router* [online]. 2024. [cit. 2024-03-27]. Dostupné z: `https://nextjs.org/docs/app/building-your-application/upgrading/app-router-migration`. Next.js Documentation.

Obsah přiloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `frontend` ..... frontendová část
    - `backend` ..... backendová část
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF