Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Návrh a implementace škálovatelného systému pro vracení zboží v e-shopech s důrazem na moderní architekturu a technologie

Bakalářská práce

2025 David Schwam

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**David Schwam**
Osobní číslo:	**I21366**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Návrh a implementace škálovatelného systému pro vracení zboží v e--shopech s důrazem na moderní architekturu a technologie**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem práce je navrhnout a implementovat škálovatelný systém pro reklamace a vratky zboží, který využívá nejmodernější technologie a architektonické principy. Důraz bude kladen na efektivní zvládání velkého objemu transakcí, spolehlivost a rychlou odezvu systému. Pro dosažení stanovených cílů a výběrů technologií bude použita kombinace analýzy existujících řešení, moderních trendů, návrhu škálovatelné architektury, implementace systému a následného testování.

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

Seznam doporučené literatury:

KEBBANI, Nassim, Piotr TYLENDA a Russ MCKENDRICK, 2022. The Kubernetes bible: the definitive guide to deploying and managing Kubernetes across major cloud platforms. Birmingham: Packt. ISBN 978-183-8827-694.

KLEPPMANN, Martin, 2017. Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems. Beijing: O'Reilly. ISBN 978-144-9373-320.

Vedoucí bakalářské práce:	**Ing. Jan Panuš, Ph.D.** Katedra informačních technologií
Datum zadání bakalářské práce:	**15. prosince 2023**
Termín odevzdání bakalářské práce:	**10. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

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

V Pardubicích dne 28. února 2024

Prohlašuji:

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

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

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

V Pardubicích dne 24. 7. 2024

David Schwam

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce Ing. Janu Panušovi, Ph.D., za jeho přístup, konzultace a veškeré rady, které přispěly k vypracování této práce.

ANOTACE

Cílem práce je navrhnout a implementovat škálovatelný systém pro reklamace a vratky zboží, který využívá nejmodernější technologie a architektonické principy. Důraz bude kladen na efektivní zvládání velkého objemu transakcí, spolehlivost a rychlou odezvu systému. Pro dosažení stanovených cílů a výběrů technologií bude použita kombinace analýzy existujících řešení, moderních trendů, návrhu škálovatelné architektury, implementace systému a následného testování.

KLÍČOVÁ SLOVA

Webová aplikace, Architektura softwaru, Škálovatelnost, Výběr technologií, Vícekriteriální rozhodováni, Moderní technologie, Kubernetes, Testování zátěže

TITLE

Design and Implementation of a Scalable Return System for E-commerce with an emphasis on Modern Architecture and Technologies

ANNOTATION

The aim of this thesis is to design and implement a scalable system for handling product returns and complaints, utilizing the latest technologies and architectural principles. The focus will be on efficiently managing a high volume of transactions, ensuring reliability, and providing a rapid system response. To achieve the set goals and select appropriate technologies, a combination of analyzing existing solutions, modern trends, designing a scalable architecture, implementing the system, and subsequent testing will be used.

KEYWORDS

Web application, Software architecture, Scalability, Technology selection, multi-criteria decision making, Modern technologies, Kubernetes, Load testing

Obsah

Seznam ilustrací a tabulek 9
Úvod 10
Metodika práce 11
1 Analýza a požadavky systému 12
1.1Popis aplikace 12
1.1.1 Účel systému 12
1.1.2 Hlavní uživatelské scénáře 12
1.2 Požadavky na systém 14
1.2.1 Funkční požadavky 14
1.2.2 Nefunkční požadavky 15
2 Technologie a implementace 16
2.1 Teoretický základ 16
2.2 Výběr technologií 17
2.2.1 Moderní trendy ve vývoji frontendu 18
2.2.2 Výběr technologií pro frontend 20
2.2.2 Výběr dodatečných knihoven pro frontend 27
2.2.3 Moderní trendy ve vývoji backendu 27
2.2.4 Výběr technologií pro backend 28
2.2.5 Výběr databáze 35
2.2.6 Výběr dodatečných knihoven pro backend 35
2.3 Architektura aplikace 36
2.3.1 Klíčové komponenty systému 36
2.3.2 Plánování architektury 37
2.3.2 Strategie škálování 39
2.3.3 Návrh API 40
2.3.4 Návrh databázového schématu 42
2.4 Implementace 43
2.4.1 Implementace databáze 44
2.4.2 Implementace backendu 45
2.4.3 Implementace frontednu 46
2.4.4 implementace nasazení 48
3 Testování 49
3.1 Metodika testování 49
3.2 Výsledky testování 51

Závěr 52
Použitá literatura 53
Seznam příloh 57

Seznam ilustrací a tabulek

Tabulka 1: Matice párových porovnání kritérií pro výběr frontendového frameworku 22
Tabulka 2: Normalizována matice vah pro výběr frontendového frameworku 23
Tabulka 3:Hodnoty pro výběr frontendového frameworku 25
Tabulka 4: Matice párových porovnání kritérií pro výběr CSS frameworku 26
Tabulka 5: Váhy pro výběr CSS frameworku 26
Tabulka 6: Hodnoty pro výběr CSS frameworku 26
Tabulka 7: Matice párových porovnání kritérií pro výběr backendového jazyk 30
Tabulka 8: Váhy pro výběr backendového jazyk 30
Tabulka 9: Hodnoty pro výběr backendového jazyk 30
Tabulka 10: Výsledky výběru backendového jazyka 31
Tabulka 11: Matice párových porovnání kritérií pro výběr JavaScript runtime.......................32
Tabulka 12: Váhy pro výběr JavaScript runtime 32
Tabulka 13: Hodnoty pro výběr JavaScript runtime...33
Tabulka 14: Matice párových porovnání kritérií pro výběr backend frameworku 34
Tabulka 15: Váhy pro výběr backend frameworku 34
Tabulka 16: Data pro výběr backend frameworku 34
Tabulka 17: Výsledky testování 51

Obrázek 1: Graf základní struktury systému 38
Obrázek 2: Graf struktury systému s webovým serverem 39
Obrázek 3: Graf struktury systému s vyznačením podů 40

Úvod

Motivace k výběru tématu bakalářské práce vychází z mých předchozích zkušeností s vývojem softwarových aplikací. Jedním z hlavních problémů, se kterými jsem se při vývoji setkal, bylo množství dostupných technologií a nástrojů, ve kterých bylo obtížné se orientovat a zvolit ty nejvhodnější pro konkrétní projekt. Zároveň jsem si nebyl jistý, zda aplikace, kterou jsem vytvořil, dokáže bez ztráty výkonu a spolehlivosti obsloužit větší počet uživatelů.

Téma tedy souvisí s reálným projektem, na kterém jsem dříve pracoval, avšak jeho výběr pro bakalářskou práci byl do určité míry náhodný. Problematika škálovatelnosti a výběru vhodných technologií je však natolik obecná a aktuální, že nachází uplatnění prakticky v každém softwarovém projektu.

Cílem práce je prozkoumat a kriticky zhodnotit nejnovější trendy ve vývoji webových aplikací a posoudit jejich vhodnost pro reálné nasazení v prostředí s vyšší zátěží. Zvláštní důraz bude kladen na návrh architektury a způsob nasazení systému tak, aby byl od začátku koncipován s ohledem na škálovatelnost, rozšiřitelnost a efektivní správu v produkčním prostředí.

Čtenáři by si z práce měli odnést ucelený rámec pro vývoj webových aplikací – od výběru technologií, přes návrh architektury až po testování – který jim pomůže při řešení podobných výzev ve vlastních projektech. Praktickým výstupem bude funkční aplikace, která demonstruje popsané principy a postupy v praxi.

Metodika práce

Vývoj systému probíhal v několika vzájemně navazujících fázích. Prvním krokem byla analýza existujících řešení využívaných v českém e-commerce sektoru. Na základě zjištěných poznatků byly definovány uživatelské scénáře a specifikovány funkční i nefunkční požadavky na systém.

Ve druhé fázi následoval výběr vhodných technologií a návrh systémové architektury. Pro objektivní posouzení technologických alternativ byla využita technika pro porovnání podle více kritérií, která umožnila systematicky porovnat dostupná řešení na základě dat z průzkumů a technických benchmarků.

Samotný vývoj systému probíhal iterativně. Práce byla rozdělena do logických celků – frontend, backend a databázová vrstva – s postupnou implementací jednotlivých komponent.

Závěrečná fáze se zaměřila na testování a validaci výsledného řešení. Proběhlo zátěžové testování a analýza výkonnostních metrik pro posouzení splnění stanovených požadavků a připravenosti pro nasazení v produkčním prostředí.

1 Analýza a požadavky systému

1.1 Popis aplikace

1.1.1 Účel systému

Pro vytvoření systému je nejprve třeba jasně definovat, jaký je jeho účel.

Systém vratek bývá součástí kritické infrastruktury e-commerce platformy, kde každý výpadek nebo problém může znamenat přímý dopad [1] Proto je nezbytné navrhnout architekturu, která obstojí v reálném provozu s proměnlivou zátěží a potenciálně vysokým počtem souběžných požadavků.

V kontextu elektronického obchodování lze rozlišit dva základní typy požadavků na vrácení zboží, které systém musí obsloužit – vratky a reklamace. Vratka je situace, kdy zákazník vrací zboží, které zakoupil, avšak z různých subjektivních důvodů – například kvůli nesprávné velikosti, barvě nebo prostě proto, že si nákup rozmyslel. Zákazník má v tomto případě zpravidla právo zboží vrátit do 14 dnů od převzetí bez udání důvodu, jak stanoví zákon o ochraně spotřebitele [2]. Naproti tomu reklamace se týká zboží, které je vadné, poškozené nebo neodpovídá popisu, a představuje právní nárok zákazníka na opravu, výměnu nebo vrácení peněz. Na rozdíl od vratky není reklamace omezena lhůtou 14 dní a zákazník ji může uplatnit kdykoliv během záruční doby, která obvykle trvá 24 měsíců.

Proces vrácení zboží přirozeně zahrnuje dvě hlavní strany – zákazníka, který požadavek na vrácení nebo reklamaci iniciuje, a obchodníka, který tento požadavek přijímá, zpracovává a podle typu případu dále řeší. Účelem navrhovaného systému je tento proces v maximální možné míře digitalizovat, zjednodušit a zpřehlednit. Aplikace má zákazníkům umožnit pohodlně a efektivně podat žádost o vrácení nebo reklamaci zboží, zatímco obchodníkům nabídne nástroje pro přehlednou správu a rychlé vyřízení těchto požadavků.

1.1.2 Hlavní uživatelské scénáře

Byl specifikován účel aplikace, ale pro její úspěšnou implementaci je nutné podrobně definovat, jaké konkrétní funkce má aplikace nabízet. Pro dosažení tohoto, musíme projít a analyzovat uživatelské scénáře, které popisují běžné situace, ve kterých bude systém používán. Tento přístup umožní jednodušeji vyvodit nezbytné funkční požadavky a zajistit, že aplikace bude plně odpovídat potřebám uživatelů.

Pro získání relevantních scénářů byla provedena neformální analýza existujících řešení, přičemž byl kladen důraz zejména na funkce, které jsou v praxi běžně používané. Na základě průzkumu trhu byl zvolen jako hlavní referenční systém platforma Retino, která je široce rozšířená v českém e-commerce sektoru – podle dostupných údajů ji využívá více než tisíc e-shopů [5]. Tento systém navíc nabízí demo verzi, která umožnila detailní praktické seznámení s jeho funkcionalitou. Analýza systému Retino poskytla cenný vhled do očekávaného chování a možností podobné aplikace. Pomohla vyjasnit klíčové aspekty návrhu, identifikovat důležité funkce a získat inspiraci pro návrh vlastního řešení, které bude odpovídat standardům moderních e-commerce platforem.

Ze stanoveného účelu vyplývá, že aplikace zpracovávající požadavky na vrácení a reklamace by měla mít dva typy uživatelů: zákazníky a administrátory. To je patrné i na příkladu systému Retino. Průzkum scénářů byl proto rozdělen do dvou částí, které probíhaly následujícím způsobem:

Testování z pohledu zákazníka:

- Simulace procesu vrácení zboží od iniciace až po dokončení
- Testování různých scénářů (vrácení bez udání důvodu, reklamace vadného zboží)

Testování z pohledu administrátora:

- Přístup do administrátorského rozhraní demo verze
- Procházení jednotlivých funkcí pro správu vratek a reklamací

Výsledkem průzkumu jsou následující scénáře relevantní:

- *Scénář 1*: Iniciace vrácení zboží zákazníkem
  Zákazník, který si přeje vrátit zakoupené zboží, zadá číslo objednávky a e-mail, pod kterým byla objednávka vytvořena. V části "Výběr produktů" označí konkrétní položky k vrácení. Následně vyplní důvod vrácení a další požadované informace. Po odeslání žádosti systém automaticky vygeneruje štítek, který zákazník obdrží spolu s instrukcemi k odeslání zboží.
- *Scénář 2*: Sledování stavu vrácení zboží
  Po podání žádosti může zákazník sledovat stav vrácení. Systém poskytuje aktuální informace o průběhu zpracování: potvrzení o přijetí zásilky, kontrola jejího stavu a závěrečné potvrzení o vrácení peněz či výměně zboží.

- *Scénář 3*: Správa vrácení zboží administrátorem
  Administrátor e-shopu obdrží notifikaci o nové žádosti. Po přihlášení do administrátorského rozhraní má přehled o všech probíhajících případech. Po přijetí vráceného zboží provede kontrolu jeho stavu a aktualizuje záznam v systému. Zákazník je automaticky informován o aktuálním stavu i o konečném výsledku žádosti.

1.2 Požadavky na systém

Cílem specifikace požadavků je zachytit hlavní cíle systému z pohledu všech zúčastněných stran [3]. Požadavek můžeme definovat jako "specifikaci toho, co by mělo být implementováno". Existuje několik způsobů, jak požadavky kategorizovat, ale pro účely této práce je použito základní rozdělení do dvou kategorií:

- funkční požadavky – chování systému, tedy co by měl systém dělat.
- nefunkční požadavky – specifické vlastnosti nebo omezení systému

Důkladně a srozumitelně formulované požadavky významně snižují riziko, že výsledný systém nebude odpovídat očekáváním uživatelů. Jasná definice požadavků tedy přispívá k vyšší kvalitě vývoje a minimalizaci nedorozumění mezi zadavatelem a vývojovým týmem [4].

1.2.1 Funkční požadavky

Na základě provedené analýzy a uživatelských scénářů byly definovány klíčové funkční požadavky systému. Každému požadavku bylo přiřazeno označení ve formátu FPXX, kde XX představuje jedinečné identifikační číslo. Toto značení usnadňuje referenci na jednotlivé požadavky v dalších částech práce.

- **FP01** – Výběr produktu na základě objednávky
  Uživatel zadá číslo objednávky a e-mailovou adresu, na jejichž základě systém načte seznam produktů z dané objednávky. Uživatel si následně zvolí konkrétní produkt nebo produkty, které chce vrátit či reklamovat.
- **FP02** – Přehled požadavků pro správce
  Systém poskytne administrátorovi přístup k přehledu všech založených požadavků na vrácení a reklamaci. Tento přehled bude obsahovat základní informace jako číslo

objednávky, typ požadavku, datum vytvoření, aktuální stav a další relevantní údaje. Správce bude moci seznam filtrovat a třídit podle různých kritérií.

- **FP03** – Aktualizace stavu požadavku správcem
  Správce bude mít možnost měnit stav jednotlivých požadavků v průběhu jejich zpracování – např. z „Přijato" na „Zpracovává se", „Vyřešeno" apod. Každá změna bude zaznamenána v systému a uživatel o ní bude informován.

1.2.2 Nefunkční požadavky

Nefunkční požadavky jde definovat jako omezení kladené na systém. [3]

V rámci tohoto projektu hrají nefunkční požadavky klíčovou roli, jelikož nám výrazně ovlivní prostor pro návrh a implementaci.

Tyto požadavky mohou ovlivnit výběr konkrétních návrhových a implementačních řešení a zajistit, že systém bude splňovat určité kvalitativní a výkonové parametry. [4]

Na základě cílů této práce byly formulovány následující nefunkční požadavky:

- **NP01** – Použití moderních a populárních technologií
  Systém musí být postaven na aktuálních a široce používaných technologiích, které odpovídají současným standardům v oblasti softwarového vývoje. To zahrnuje využití moderních frameworků, knihoven a nástrojů, které podporují efektivitu, výkon a bezpečnost. Popularitu technologií budeme hodnotit na základě veřejně dostupných statistik a také na základě odpovědí z dotazníků.
- **NP02** – Výkon a odolnost vůči zátěži
  Systém musí být schopen obsloužit vysoký počet souběžných uživatelů a transakcí bez zásadního poklesu výkonu. Musí být optimalizován pro běžný i špičkový provoz, aby zůstával stabilní a spolehlivý. Splnění tohoto požadavku bude testováno pomocí zátěžových testů.
- **NP03** – Škálovatelnost systému
  Architektura systému musí umožňovat škálování v závislosti na rostoucích nárocích. To jak z výkonnostního hlediska, tak i z hlediska implementace nových funkcionalit.
- **NP04** – Technologie využité systémem musí být jednoduché
  Vzhledem k omezenému rozsahu této práce je kladen důraz na jednoduchost implementace i provozu. Vybrané technologie musí být srozumitelné, snadno

nasaditelné a intuitivní pro správu. Tento požadavek zahrnuje jak počáteční jednoduchost nasazení, tak i provozní jednoduchost – tedy snadnou správu, aktualizace a případné škálování. Míra jednoduchosti budu hodnotit na základě praktické zkušenosti s vybranými technologiemi a případně doplním o zpětnou vazbu z dotazníků.

2 Technologie a implementace

Předchozí kapitoly definovaly požadavky a funkční specifikace navrhovaného systému. Tato část se zaměří na jejich převedení do konkrétního technického řešení. Před samotným rozborem implementace považuji za důležité představit základní teoretické koncepty, které stojí za technologickými rozhodnutími. Tyto koncepty poskytují kontext a oporu pro volbu konkrétních nástrojů a architektury systému.

2.1 Teoretický základ

Webová aplikace je typ softwaru, který běží v internetovém prohlížeči a není třeba jej instalovat lokálně jako nativní aplikaci. Tento přístup umožňuje uživatelům přístup k aplikaci odkudkoliv, pokud mají připojení k internetu a kompatibilní prohlížeč. Princip fungování je založen na tom, že při přístupu na danou adresu si prohlížeč stáhne tzv. klientskou část aplikace ze serveru. [6] Komunikace mezi klientem a serverem probíhá pomocí protokolu http [7] (Hypertext Transfer Protocol) – sady pravidel pro přenos dat mezi zařízeními. Přijatý dokument bývá strukturován pomocí jazyka HTML (HyperText Markup Language), který definuje obsah a základní rozvržení stránky. Samotný HTML dokument však obvykle nestačí k vytvoření plnohodnotného uživatelského rozhraní, a proto se doplňuje dalšími dvěma klíčovými technologiemi. Tyto technologie jsou buď použity přímo jako součást html dokumentu, nebo jsou uložený v rozdílných souborech a je opět potřeba provést požadavek na server pro její získání.

První z klíčových technologií je CSS (Cascading Style Sheet). jazyk pro definici vzhledu. CSS umožňuje určit, jak mají jednotlivé prvky vypadat – jejich barvu, velikost, pozici na stránce – ale i složitější vlastnosti, jako jsou animace nebo responzivní zobrazení na různých zařízeních.

Druhou klíčovou technologií je JavaScript, historicky i současně primárním a nejrozšířenějším jazykem, který prohlížeče nativně podporují pro rozšiřování funkcionality webových stránek

[8]. Je to programovací jazyk, který dodává webovým stránkám dynamiku a interaktivitu. JavaScript běží přímo v prohlížeči uživatele a umožňuje reagovat na jeho akce, pracovat s lokálním úložištěm, komunikovat se serverem na pozadí a v neposlední řadě i měnit obsah stránky bez jejího znovunačtení. Tyto poslední dvě funkcionality umožnily vykreslování nového obsahu přímo na straně klienta – tzv. Client-Side Rendering (CSR). Tento nový paradigma umožnil vznik velice interaktivních a dynamických aplikací bez odezvy a představil nový způsob vykreslování stránek proti klasickému renderování na straně serveru (server-side rendering – SSR) [9, 15]. Toto nové vykreslovacím paradigma se stalo velice populární, avšak vznikly nové výzvy – například s rychlostí načítání stránky, která se mohla výrazně snížit kvůli velkému množství Javascriptu, jenž je potřeba stáhnout a interpretovat nebo fakt, že vyhledávače mají problém tento JavaScript interpretovat [9]. Postupně se částečně začal vracet i přístup renderování na serveru a vznikaly různé nové hybridní strategie vykreslování podle potřeb konkrétní aplikace. [10, 15]

Zatímco klientská část aplikace se stará o interaktivitu a uživatelské rozhraní, serverová část je zodpovědná za poskytování obsahu, přístup k databázím, ověřování uživatelů a zpracování obchodní logiky. Server poskytuje nejen HTML, CSS a JavaScript soubory, ale také zajišťuje provádění operací, které nelze efektivně nebo bezpečně realizovat v prohlížeči. Z architektonického hlediska je běžné rozlišovat dvě serverové vrstvy – aplikační server a jeho podmnožinu webový server. Webový server zajišťuje směrování požadavků a statického obsahu (např. HTML, CSS, JavaScript), zatímco aplikační server zpracovává aplikační logiku a komunikuje s databázemi. V praxi bývají tyto dvě vrstvy někdy spojeny v jednom procesu, zvláště u jednodušších aplikací nebo při použití moderních vývojových platforem, které integrují obě role.[12] Volba konkrétní architektury závisí zejména na požadavcích.

2.2 Výběr technologií

Cílem této kapitoly je provést podrobnou analýzu aktuálních technologických trendů a na jejím základě zdůvodnit výběr nástrojů použitých při vývoji frontendu i backendu. Nejde však pouze o představení jednotlivých technologií. Tato kapitola má také ukázat rozhodovací proces, který k jejich výběru vedl, a poskytnout jasnou argumentaci, jak každé z těchto rozhodnutí přispívá ke splnění konkrétních požadavků systému. Navazuje se tím na předchozí části práce, ve kterých byly identifikovány klíčové funkční a nefunkční požadavky a hlavní scénáře použití.

Tyto poznatky nyní slouží jako základ pro informovaná a odůvodněná rozhodnutí o architektuře a použitých technologiích.

2.2.1 Moderní trendy ve vývoji frontendu

Při vývoji frontendu aplikace, existují v zásadě dvě základní možnosti v závislosti na přístupu vykreslení obsahu [17]. První variantou je generování obsahu přímo na straně klienta, tedy v prohlížeči. Jak již bylo zmíněno, primárním nástrojem pro tento přístup je jazyk JavaScript, který umožňuje vytvářet nový obsah a tím vytvářet interaktivní, dynamické uživatelské rozhraní [8]. Při vývoji větších systémů se často ukazuje, že ruční tvorba vlastních abstrakcí může být časově náročná a obtížně udržitelná. Aby se těmto problémům předešlo, vznikla řada frontendových frameworků, které vývojářům poskytují předdefinovanou strukturu aplikace, osvědčené vývojové postupy a množství integrovaných nástrojů. Díky tomu se mohou soustředit na vývoj samotné obchodní logiky a vizuální podoby aplikace, aniž by museli opakovaně řešit nízko-úrovňové technické detaily. Tento přístup, který se v posledních letech stal de facto průmyslovým standardem, výrazně urychluje vývoj, zvyšuje kvalitu výsledného kódu a podporuje dlouhodobou udržitelnost i škálovatelnost aplikací [13]. Současné frameworky podporují vývoj v duchu komponentového modelu, kdy je obchodní logika a uživatelské rozhraní rozděleno do modulárních, opakovaně použitelných celků [14]. Tento přístup nejen zvyšuje přehlednost a znuvupoužitelnost kódu, ale ve spojení s moderními nástroji pro sestavování aplikací výrazně zjednodušuje celý proces tvorby frontendové části systému.

Druhým přístupem k předávání obsahu uživateli je jeho generování na straně serveru a následné odesílání klientovi, případně využití hybridních technik. Tradičně byl velmi rozšířený model Multi-Page Applications (MPA), který je spojen například s technologiemi jako PHP, Ruby on Rails, ASP.NET nebo JSP. Tyto aplikace generovaly kompletní HTML stránku pro každou uživatelskou interakci, což znamenalo opakované načítání celé stránky a vedlo k méně plynulému a interaktivnímu uživatelskému zážitku. Ačkoliv byl tento model ve své době robustní a stabilní, přestal vyhovovat moderním nárokům na dynamiku a rychlou odezvu webových aplikací [15]. V posledních letech se však server-side rendering (SSR) opět dostává do popředí zájmu. Důvodem je snaha skloubit výhody klasických MPA – zejména plně renderovaného HTML, optimalizace pro vyhledávače (SEO) a dobré fungování i na slabších zařízeních – s výhodami moderních Single-Page Applications (SPA), které nabízejí vysokou míru interaktivity [9, [10]. Tyto přístupy jsou často postaveny na Javascriptu a dobře se integrují

s existujícími klientskými frameworky. Výsledkem jsou tzv. meta-frameworky, jako je například Next.js pro React nebo Nuxt.js pro Vue, které umožňují hladkou spolupráci mezi serverem a klientem a podporují různé strategie vykreslování (např. CSR, SSR, SSG nebo ISR) [9]. Rostoucí popularitu si v současnosti získává i lehký framework HTMX, který umožňuje budování interaktivních uživatelských rozhraní bez nutnosti psát rozsáhlý JavaScript kód [18].

Jak již bylo zmíněno, každá renderování technika má své výhody i nevýhody, a je proto nutné zvolit tu, která nejlépe odpovídá specifickým požadavkům mé aplikace. Vzhledem k tomu, že mezi těmito požadavky není SEO ani okamžitá odezva, ale je podle požadavků kladen důraz na co nejjednodušší vývoj, dává smysl, že nejvhodnější volbou je SPA – nyní zbývá už jen vybrat konkrétní framework.

Průzkum "State of JS", byl vytvořen, aby identifikoval nadcházející trendy v ekosystému vývoje webu a tím pomohl vývojářům činit technologická rozhodnutí.[19] Od průzkumu máme několik metrik, podle kterých si můžeme vybírat technologie:

- Použití (Usage) – Podíl respondentů, kteří položku použili
- Povědomí (Awareness) – Podíl respondentů, kteří o položce slyšeli nebo ji použili
- Zájem (Interest) – Podíl pozitivního sentimentu mezi respondenty, kteří o položce slyšeli
- Udržení (Retention) – Podíl pozitivního sentimentu mezi respondenty, kteří položku použili
- Pozitivita (Posititvity) – Podíl pozitivního sentimentu mezi respondenty, kteří vyjádřili sentiment

Tyto metriky je možné dále filtrovat podle demografických a jiných faktorů, jako například velikost týmu, ve kterém ji daný člověk používá.

Pro doplnění rozhodovacího procesu byl vybrán také průzkum „Rising Stars JS 2024“, který přináší další důležitou metriku – popularitu měřenou počtem hvězdiček na GitHubu za posledních 12 měsíců.[20]

Poslední metrika, která byla použitá při rozhodování byl benchmark výkonu mezi frontendovými frameworky [26].

Zde je seznam technologií, ze kterého bude prováděn výběr – průřez čtyř dostupných technologií přes všechny dotazníky a benchmarky: React, Angular, Vue.js, Svelte, HTMX

Aby byla zajištěna spokojenost uživatelů a intuitivní ovládání aplikace, je třeba zvážit i způsob stylování. To zásadně ovlivňuje vizuální přitažlivost i přehlednost rozhraní, usnadňuje orientaci a zvyšuje uživatelský komfort. Stejně jako v JavaScript ekosystému existují pro CSS frameworky, které urychlují vývoj a zajišťují jednotný vzhled napříč aplikací. Kromě toho lze využít preprocesory jako SASS nebo LESS, jež rozšiřují čisté CSS o proměnné, mixiny či funkce, a umožňují tak psát čitelnější a udržitelnější kód i ve velkých projektech.[22]

Podobně jako průzkum „State of JS“ pomáhá sledovat trendy v Javascriptu, data a metriky z „State of CSS“ poskytnou přehled o tom, které nástroje a techniky jsou v oblasti stylování nejvíce využívané a doporučené [21]. Zde je výčet technologii: Tailwind, Boostrap, Materialize

2.2.2 Výběr technologií pro frontend

Nyní je k dispozici vymezený seznam technologií, ze kterých bude probíhat výběr, a je možné teď přistoupit k samotnému rozhodování. V praxi by volba často vycházela ze subjektivních preferencí, zkušeností či dalších faktorů. V této práci bude zvolen strukturovaný přístup, který umožní najít teoreticky optimální řešení s ohledem na definované požadavky a dostupná data. Jednou z metod, jež systematizují složitá rozhodnutí, je vícekriteriální analýza variant (Multiple Criteria Decision Analysis, MCDA). MCDA pomáhá hodnotit alternativy podle více, často vzájemně protichůdných kritérií a rozkládá tak rozhodovací proces do přehledné hierarchie cílů, kritérií a alternativ [23] [24]. Tento postup zajišťuje transparentní a opakovatelnou evaluaci a usnadňuje výběr nejvhodnější varianty na základě explicitně definovaných kritérií a jejich vah [25].

Pro účely této práce bylo rozhodnuto aplikovat metodu Analytic Hierarchy Process (AHP), která patří mezi nejrozšířenější techniky MCDA. AHP strukturalizuje rozhodovací problém do hierarchie a využívá párových porovnání pro stanovení vah kritérií a následně i ohodnocení alternativ vzhledem k těmto kritériím. Rozhodovatelé provádějí párová porovnání kritérií mezi sebou a alternativ vzhledem k jednotlivým kritériím pomocí standardizované stupnice, typicky Saatyho devítibodové škály [25]. Výsledkem je soubor číselných vah vyjadřujících relativní důležitost jednotlivých kritérií. Tyto váhy se následně agregují s výkonnostními skóre alternativ, čímž se vypočítá celkové skóre každé varianty. Proces navíc zahrnuje kontroly konzistence, které ověřují logickou provázanost porovnání a zvyšují tak spolehlivost i transparentnost rozhodování [23, 24, 25].

Při aplikaci metody AHP je nezbytné jasně definovat kritéria a stanovit jejich váhy. Vzhledem k tomu, že cílem je vybrat technologii na základě předem definovaných nefunkčních požadavků (NP01-NP04), ale primárně dostupné kvantitativní a srovnávací data přímo neodpovídají abstraktní formulaci požadavků, bylo nutné provést mapování těchto požadavků na kvantifikovatelná a měřitelná kritéria. Tato kritéria byla zvolena tak, aby co nejlépe reprezentovala a postihovala podstatu daných požadavků na základě dostupných datových zdrojů. Níže je uvedeno mapování požadavků na konkrétní kritéria a zdůvodnění volby jednotlivých metrik:

- **NP01** - "Použití moderních a populárních technologií": Kvantitativně bylo toto kritérium hodnoceno na základě dat o popularitě a míře adopce technologií z dotazníkového šetření "State of JS 2024"[19]. Konkrétně byly využity metriky Zájem (C1), Použití (C2), Udržení (C3) a Pozitivita (C4), které přímo indikují postoj vývojářské komunity k dané technologii. Dále byl zařazen počet hvězdiček na GitHubu (C5), získaný z průzkumu "Rising Stars JS 2024"[20], sloužící jako indikátor velikosti a aktivity komunitní podpory a rozšíření dané technologie. Ačkoli GitHub hvězdy nejsou přímou metrikou modernosti nebo popularity, jejich dynamika a absolutní počet dobře korelují se zájmem komunity a aktivním vývojem ekosystému.
- **NP02** - "Výkon a odolnost vůči zátěži": Zatímco primární odolnost systému vůči zátěži je dominantně ovlivněna backendovou architekturou a infrastrukturou, výkon frontendové aplikace v prohlížeči pod zátěží velkého objemu dat a komplexního vykreslování má přímý dopad na uživatelskou zkušenost a vnímanou rychlost systému. Toto kritérium bylo hodnoceno na základě srovnávacích benchmarkových metrik benchmarkových dat pro JS frameworky v prohlížeči chrome verze 135[26]. Pro posouzení výkonu při vytváření jednotlivých prvků byla využita metrika "create rows" (C6), která měří čas potřebný k vykreslení pár malých řádků. Pro hodnocení výkonu při zpracování a manipulaci s větším množstvím existujících prvků slouží metrika "create many rows" (C7). Tyto metriky poskytují kvantitativní pohled na renderovací schopnosti frameworků za simulované zátěže.
- **NP03** - "Škálovatelnost systému": Škálovatelnost frontendového řešení je komplexní vlastnost, která závisí nejen na zvolené technologii, ale do značné míry i na architektonickém návrhu a dodržování správných vývojových postupů. Nicméně, některé charakteristiky technologie a jejího ekosystému mohou sloužit jako indikátory její vhodnosti pro škálovatelné aplikace a rostoucí vývojové týmy. Toto kritérium bylo

hodnoceno na základě nepřímých indikátorů. Konkrétně bylo posouzeno použití technologie v týmech o velikosti 101-1000 lidí (C8), získané z dotazníku "State of JS 2024"[19]. Vysoká míra adopce ve větších týmech signalizuje, že technologie je vhodná pro rozsáhlejší a komplexnější projekty s distribuovaným vývojem. Dále byl znovu zařazen počet hvězdiček na GitHubu (C5) jako nepřesný, ale relevantní indikátor velikosti a aktivity ekosystému a dostupnosti knihoven a nástrojů, které jsou pro škálovatelnost a dlouhodobý vývoj velkých aplikací důležité.

- **NP04** - "Technologie využité systémem musí být jednoduché": Jednoduchost technologie je v kontextu této práce vnímána především z pohledu snadnosti osvojení a efektivního použití pro běžné vývojové úkoly. Toto kritérium je klíčové pro rychlé zaškolení nových členů týmu a efektivitu vývoje, zejména v menších týmech na počátku projektu. Bude hodnoceno na základě použití technologie v malých týmech 1-5 lidí (C9), získaného z dotazníku "State of JS 2024"[19]. Předpokládá se, že technologie s vysokou mírou adopce a spokojenosti v menších týmech jsou obecně vnímány jako jednodušší na osvojení a práci s nimi.

Pro aplikaci metody AHP a stanovení vah jednotlivých kritérií (C1-C9) byla sestavena matice párových porovnání. Hodnocení relativního významu každého kritéria vůči ostatním bylo provedeno na základě vlastního subjektivního posouzení. Bylo použito Saatyho standardizované devítibodové škály, kde například hodnota 1 znamená stejnou důležitost a hodnota 9 znamená absolutní důležitost jednoho kritéria oproti druhému. Hodnota 1/9 pak značí absolutní podřadnost. Například, pokud je považován roční nárůst GitHub hvězd z průzkumu "Rising stars 2024"[20] (C5) za „výrazně důležitější" než úroveň povědomí (C1), je přiřazeno při porovnání C5 vůči C1 hodnotu 5 a opačný poměr (C1 vůči C5) hodnotu 1/5 [23, 25]. Tento postup se opakoval pro každý pár kritérií, čímž vznikla následující matice 9x9:

Tabulka 1: Matice párových porovnání kritérií pro výběr frontendového frameworku – Zdroj: vlastní

Kritéria	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**
C1: Zájem	1	1/2	1	2	3	5	7	1	1
C2: Použití	2	1	2	3	5	9	9	2	3
C3: Udržení	1	1/2	1	2	4	6	9	1	2
C4: Pozitivita	1/2	1/3	1/2	1	2	3	4	1	1
C5: GitHub hvězdy	1/3	1/5	1/4	1/2	1	2	2	1/2	1/2
C6: create rows	1/5	1/9	1/6	1/3	1/2	1	1	1/4	1/3
C7: cr. many rows	1/7	1/9	1/9	1/4	1/2	1	1	1/6	1/5

C8: Použití 101-1000	1	1/2	1	1	2	4	6	1	1
C9: Použití 1-5	1	1/3	1/2	1	2	3	5	1	1

Nedílnou součástí metody AHP je ověření konzistence provedených párových porovnání. Cílem je zkontrolovat, zda jsou naše úsudky vyjádřené v matici párových porovnání logicky soudržné a neobsahují významné rozpory. Pro tento účel se vypočítávají dva ukazatele: index konzistence (CI) a poměr konzistence (CR). Pokud zkusíme spočítat, zda je naše rozhodnutí konzistentní, nejprve z matice párových porovnání spočítáme největší vlastní hodnotu λ_{max} (eigenvalue). Je to číslo, které zhruba ukazuje, jak moc se matice odchyluje od dokonale logické (ideálně by se rovnalo počtu kritérií n). Tuto odchylku vyjádříme jako index konzistence CI = $(\lambda_{max} - n)/(n - 1)$. Abychom zjistili, zda je odchylka přijatelná, vydělíme CI takzvaným náhodným indexem RI (tabulková hodnota pro dané n, zde 1,45) a dostaneme poměr konzistence CR = CI/RI. Platí-li CR <0,10, jsou porovnání považována za konzistentní; v našem případě CR = 0,009, takže matice je v pořádku a není třeba ji upravovat. [23]

Jakmile je matice párových porovnání vyplněna, bude přistoupeno k normalizaci matice a výpočtu vah kritérií. Normalizace se provede tak, že každou hodnotu v matici vydělíme součtem sloupce, ve kterém se nachází. Součty sloupců jsou:

C1: 1 + 2 + 1 + 1/2 + 1/3 + 1/5 + 1/7 + 1 + 1 ≈ 7.176
C2: 1/2 + 1 + 1/2 + 1/3 + 1/5 + 1/9 + 1/9 + 1/2 + 1/3 ≈ 3.589
C3: 1 + 2 + 1 + 1/2 + 1/4 + 1/6 + 1/9 + 1 + 1/2 ≈ 6.528
C4: 2 + 3 + 2 + 1 + 1/2 + 1/3 + 1/4 + 1 + 1 ≈ 11.084
C5: 3 + 5 + 4 + 2 + 1 + 1/2 + 1/2 + 2 + 2 = 20
C6: 5 + 9 + 6 + 3 + 2 + 1 + 1 + 4 + 3 = 34
C7: 7 + 9 + 9 + 4 + 2 + 1 + 1 + 6 + 5 = 44
C8: 1 + 2 + 1 + 1 + 1/2 + 1/4 + 1/6 + 1 + 1 ≈ 7.917
C9: 1 + 3 + 2 + 1 + 1/2 + 1/3 + 1/5 + 1 + 1 ≈ 10.034

Normalizovaná matice je tedy následující:
Tabulka 2: Normalizována matice vah pro výběr frontendového frameworku – Zdroj: vlastní

Criteria	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**
C1: Zájem	0.139	0.139	0.153	0.180	0.150	0.147	0.159	0.126	0.100
C2: Použití	0.279	0.279	0.306	0.271	0.250	0.265	0.205	0.253	0.299
C3: Udržení	0.139	0.139	0.153	0.180	0.200	0.176	0.205	0.126	0.199
C4: Pozitivita	0.070	0.093	0.077	0.090	0.100	0.088	0.091	0.126	0.100
C5: GitHub hvězdy	0.046	0.056	0.038	0.045	0.050	0.059	0.045	0.063	0.050
C6: create rows	0.028	0.031	0.026	0.030	0.025	0.029	0.023	0.032	0.033
C7: cr. many rows	0.020	0.031	0.017	0.023	0.025	0.029	0.023	0.021	0.020
C8: Použití 101-1000	0.139	0.139	0.153	0.090	0.100	0.118	0.136	0.126	0.100
C9: Použití 1-5	0.139	0.093	0.077	0.090	0.100	0.088	0.114	0.126	0.100

Váhy jednotlivých kritérií (vektor priorit) získáme z normalizované matice průměrováním hodnot v každém řádku:
C1: 0.139 + 0.139 + 0.153 + 0.180 + 0.150 + 0.147 + 0.159 + 0.126 + 0.100 = 1.293 / 9 = 0.144 = 14.4%

C2: 0.279 + 0.279 + 0.306 + 0.271 + 0.250 + 0.265 + 0.205 + 0.253 + 0.299 = 2.208 / 9 = 0.245 = 24.5%
C3: 0.139 + 0.139 + 0.153 + 0.180 + 0.200 + 0.176 + 0.205 + 0.126 + 0.199 = 1.413 / 9 = 0.157 = 15.7%
C4: 0.070 + 0.093 + 0.077 + 0.090 + 0.100 + 0.088 + 0.091 + 0.126 + 0.100 = 0.843 / 9 = 0.094 = 9.4%
C5: 0.046 + 0.056 + 0.038 + 0.045 + 0.050 + 0.059 + 0.045 + 0.063 + 0.050 = 0.452 / 9 = 0.050 = 5.0%
C6: 0.028 + 0.031 + 0.026 + 0.030 + 0.025 + 0.029 + 0.023 + 0.032 + 0.033 = 0.247 / 9 = 0.027 = 2.7%
C7: 0.020 + 0.031 + 0.017 + 0.023 + 0.025 + 0.029 + 0.023 + 0.021 + 0.020 = 0.209 / 9 = 0.023 = 2.3%
C8: 0.139 + 0.139 + 0.153 + 0.090 + 0.100 + 0.118 + 0.136 + 0.126 + 0.100 = 1.095 / 9 = 0.122 = 12.2%
C9: 0.139 + 0.093 + 0.077 + 0.090 + 0.100 + 0.088 + 0.114 + 0.126 + 0.100 = 0.933 / 9 = 0.104 = 10.4%

Nyní, když máme stanoveny váhy kritérií, potřebujeme ohodnotit jednotlivé technologie vzhledem ke každému kritériu. Pro většinu kritérií (C1-C4, C8, C9) jsou hodnoty přímo získána z dotazníkových šetření a jsou již v relativním měřítku. Pro kritéria C5 (Github hvězdy), C6 (create rows) a C7 (create many rows) je potřeba naměřené nebo získané hodnoty normalizovat, aby byly srovnatelné s ostatními kritérii a s váhami. Normalizaci jsem provedl následovně:

- C5: GitHub hvězdy: Vyšší počet hvězd je lepší. Normalizace se provede vydělením počtu hvězd dané technologie maximálním počtem hvězd mezi všemi alternativami.
  React: 14,179 / 14,179 ≈ 1.000
  Vue.js: 5,917 / 14,179 ≈ 0.417
  Angular: 3,530 / 14,179 ≈ 0.249
  Svelte: 8,000 / 14,179 ≈ 0.564

- C6: create rows: Nižší hodnota (čas) je lepší. Normalizace se provede vydělením minimální hodnoty (nejlepšího času) mezi alternativami hodnotou dané technologie.
  Svelte mean: 25.5 (Minimum)
  Vue mean: 29.3
  React mean: 30.9
  Angular mean: 35.1
  Dostáváme tedy:
  Svelte: 25.5 / 25.5 = 1.00
  Vue: 25.5 / 29.3 ≈ 0.87
  React: 25.5 / 30.9 ≈ 0.83
  Angular: 25.5 / 35.1 ≈ 0.73

- C7: cr. many rows: Nižší hodnota (čas) je lepší. Normalizace se provede vydělením minimální hodnoty (nejlepšího času) mezi alternativami hodnotou dané technologie.

Svelte mean: 263.6 (Minimum)
Vue mean: 304.9
React mean: 463.7
Angular mean: 367.2
Dostáváme tedy:
Svelte (C7): 263.6 / 263.6 = 1.00
Vue (C7): 263.6 / 304.9 ≈0.86
React (C7): 263.6 / 463.7 ≈ 0.57
Angular (C7): 263.6 / 367.2 ≈ 0.72

Pro získání celkového skóre pro každou technologii (alternativu) se provede vážený součet normalizovaných hodnot alternativ pro každé kritérium a odpovídající váhy daného kritéria. Následující tabulka shrnuje normalizované hodnoty (skóre) pro každou technologii vůči jednotlivým kritériím:

Tabulka 3:Hodnoty pro výběr frontendového frameworku – Zdroj: vlastní

Criteria	React	Vue	Svelte	Angular
C1: Zájem	0.84	0.48	0.65	0.17
C2: Použití	0.61	0.51	0.26	0.50
C3: Udržení	0.93	0.87	0.88	0.54
C4: Pozitivita	0.84	0.42	0.40	0.23
C5: GitHub hvězdy	1.00	0.42	0.56	0.25
C6: create rows	0.83	0.87	1.00	0.73
C7: cr. many rows	0.57	0.86	1.00	0.72
C8: Použití 101-1000	0.86	0.48	0.27	0.55
C9: Použití 1-5	0.77	0.50	0.29	0.38

Celkové skóre pro jednotlivé technologie jsou:

React:(0.84×0.144)+(0.61×0.245)+(0.93×0.157)+(0.84×0.094)+(1.00×0.050)+(0.83×0.027)+(0.57×0.023)+(0.86×0.122)+(0.77×0.104) =
0.12096+0.14945+0.14581+0.07896+0.05000+0.02241+0.01311+0.10492+0.08008 ≈ 0.7657
Vue:
(0.48×0.144)+(0.51×0.245)+(0.87×0.157)+(0.42×0.094)+(0.42×0.050)+(0.87×0.027)+(0.86×0.023)+(0.48×0.122)+(0.50×0.104) =
0.06912+0.12495+0.13639+0.03948+0.02100+0.02349+0.01978+0.05856+0.05200 ≈0.5448
Svelte:
(0.65×0.144)+(0.26×0.245)+(0.88×0.157)+(0.40×0.094)+(0.56×0.050)+(1.00×0.027)+(1.00×0.023)+(0.27×0.122)+(0.29×0.104) =
0.09360+0.06370+0.13816+0.03760+0.02800+0.02700+0.02300+0.03294+0.03016 ≈ 0.4942
Angular:
(0.17×0.144)+(0.50×0.245)+(0.54×0.157)+(0.23×0.094)+(0.25×0.050)+(0.73×0.027)+(0.72×0.023)+(0.55×0.122)+(0.38×0.104) =
0.02448+0.12250+0.08478+0.02162+0.01250+0.01971+0.01656+0.06710+0.03952 ≈ 0.4088

Je zde tedy jasný výsledek, že na základě stanovených požadavků, definovaných kritérií, jejich vah odvozených z párového srovnání a dostupných dat o alternativách, je pro účely práce

metodou AHP identifikována jako nejlepší volba technologie **React**. Toto rozhodnutí je podloženo transparentním a strukturovaným procesem.

Dalším důležitým krokem je volba vhodného nástroje či frameworku pro stylování aplikace. Pro tento výběr aplikujeme podobné strukturované metody. Při definici kritérií pro hodnocení CSS frameworků se odchylujeme od těch pro JavaScript – výkon pro nás není určujícím faktorem. Místo toho se kritéria zaměří výhradně na aspekty jako popularita (NP01), jednoduchost použití (NP04) a škálovatelnost (NP03). Definuji tedy následující kritéria:

- Pro NP01 - "Použití moderních a populárních technologií": Dotazníku State of CSS 2024, poskytuje data o využití jednotlivých frameworků. (C1)
- Pro NP03 - "Škálovatelnost": Opět byly použity data z průzkumu "State of CSS 2024" zaměřené na týmy o velikosti 101-1000. (C2)
- Pro NP04 - "Jednoduchost použití": Inverzní přistup k NP03 budou využity data z průzkumu zaměřené na týmy o velikosti 1-5. (C3)

Zde je tabulka s párovým srovnáním alternativ pro jednotlivé kritéria:

Tabulka 4: Matice párových porovnání kritérií pro výběr CSS frameworku – Zdroj: vlastní

Criteria	C1	C2	C3
Usage celkem	1	1/3	1
Usage 101-1000	3	1	3
Usage 1-5	1	1/3	1

Tato matice má poměr konzistence CR = 0, což znamená, že je dokonale konzistentní. Váhy kritérií odvozené z této matice budou:

Tabulka 5: Váhy pro výběr CSS frameworku – Zdroj: vlastní

C1	C2	C3
0.16	0.50	0.33

Pokud se aplikují na tyto data:

Tabulka 6: Hodnoty pro výběr CSS frameworku – Zdroj: vlastní

Criteria	C1	C2	C3
TailwindCSS	0.75	0.76	0.74
Bootstrap	0.54	0.53	0.51
Materialize	0.13	0.14	0.10
Ant Design	0.13	0.16	0.08

Výsledek je jasný:

TailwindCSS: $(0.75\times0.167) +(0.76\times0.500) +(0.74\times0.333) =0.12525+0.38000+0.24642\approx0.752$
Bootstrap: $(0.54\times0.167) +(0.53\times0.500) +(0.51\times0.333) =0.09018+0.26500+0.16983\approx0.525$
Materialize: $(0.13\times0.167) +(0.14\times0.500) +(0.10\times0.333) =0.02171+0.07000+0.03330\approx0.125$
Ant Design: $(0.13\times0.167)+(0.16\times0.500) +(0.08\times0.333) =0.02171+0.08000+0.02664\approx0.128$
TailwindCSS je tedy nejvhodnějším frameworkem pro stylování aplikace.

2.2.2 Výběr dodatečných knihoven pro frontend

Knihovna React řeší především samotné vykreslování uživatelského rozhraní, většina reálných projektů nakonec musí řešit také problémy jako routování URL nebo získávání dat — oblasti, které React záměrně nepokrývá. Je sice možné vybudovat například router jen s nativními funkcemi Javascriptu a prohlížeče, avšak populární a aktivně vyvíjené frameworky k Reactu nabízejí doprovodné knihovny, které činí routování daleko intuitivnější součástí vývoje [31]. S ohledem na ekosystém Reactu byly vybrány následující knihovny:

- Pro zajištění routování na straně klienta byla vybrána knihovna React Router. Jedná se o standardní řešení v ekosystému Reactu [31], které díky své deklarativní povaze a efektivní správě historie prohlížeče zjednodušuje implementaci navigace (NP04).
- Specializované knihovny pro získávání dat přebírají veškerou práci spojenou s načítáním, cachováním a synchronizací – vývojář se tak může soustředit jen na to, jaká data aplikace potřebuje a jak je prezentovat. Typickým příkladem je React Query[31], dnes velmi populární a aktivně vyvíjená knihovna [20].
- Pro rychlý a efektivní vývoj uživatelského rozhraní byla zvolena kolekce komponent shadcn/ui, která je postavená na vybraném CSS frameworku Tailwind CSS. Ačkoli její výběr nebyl striktně nutný, představuje nedávno velmi populární přístup [20], který významně zjednodušuje a urychluje tvorbu UI (NP04). Tato efektivita je obzvláště důležitá v kontextu omezeného rozsahu této práce a jejího cíle demonstrovat funkční systém.
- V neposlední řadě je u moderního frontendového projektu je instalace build nástroje [45]. Tyto nástroje jsou nezbytné pro balíčkování a spouštění zdrojového kódu, poskytují vývojový server s funkcemi jako "hot module replacement" pro lokální vývoj a generují optimalizované sestavení aplikace pro nasazení na produkční server [45]. Pro účely tohoto projektu byl jako build nástroj vybrán Vite. Vite je navržen s cílem poskytnout rychlejší a úspornější vývojářskou zkušenost pro moderní webové projekty. Je názorově ukotvený ("opinionated") a přichází s rozumnými výchozími nastaveními hned po instalaci, což zjednodušuje jeho nastavení a použití [45].

2.2.3 Moderní trendy ve vývoji backendu

Nyní jsou jasně zvolené technologie pro frontendový vývoj. Aby však aplikace fungovala jako celek, je nezbytné věnovat stejnou pozornost i výběru technologií pro backendovou část

systému. Zatímco frontend tvoří vizuální a interaktivní vrstvu běžící přímo v prohlížeči uživatele, backend představuje kritickou součást softwarové architektury, která zajišťuje zpracování dat a business logiky [11] a poskytuje základnu, s níž frontendové uživatelské rozhraní komunikuje [7]. Tato klíčová komponenta tedy odpovídá za komplexní obchodní logiku, bezpečnou správu dat (ukládání i získávání z databází), autentizaci, autorizaci a efektivní komunikaci s frontendem i dalšími interními či externími službami prostřednictvím standardizovaných API, nejčastěji REST nebo GraphQL [6, 12]. Na rozdíl od frontendového vývoje, kde po léta hraje dominantní roli jazyk JavaScript a jeho ekosystém [8], vývoj backendu se vyznačuje enormní šíří a diverzitou dostupných technologií. Pro implementaci backendové logiky lze totiž vybírat z celé řady programovacích jazyků a k nim přidružených mnoha vyspělých frameworků [32]. Správná volba technologického stacku je totiž přelomová pro tvorbu škálovatelných a robustních webových platforem – výrazně určuje jejich výkon, škálovatelnost, udržovatelnost a celkový úspěch [33]. Technologie optimalizované na výkon navíc zvládnou více operací s menší spotřebou prostředků, což vede k rychlejší odezvě a lepší uživatelské zkušenosti [33]. V kontextu moderního webového vývoje tedy nejde jen o čistě technickou preferenci, ale o strategické rozhodnutí, které zásadně ovlivňuje jak samotný proces vývoje, tak klíčové vlastnosti aplikace po nasazení. Jednotlivé jazyky a frameworky se totiž výrazně liší v charakteristikách, jež jsou pro současné webové systémy rozhodující [33]. Aby toto rozhodnutí bylo opřeno o relevantní podklady a minimalizoval vliv subjektivních preferencí budou pro účely této práce využívána data z několika zdrojů:

- Stack Overflow Developer Survey 2024: Každoroční průzkum mezi vývojáři, zaměřený na popularitu programovacích jazyků, frameworků a technologických trendů v praxi. [27].
- JetBrains State of Developer Ecosystem Report 2024: Podrobný průzkum globálního vývojářského ekosystému s důrazem na používané technologie a pracovní návyky. [28]
- the-benchmarker/web-frameworks: Srovnávací test výkonu vybraných backendových frameworků pod simulovanou zátěží [29]
- Techempower web application frameworks benchmark: Porovnání výkonu mnoha webových aplikačních frameworků při provádění základních úloh, jako je serializace JSON, přístup k databázi a serverová kompozice šablon. Každý framework je provozován v realistickém produkčním nastavení. [30]
- attractivechaos/plb2: Benchmark Programovacích Jazyků v2 (plb2) hodnotí výkon 25 programovacích jazyků při čtyřech výpočetně náročných úlohách.[34]

2.2.4 Výběr technologií pro backend

Po identifikaci klíčových aspektů a datových zdrojů pro rozhodování je možné nyní přistoupit k samotnému výběru konkrétní technologie. Stejně jako u frontendové části aplikace bude i zde

použitý strukturovaný přístup založeným na vícekriteriální analýze variant (MCDA). V souladu s metodikou popsanou v předchozí, která využívá metodu AHP, je nutné převést naše obecné nefunkční požadavky na konkrétní, měřitelné a kvantifikovatelná kritéria, pro která existují k dispozici data. Při prvotním výběru potenciálních backendových technologií, které budou v tomto procesu hodnoceny, se vycházelo z programovacích jazyků a jejich hlavních frameworků figurujících v zmíněných průzkumech a benchmarcích. Ze seznamu byly odstraněny jazyky a technologie, které nejsou primárně určeny k běhu webového serveru a implementaci backendové logiky.

Bylo zváženo zahrnutí jazyků jako TypeScript a Kotlin, které jsou sice velmi populární pro vývoj, ale typicky se kompilují do jiných jazyků nebo běží na existujících platformách. Pro účely výběru primární backendové platformy/technologie došlo k rozhodnutí se soustředit na základní runtime prostředí/jazyky, pod kterými tyto jazyky nejčastěji pro backend běží. Jejich popularita a výkonnostní data se totiž často vážou k těmto základním platformám. Volba konkrétního jazyka jako TypeScriptu nebo Kotlinu v rámci již zvolené platformy je pak spíše implementačním rozhodnutím v rámci daného ekosystému, nikoliv rozhodnutím o základní backendové technologii pro server.

Definice kritérií pro hodnocení backendových technologií vychází z našich nefunkčních požadavků a dat dostupných pro vyfiltrované kandidátní technologie. Tato kritéria jsou odvozena z metrik, které byly představeny v předchozí sekci, a mapují se na naše NP následovně:

- NP01 - "Použití moderních a populárních technologií": Kvantifikace popularity, rozšířenosti a žádanosti technologie v komunitě. Využijí se metriky z dotazníkových šetření [27, 28]: Popularita C1, Obdivovanost C2 a Žádanost C3
- NP02 - "Výkon a odolnost vůči zátěži": Posouzení schopnosti technologie rychle a efektivně zpracovávat požadavky pod zátěží na základě benchmarkových dat průměr elapsed z plb2[34] (C4) pouze pro samotné programovací jazyky, nikoliv pro frameworky.
- NP04 - "Technologie využité systémem musí být jednoduché": Hodnoceno nepřímo na základě metrik odrážejících pozitivní zkušenost a preference vývojářů, jako jsou Obdivovanost C2 a Žádanost C3, u kterých se předpokládá korelace s vnímanou jednoduchostí a příjemností vývoje.

- Kompatibilita s front-endem C5: Protože uživatelské rozhraní běží v Javascriptu, považuje se za výhodu, když lze tentýž jazyk využít i na serveru. Současné používání více programovacích jazyků totiž zbytečně komplikuje projekt – integrace mezi jednotlivými částmi aplikace i celková architektura jsou přehlednější, když se držíte jednoho populárního jazyka, což snižuje složitost kódu a zvyšuje produktivitu vývojářů [35]. Díky možnosti sdílet a znovu použít úseky kódu mezi front i backendem mohou full-stack developeři využít existující knihovny a komponenty, což zkracuje vývojové cykly a zlepšuje konzistenci kódu [35]. Na základě těchto zkušeností bylo proto do vícekriteriálního hodnocení zařazeno binární kritérium kompatibility s frontendem: technologie v Javascriptu/Typescriptu, které takové sdílení kódu umožňují, dostávají nejvyšší váhu, zatímco řešení v jiných jazycích jsou ohodnocena nižší.

Zde je tabulka s párovým srovnáním alternativ pro jednotlivé kritéria:

Tabulka 7: Matice párových porovnání kritérií pro výběr backendového jazyk – Zdroj: vlastní

Jazyk	C1	C2	C3	C4	C5
C1: Popularity	1	3/2	3	3/2	3
C2: Admired	2/3	1	2	1	2
C3: Desired	1/3	1/2	1	1/2	2
C4: Elapsed	2/3	1	1/2	1	1
C7: FE Comp.	1/3	1/2	1/2	1/2	1

Tato matice má poměr konzistence CR = 0.0498 <0.10, což znamená, že je dostatečně konzistentní. Váhy kritérií odvozené z této matice jsou:

Tabulka 8: Váhy pro výběr backendového jazyk – Zdroj: vlastní

Popularity	Admired	Desired	Elapsed	FE Comp.
0.332	0.221	0.180	0.155	0.112

Zde je tabulka s normalizovanými daty pro jednotlivé jazyky vůči kritériím (C1-C5):

Tabulka 9: Hodnoty pro výběr backendového jazyk – Zdroj: vlastní

Language	Popularity	Admired	Desired	Elapsed	FE Comp.
JavaScript	0.61	0.583	0.398	0.538	1.00
Python	0.57	0.676	0.419	0.205	0.00
Java	0.46	0.476	0.179	0.487	0.00
C#	0.22	0.641	0.216	0.809	0.00
C	0.18	0.474	0.139	1.000	0.00
Go	0.18	0.677	0.231	0.780	0.00

PHP	0.17	0.483	0.096	0.023	0.00
Rust	0.11	0.822	0.287	0.977	0.00

Pro kritérium C4 (Elapsed), měřící výkon časem dokončení úkolu z benchmarku PLB2 [34]. Tento benchmark zahrnuje sadu standardizovaných algoritmů (nqueen, matmul, sudoku, bedcov). Pro každý jazyk jsme získali data z nejlepšího dostupného runtime v tomto benchmarku. Hodnota 'Elapsed' pro jazyk je průměrem časů naměřených na těchto standardizovaných algoritmech pro vybraný runtime. Tento průměr reprezentuje typický výkonnostní potenciál nejlepší implementace na dané sadě testů benchmarku. Protože nižší čas znamená lepší výkon, získaná průměrná hodnota je normalizována na škálu 0 až 1

Zde je tabulka s výsledky:

Tabulka 10: Výsledky výběru backendového jazyka – Zdroj: vlastní

JS	Python	Rust	C#	Go	Java	C	PHP
0.599	0.446	0.421	0.378	0.372	0.366	0.345	0.184

Nejvhodnější volbou je tedy JavaScript. Volba jakékoli programovací technologie nebo platformy s sebou nese nutnost detailně se seznámit a pracovat s jejími specifickými nástroji, ekosystémem a způsoby nasazení. V případě programovacího jazyka JavaScript pro backend se rozhodovací proces nyní posouvá k další úrovni abstrakce, a to k výběru vhodného runtime prostředí, které je pro spuštění Javascriptového kódu mimo prostředí webového prohlížeče nezbytné. V současném ekosystému backendového Javascriptu existuje několik významných runtime prostředí, z nichž každé přináší specifické vlastnosti, výhody a nevýhody z hlediska výkonu, bezpečnosti, kompatibility s existujícími knihovnami a vestavěných funkcionalit. Volba konkrétního runtime prostředí pro backendovou aplikaci napsanou v Javascriptu je dalším kritickým rozhodnutím, které ovlivní jak vývojový proces, tak vlastnosti finální aplikace. Stejně jako při výběru programovacího jazyka, i zde je nutné zvážit specifické požadavky projektu a opřít se o dostupná data a srovnání těchto runtime prostředí. Z průzkumu State of JS patří mezi hlavní hráče na tomto poli:

- Node.js: asynchronní runtime prostředí Javascriptu řízené událostmi, je dlouhodobě zavedeným nástrojem pro server-side JavaScript, známý pro svůj rozsáhlý ekosystém balíčků (npm) a design zaměřený na škálovatelné síťové aplikace s neblokujícím I/O [36]

- Deno: Novější runtime prostředí Javascriptu s důrazem na bezpečnost – ve výchozím stavu běží kód v sandboxu a přístup k systému vyžaduje explicitní povolení. Je postavené na webových standardech a nabízí nativní podporu pro TypeScript a moderní ECMAScript funkce s nulovou konfigurací [37]
- Bun: Runtime prostředí, navržené s primárním cílem extrémního výkonu a fungující jako rychlý all-in-one JavaScript runtime a toolkit pro vývoj v Javascriptu/Typescriptu [38].

Podobně jako pro programovací jazyk, kritéria pro hodnocení a výběr nejvhodnějšího Javascriptového runtime prostředí byly definována tak, aby navazovala na naše nefunkční požadavky:

- Pro posouzení modernosti a popularity (NP01) runtime prostředí se bude vycházet z jeho popularity a míry adopce (C1) z průzkumu State of JS. Důležitým ukazatelem je také vitalita ekosystému, kterou lze sledovat například růstem hvězdiček na GitHubu za rok 2024 (C2), což indikuje rostoucí zájem a aktivní vývoj.
- Pro posouzení výkonu (NP02) runtime prostředí se bude vycházet z jeho rychlosti kterou založíme na benchmarku PLB2[34] (C3)
- Pro posouzení škálovatelnosti (NP03) runtime prostředí se bude vycházet podobně jako u výběru frontendového frameworku z metriky použití ve větších týmech 101-1000 lidí (C4)
- Pro posouzení jednoduchosti (NP04) runtime prostředí se bude vycházet opět podobně jako u výběru frontendového frameworku inverzním způsobem z metriky použití ve menších týmech 1-5 lidí (C5)

Tabulka 11: Matice párových porovnání kritérií pro výběr JavaScript runtime – Zdroj: vlastní

Kritéria	C1	C2	C3	C4	C5
C1: Popularita	1	1/7	1/9	1/2	1/3
C2: Růst GH hvězd/rok	7	1	1/7	5	4
C3: Elapsed Avg.	9	7	1	9	9
C4: Tým 101-1000	2	1/5	1/9	1	1/2
C5: Tým 1-5	3	1/4	1/9	2	1

Tato matice má poměr konzistence CR = 0,085 <0.10, což znamená, že je dostatečně konzistentní. Váhy kritérií odvozené z této matice jsou:

Tabulka 12: Váhy pro výběr JavaScript runtime – Zdroj: vlastní

C1	C2	C3	C4	C5
0.038	0.217	0.604	0.056	0.084

Zde je tabulka s normalizovanými daty pro jednotlivé jazyky vůči kritériím (C1-C5):

Tabulka 13: Hodnoty pro výběr JavaScript runtime – Zdroj: vlastní

Runtime	Usage	GH hvězd/rok	Elapsed průměr	Použítí 101-1000	Použítí 1-5
NodeJS	0.91	0.07	0.80	0.91	0.91
Bun	0.16	0.11	1.00	0.13	0.24
Deno	0.12	0.09	0.75	0.10	0.15

Celkové skóre pro jednotlivé technologie jsou:

Bun: (0.16×0.038) + (0.11×0.217) + (1.00×0.604) + (0.13×0.056) + (0.24×0.084) = 0.0061 + 0.0239 + 0.6040 + 0.0073 + 0.0202 = 0.6615
NodeJS: (0.91×0.038) + (0.07×0.217) + (0.80×0.604) + (0.91×0.056) + (0.91×0.084) = 0.0346 + 0.0152 + 0.4832 + 0.0510 + 0.0764 ≈ 0.6604
Deno: (0.12×0.038)+ (0.09×0.217) + (0.75×0.604) + (0.10×0.056) + (0.15×0.084) = 0.0046 + 0.0195 + 0.4530 + 0.0056 + 0.0126 = 0.4953

Vychází z toho, že nejvhodnějším runtime je **Bun**.

Tím je tedy vybraný jazyk a runtime. V neposlední řadě je vhodné vybrat framework, kterých je trochu více, a tedy výběr proběhne z frameworků, které podporují NodeJS a užití je nad 3 %, metriky:

- NP01 – Pro posouzení modernosti a frameworku runtime se bude vycházet z míry použití (C1) z průzkumu State of JS. Důležitým ukazatelem je také vitalita ekosystému, kterou lze sledovat například růstem hvězdiček na GitHubu za rok 2024 (C2), což indikuje rostoucí zájem a aktivní vývoj.
- NP02 - "Výkon a odolnost vůči zátěži": Posouzení schopnosti technologie rychle a efektivně zpracovávat požadavky pod zátěží na základě benchmarkových dat průměr z několika již zmíněných zdrojů konkrétně – RPS (C3) a Latecty(C4) z the-benchmarker/web-frameworks [29] a Fortunes(C5) z TechEmpower [30].
- NP03 – Škálovatelnost systému – bude se vycházet podobně jako u výběru frontendového frameworku z metriky použití ve větších týmech 101-1000 lidí (C6)
- Pro NP04 - "Jednoduchost použití" - bude se vycházet opět podobně jako u výběru frontendového frameworku inverzním způsobem z metriky použití ve menších týmech 1-5 lidí (C7)

Zde je tabulka s párovým srovnáním alternativ pro jednotlivé kritéria:

Tabulka 14: Matice párových porovnání kritérií pro výběr backend frameworku – Zdroj: vlastní

Criteria	C1	C2	C3	C4	C5	C6	C7
C1: Použití	1	1/7	1/3	1/5	1/5	1/2	1/2
C2: GH stars	7	1	5	3	3	6	6
C3: RPS	3	1/5	1	1/3	1/3	2	2
C4: Fortunes	5	1/3	3	1	1	4	4
C5: Latence (ms)	5	1/3	3	1	1	4	4
C6: Použití týmu 101-1000	2	1/6	1/2	1/4	1/4	1	1
C7: Použití týmu 1-5	2	1/6	1/2	1/4	1/4	1	1

Tato matice má poměr konzistence CR = 0.0205 <0.10, což znamená, že je dostatečně konzistentní. Váhy kritérií odvozené z této matice budou:

Tabulka 15: Váhy pro výběr backend frameworku – Zdroj: vlastní

C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
0.040	0.409	0.098	0.177	0.177	0.049	0.049

Data pro hodnocení:

Tabulka 16: Data pro výběr backend frameworku – Zdroj: vlastní

Runtime	**Usage**	**GH stars**	**RPS**	**Fortunes**	**Latence**	**Použití 101-1000**	**Použití 1-5**
Express	0.68	0.05	0.09	0.01	3.56	0.68	0.66
Nest	0.29	0.09	0.00	0.00	0	0.30	0.22
Fastify	0.15	0.09	1.00	1.00	0.93	0.15	0.16
Hono	0.11	0.54	0.36	0.94	1.00	0.08	0.15

Výsledek:

Hono: (0.110.040) + (0.540.409) + (0.360.098) + (0.940.177) + (0.930.177) + (0.080.049) + (0.15*0.049) ≈ 0.603
Fastify: (0.150.040) + (0.090.409) + (1.000.098) + (1.000.177) + (1.000.177) + (0.150.049) + (0.16*0.049) ≈ 0.510
Express: (0.680.040) + (0.050.409) + (0.090.098) + (0.010.177) + (0.2610.177) + (0.680.049) + (0.66*0.049) ≈ 0.170
Nest: (0.290.040) + (0.090.409) + (0.000.098) + (0.000.177) + (0.000.177) + (0.300.049) + (0.22*0.049) ≈ 0.074
Z tohoto vychází, že nejvhodnějším frameworkem je **Hono**.

2.2.5 Výběr databáze

Databáze je klíčovou součástí backendu aplikace a zajišťuje trvalé uchování dat pro serverovou logiku i další operace [39]. Při výběru databázového systému bylo rozhodnuto nepoužít formální metodu, jako je MCMD, protože počet relevantních alternativ byl omezený a PostgreSQL již v počáteční analýze splňoval všechna hlavní kritéria bez nutnosti kompromisů. Výběr tak vycházel ze stanovených technických požadavků projektu, praktických zkušeností a vlastností jednotlivých systémů.

Prvním posuzovaným kritériem byl typ datového modelu, tedy způsob organizace a propojení dat. Jelikož aplikace pracuje se strukturovanými daty a jasně definovanými vztahy, je vhodné použít relační databázi [40].

Z hlediska škálovatelnosti byl s ohledem na požadavky zohledněn budoucí růst, a proto bylo důležité zvolit databázové řešení, které by potenciálně umožnilo horizontální škálování s minimálním dopadem na aplikační architekturu [41]. Ačkoli standardní instalace PostgreSQL ve výchozím stavu nenabízí vestavěné horizontální škálování dat napříč více servery, bylo zvoleno PostgreSQL. Toto rozhodnutí vychází z toho, že PostgreSQL nejlépe odpovídá požadavkům aplikace z hlediska datové struktury, výkonu a spolehlivosti. Toto rozhodnutí bylo učiněno s cílem nezvětšovat rozsah práce a zachovat jednoduchost demonstrace funkčního systému. To však neznamená, že budoucí škálování databáze není možné. Pro zvýšení propustnosti databáze a zajištění vysoké dostupnosti lze v budoucnu využít replikaci nebo implementovat pokročilejší distribuované strategie pro škálování zápisů a velmi velkých datasetů pomocí rozšíření jako je Citus [61] či migrovat na kompatibilní distribuované databázové systémy jako je YugabyteDB, která je PostgreSQL kompatibilní.[61]

2.2.6 Výběr dodatečných knihoven pro backend

Po definování základního technologického stacku backendu je vytvořen základ pro vývoj serverové aplikace. Ačkoli některá moderní runtime prostředí, jako například Bun, nabízejí vestavěné implementace určitých běžných funkcionalit [38], bylo rozhodnuto pro účely tohoto projektu využít dedikovanou externí knihovny pro některé z těchto oblastí s cílem maximálně zjednodušit vývojový proces a využít výhod specializovaných řešení.

Pro interakci s relační databází a překonání rozdílů mezi relačním a objektovým modelem, což ORM řeší abstrakcí databázových operací pro zjednodušení psaní dotazů a správu schémat, byl pro databázi PostgreSQL vybrán Drizzle ORM. S Drizzle je možné definovat a spravovat databázová schémata přímo v TypeScriptu a přistupovat k datům způsobem podobným SQL

nebo relačně, což přispívá k lepší vývojářské zkušenosti. Jedná se o moderní ORM zaměřený na typovou bezpečnost a vysoký výkon, poskytující pohodlné a typově bezpečné API pro práci s daty [44]

2.3 Architektura aplikace

Architektura aplikace představuje architektonický návrh a organizaci systému v rámci, které činíme rozhodnutí o uspořádání, včetně výběru strukturálních prvků a jejich rozhraní, jimiž je systém sestaven, a chování definovaného spoluprací těchto prvků [46]. Důsledně promyšlený návrh architektury umožnuje dosažení klíčových vlastností, jako jsou škálovatelnost, udržitelnost, spolehlivost a snadná údržba celého systému [45]. Následující kapitola poskytuje celkový přehled navržené architektury aplikace, popisují její klíčové komponenty, strategie škálování a návrh aplikačního rozhraní (API). Je třeba zdůraznit, že s ohledem na požadavek na jednoduchost řešení (NP04), a zároveň vzhledem k rozsahu této práce a cílové pochopitelnosti navrženého systému, byla komplexita architektury záměrně omezena na nezbytné minimum pro demonstraci funkčního celku. Přesto jsou do popisu architektury zahrnuty informace o možnostech jejího budoucího rozšíření a adaptace na rostoucí požadavky.

2.3.1 Klíčové komponenty systému

Při návrhu architektury systému bylo klíčové vycházet z definovaných Scénářů a souvisejících funkčních požadavků, které pomohly identifikovat jednotlivé prvky a klíčové komponenty systému:

Analýza Scénáře 1 odhalila potřebu uživatelského rozhraní pro iniciaci vratky (dále značeno jako R1), které musí zákazníkům umožňovat jednoduchou a intuitivní cestu pro podání požadavku na vrácení zboží. Zásadní funkcionalitou je identifikace objednávky (prostřednictvím čísla a emailu), výběr konkrétních produktů k vrácení a vyplnění detailního formuláře s důvodem vratky. Z tohoto scénáře přímo vyplývá potřeba specializovaného webového rozhraní (R1), jež bude výhradně sloužit zákazníkům pro odeslání požadavku žádosti o vrácení zboží.

Po odeslání požadavku zákazníkem systém vyžaduje automatické vygenerování průvodní dokumentace. Tato funkcionalita indikuje potřebu serverové procesní služby (dále značeno jako S1), jež bude zodpovědná za komplexní zpracování požadavku. Služba musí být schopna validovat vstupní data, generovat potřebné dokumenty a iniciovat navazující procesy.

Scénář 2 přináší požadavek na sledování aktuálního stavu vratky zákazníkem. Z architektonického hlediska lze tuto funkcionalitu efektivně řešit rozšířením stávajícího zákaznického rozhraní (R1), což eliminuje potřebu vytváření samostatné komponenty. Procesní služba (S1) bude zodpovědná za poskytování aktuálních informací o průběhu vratky, včetně potvrzení přijetí zboží, kontroly stavu a finálního rozhodnutí o vrácení peněz nebo výměně zboží.

Scénář 3 identifikuje potřebu druhého uživatelského rozhraní – administrátorského panelu (dále značeno jako R2), který umožní zaměstnancům e-shopu spravovat příchozí požadavky na vrácení zboží. Toto rozhraní musí poskytovat komplexní přehled žádostí, umožňovat jejich detailní kontrolu a aktualizaci stavů. Argument pro využití dvou samostatných frontend aplikací (R1 a R2) vychází z principu oddělení zodpovědností a umožnění nezávislého vývoje a specifického řízení přístupových práv. I v případě administrátorského panelu lze pro zpracování změn stavů a řízení workflow vratky využít již existující serverovou procesní službu (S1). Integrace mechanismu pro zpracování informací o požadavcích z obou uživatelských rozhraní do jedné služby (S1) umožní konsolidovat procesní logiku, zajistit konzistentní zpracování požadavků a podstatně zjednodušit údržbu systému.

Klíčovým prvkem systému bude i datové úložiště (dále značeno jako D1), které bude centrálně uchovávat veškeré informace o vratkách. Toto úložiště musí být navrženo tak, aby poskytovalo konzistentní rozhraní pro oba frontend systémy – zákaznický (R1) a administrátorský (R2) – a procesní službu (S1).

2.3.2 Plánování architektury

Po identifikaci klíčových komponent systému na základě analýzy scénářů a funkčních požadavků bylo dalším krokem zvolit vhodný architektonický pattern, který definuje celkovou strukturu aplikace a způsob interakce jejích částí. Po zvážení požadavků projektu, obzvláště s ohledem na jednoduchost (NP05) a charakter systému, se jeví jako nejvhodnější vrstvená architektura (Layered Architecture Pattern) [47]. Tento pattern nabízí jasné oddělení zodpovědností a přehlednou strukturu, což usnadňuje vývoj, testování a údržbu systému [47].

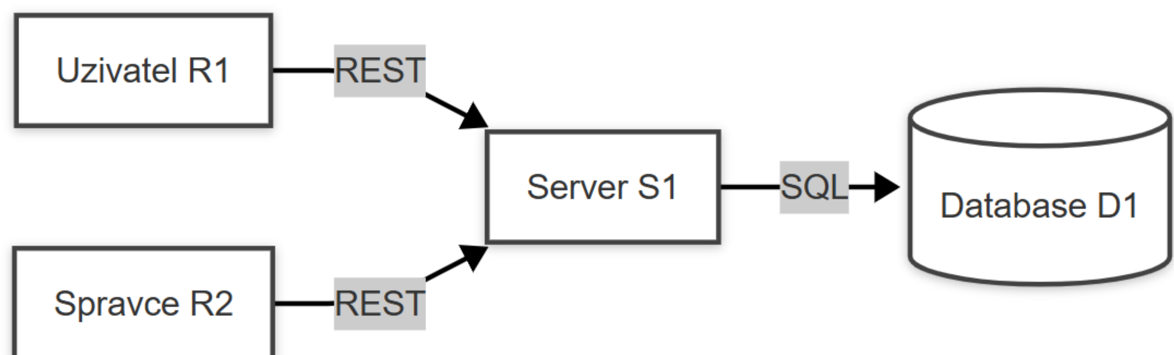
V tomto případě vrstvenou architekturu bude rozdělena v kontextu aplikace na:

- Prezentační vrstvu: Zodpovědná za uživatelské rozhraní (R1 – zákaznické, R2 – administrátorské).
- Aplikační/Business logickou vrstvu: Obsahuje hlavní procesní logiku aplikace (S1).
- Datovou/Perzistentní vrstvu Zajišťuje ukládání a správu dat (D1).

Tato architektura umožňuje relativně flexibilní škálování a nezávislý vývoj vrstev. Jednotlivé vrstvy lze nezávisle vyvíjet, testovat, optimalizovat a v budoucnu případně nahrazovat, aniž by to vyžadovalo zásadní zásahy do ostatních vrstev. Například je možné snadno aktualizovat prezentační vrstvu nebo přidat nové API rozhraní, aniž by bylo nutné měnit logiku v aplikační či datové vrstvě [47].

Komunikace mezi jednotlivými vrstvami tvoří klíčový aspekt architektury a definuje, jak komponenty v různých vrstvách spolu interagují. V kontextu zvolené vrstvené architektury se primárně držíme principu "uzavřených vrstev" (closed layers). V modelu uzavřených vrstev může komponenta v dané vrstvě interagovat pouze s komponentami ve vrstvě bezprostředně pod ní. Toto omezení zajišťuje jasnou strukturu závislostí a usnadňuje správu komplexity [47].

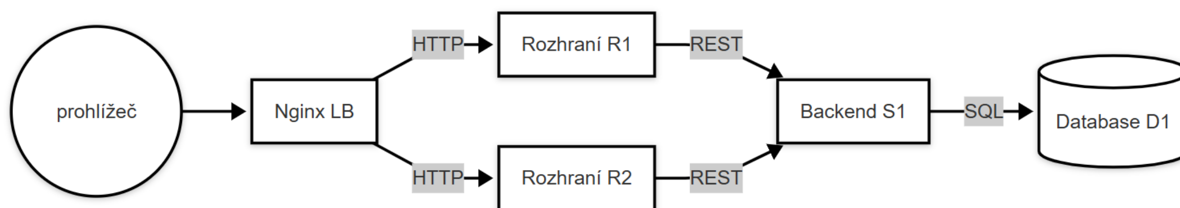
Mezi prezentační vrstvou (R1, R2) a aplikační vrstvou (S1) bude primárně využito REST API [6, 12]. Tento standardizovaný přístup zajišťuje jasně definované rozhraní a usnadňuje interakci. Aplikační vrstva (S1) se bude interagovat s datovou vrstvou (D1) prostřednictvím protokolu SQL, abstrahovaného přes zvolený ORM (Drizzle ORM), jak bylo popsáno v předchozí kapitole.



Obrázek 1: Graf základní struktury systému – Zdroj: Vlastní tvorba na základě [47]

Při navrhování architektury je třeba si uvědomit a ohodnotit architekturu z pohledu uživatele, kde první interakce uživatele začíná, když webový prohlížeč vyšle požadavek na zobrazení uživatelského rozhraní [7]. Z technického hlediska je tedy nutné mít komponentu schopnou efektivně reagovat na tyto požadavky a doručovat potřebné soubory prohlížeči. Ačkoli runtime prostředí jako Bun mohou obsahovat vestavěné schopnosti webového serveru, pro nasazení v produkčním prostředí s očekávanou zátěží můžeme použít i dedikovaný webový server. Webové servery, jako například Nginx nebo Apache [12], mají klíčovou výhodou souběžné zpracování požadavků s vysokým výkonem a efektivitou [50]. Pro tuto práci byl zvolený Nginx.

Tímto krokem se odlehčí backendové aplikační službě (S1), která se pak může plně soustředit na zpracování aplikační logiky [12].



Obrázek 2: Graf struktury systému s webovým serverem – Zdroj: Vlastní tvorba na základě [47, 12]

2.3.2 Strategie škálování

Požadavky jako NP03 a NP04 jdou nad rámec pouhého zajištění funkčního řešení a stojí za detailnější architektonické zamyšlení nad způsobem zpracování požadavků a distribuce zátěže. Pro zvýšení propustnosti systému a zajištění vysoké dostupnosti je jedním základním přístupem horizontální škálování, tedy spuštění více instancí služby, často distribuovaných na několika malých strojích [49]. V naší aplikaci backend běží na runtime prostředí Bun. Bun sice díky využití socket-options (SO_REUSEPORT, SO_REUSEADDR) umožňuje horizontální distribuci zátěže, ale pouze v rámci jednoho stroje [51]. Skutečné škálování nezávislé na stroji vyžaduje nasazení více instancí na různých strojích, kde distribuci příchozí zátěže můžeme provést pomocí load balanceru. Load balancer efektivně směruje požadavky klientů na dostupné instance, čímž předchází přetížení a zajišťuje rovnoměrné zpracování [48].

Pro orchestraci, správu a škálování těchto instancí byla zvolena platforma Kubernetes, konkrétně jeho lehká distribuce k3s. Toto řešení efektivně naplňuje požadavek na jednoduchost, neboť Kubernetes poskytuje komplexní vestavěné mechanismy pro automatizované nasazování, škálování a správu aplikací [52]. Kubernetes používá abstrakci kontejnerů, které izolují aplikace do vlastního prostředí čímž zajišťují konzistentní chování na jakémkoliv stroji [53]. Kubernetes vyžaduje, aby každý kontejner běžel v Podu [52]. Pod představuje nejmenší nasaditelnou jednotku [52] a sdružuje jeden nebo více kontejnerů [52].

Backendovou aplikaci jsem tedy nejprve zabalili do Docker kontejneru. Tento kontejner je následně nasazen v rámci Podu a Kubernetes se stará o jeho běh a distribuci zátěže mezi jednotlivými Pody napříč dostupnými uzly. Pro externí přístup k těmto Podům a řízení

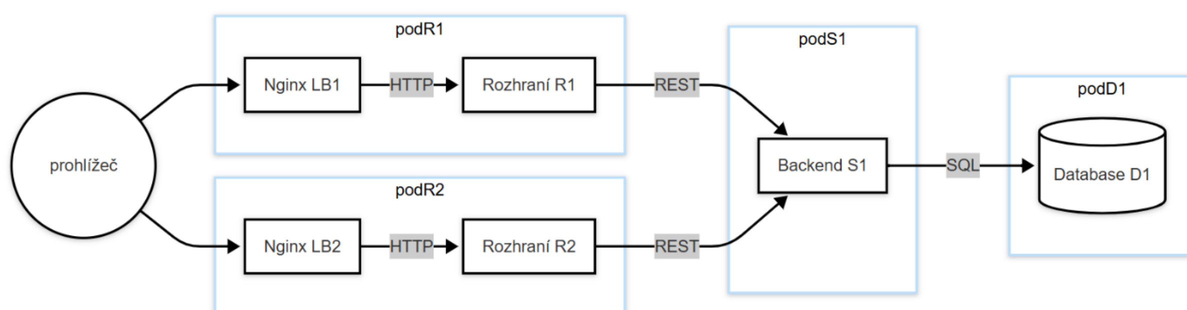
směrování příchozího provozu na základě pravidel je využíván Ingress Controller, který vystavuje více služeb přes jeden load-balancer [52].

Stejný princip nasazení a škálování se uplatní i pro oba frontendové moduly – uživatelské webové rozhraní a administrační rozhraní. Protože se jedná o statické nebo předem sestavené aplikace, kontejner obsahuje kromě aplikačních souborů také již zmíněný webový server, který servíruje finální kód. Kubernetes následně orchestruje Pody s těmito kontejnery a Ingress se opět postará o správné směrování externích požadavků na odpovídající frontendové služby a Pody, čímž zajišťuje jejich vysokou dostupnost a škálovatelnost.

Odlišný přístup ke škálování bude nutný u stavových aplikací (Stateful Applications), jako je zvolená databáze PostgreSQL, která na rozdíl od BE a FE uchovává perzistentní data. V Kubernetes se toto řeší pomocí Persistent Volumes (PV), které reprezentují externí úložiště a Persistent Volume Claims (PVC), které umožňují databázovému Podu „nárokovat" se do PV [52].

Kubernetes také podporuje automatické škálování podů na základě prostředků [52], ale pro účely demonstrace a zátěžových testů v rámci této práce jsem využili manuální škálování.

Současný návrh architektury tak představuje jednoduché, avšak robustní a flexibilní řešení, plně připravené na budoucí rozšíření funkcionalit a efektivní provoz i pod výrazně zvýšenou zátěží.



Obrázek 3: Graf struktury systému s vyznačením podů – Zdroj: Vlastní tvorba na základě [47, 12, 52]

2.3.3 Návrh API

Návrh RESTful API (Representational State Transfer Application Programming Interface) a bude vycházet z již definovaných scénářů použití. API obecně představuje sadu definic a protokolů, které umožňují softwarovým komponentám vzájemnou komunikaci a interakci [42]. RESTful API je pak specifickým typem webového API. Základní myšlenkou REST je, že

prostředek, např. dokument, je přenášen prostřednictvím obecně uznávaných, jazykově nezávislých a spolehlivě standardizovaných interakcí klient–server [43]. Toto API bude implementováno na backendové službě a bude sloužit jako primární komunikační rozhraní pro frontendové aplikace (zákaznický web i administrační rozhraní).

2.3.3.1 Zákaznické koncové body

- Endpoint pro verifikaci objednávky a získání produktů k vrácení
  - Účel: Tento `GET` endpoint umožňuje zákazníkovi ověřit detaily své objednávky pomocí čísla objednávky a e-mailové adresy. V případě úspěšné verifikace vrátí seznam produktů z dané objednávky, které jsou způsobilé k vrácení.
  - Metoda a cesta: `GET /api/order?orderId={orderId}&email={email}`
  - Vstupní parametry:
    - `orderId` (query parametr): Jedinečné číslo objednávky.
    - `email` (query parametr): E-mailová adresa spojená s objednávkou pro účely verifikace.
  - Očekávaná odpověď: Seznam produktů. Následně si zákazník může z tohoto seznamu vybrat jeden nebo více produktů, které chce vrátit
- Endpoint pro vytvoření požadavku na vrácení zboží
  - Účel: Tento `POST` endpoint slouží k vytvoření nového požadavku na vrácení zboží. Zákazník specifikuje produkty, které chce vrátit, a uvede důvod vrácení.
  - Metoda a cesta: `POST /api/request`
  - Vstupní data: V "body" bude objekt obsahující seznam vybraných ID produktů, důvod vrácení
  - Očekávaná odpověď: Unikátní identifikátor (ID) nově vytvořeného požadavku na vrácení zboží.
- Endpoint pro zjištění stavu vratky
  - Účel: Tento `GET` endpoint umožňuje zákazníkovi dotázat se na aktuální stav konkrétního požadavku na vrácení zboží pomocí jeho ID.
  - Metoda a cesta: `GET /api/request/{requestId}`
  - Vstupní parametr: `requestId` (path parametr): ID požadavku na vrácení zboží.
  - Očekávaná odpověď: Objekt obsahující historii stavů požadavku (např. "čeká na schválení", "schváleno", "zamítnuto", "zpracováno") a další související detaily.

2.3.3.2 Administrační koncové body

- Endpoint pro autentifikaci (přihlášení)
  - Účel: Tento `POST` endpoint slouží k přihlášení administrátorů do systému.
  - Metoda a cesta: `POST /api/auth/login`
  - Vstupní data: Objekt obsahující uživatelské jméno a heslo.
  - Očekávaná odpověď: V případě úspěšné autentifikace je vrácena autentifikační cookie pro správu session.
- Endpoint pro odhlášení administrátora

    - Účel: Tento `POST` endpoint umožňuje administrátorovi odhlásit ze systému, což invaliduje jeho session a smaže cookie.
    - Metoda a cesta: `POST /api/auth/logout`
- Endpoint pro verifikaci autentifikace
    - Účel: Tento `GET` endpoint umožňuje frontendu ověřit, že je session uživatele platná.
    - Metoda a cesta: `GET /api/auth/me`
    - Očekávaná odpověď: V případě platné session je vrácena autentifikační cookie pro správu session.
- Endpoint pro získání seznamu všech požadavků na vrácení zboží
    - Účel: Tento `GET` endpoint umožňuje administrátorům získat přehled všech požadavků na vrácení zboží v systému.
    - Metoda a cesta: `GET /api/requests`
    - Očekávaná odpověď: Seznam objektů, kde každý objekt reprezentuje požadavek na vrácení zboží s klíčovými informacemi
- Endpoint pro získání detailu konkrétního požadavku na vrácení zboží
    - Účel: Tento `GET` endpoint umožňuje administrátorům získat detailní informace o konkrétním požadavku na vrácení zboží. Tento endpoint je identický s endpointem pro zjištění stavu vratky určeným pro zákazníky.
    - Metoda a cesta: `GET /api/request/{requestId}`
    - Vstupní parametr: `requestId` (path parametr): ID požadavku na vrácení zboží.
    - Očekávaná odpověď: Detailní objekt požadavku na vrácení zboží, včetně všech souvisejících dat.
- Endpoint pro změnu stavu požadavku na vrácení zboží
    - Účel: Tento `PATCH` endpoint umožňuje administrátorům aktualizovat stav konkrétního požadavku na vrácení zboží
    - Metoda a cesta: `PATCH /api/request/{requestId}`
    - Vstupní parametry:
        - `requestId` (path parametr): ID požadavku na vrácení zboží.
        - Tělo požadavku: Objekt obsahující nový stav
    - Očekávaná odpověď: Potvrzení úspěšné změny stavu

2.3.4 Návrh databázového schématu

Databáze byla navržena tak, aby efektivně zachycovala hlavní prvky definovaných scénářů použití, především objednávky a požadavky na vrácení zboží. Pro prvky jako jsou produkty, které nejsou nutně určeny k detailnímu zobrazení v samostatných databázových entitách, bylo zvoleno zjednodušení použitím semistrukturovaného datového typu JSONB. Toto rozhodnutí bylo učiněno s ohledem na zachování jednoduchosti návrhu a snížení implementační složitosti.

- Tabulka users slouží k ukládání informací o administrátorech systému, kteří mají přístup do administračního rozhraní.
    - `id`: Jedinečný primární klíč, automaticky generovaný databází s inkrementální hodnotou.

    - `email`: E-mailová adresa uživatele. Je povinná a musí být unikátní, slouží jako primární identifikátor pro přihlášení.
    - `password`: Hashované heslo uživatele. Povinný sloupec.
- Tabulka `orders` (Objednávky)
    - `id`: Jedinečný primární klíč, automaticky generovaný databází s inkrementální hodnotou.
    - `orderNumber`: Unikátní číslo objednávky, povinný sloupec. Slouží jako referenční bod pro požadavky na vrácení.
    - `email`: E-mailová adresa zákazníka spojená s objednávkou. Slouží pro verifikaci objednávky v kombinaci s `orderNumber`.
    - `paidAt`: Časové razítko, kdy byla objednávka zaplacena, uloženo jako ISO řetězec. Slouží jako výchozí hodnota.
    - `createdAt`: Časové razítko, kdy byla objednávka vytvořena, uloženo jako ISO řetězec. Slouží jako výchozí hodnota.
    - `orderStatus`: Aktuální stav objednávky (např. "zaplaceno", "odesláno").
    - `products`: Pole objektů ve formátu JSONB, kde každý objekt reprezentuje jeden produkt z objednávky. Tento přístup denormalizace byl zvolen pro flexibilní ukládání produktových dat přímo v záznamu objednávky, s účelem zjednodušení implementace, jelikož nejsou detaily produktu nutné pro správu požadavků na vrácení.
- Tabulka `requests` (Požadavky na vrácení)
    - `id`: Jedinečný primární klíč, automaticky generovaný databází s inkrementální hodnotou.
    - `orderNumber`: Číslo objednávky, ke které se požadavek na vrácení vztahuje. Implicitně odkazuje na `ordersTable`.
    - `email`: E-mailová adresa zákazníka, který požadavek podal. Implicitně odkazuje na zákazníka z `ordersTable`.
    - `createdAt`: Časové razítko, kdy byl požadavek na vrácení vytvořen, uloženo jako ISO řetězec. Slouží jako výchozí hodnota.
    - `products`: Pole objektů ve formátu JSONB, kde každý objekt reprezentuje jeden produkt z objednávky. Tento přístup denormalizace byl zvolen pro flexibilní ukládání produktových dat přímo v záznamu objednávky, s učelem zjednodušení implementace, jelikož nejsou detaily produktu nutné pro správu požadavků na vrácení.
    - `events`: Pole objektů ve formátu JSONB, které zaznamenává chronologickou historii událostí souvisejících s daným požadavkem na vrácení. Tento přístup denormalizuje historii událostí přímo do záznamu požadavku, což usnadňuje rychlé načítání a zobrazení celého průběhu požadavku bez potřeby komplexních JOIN operací na samostatnou tabulku událostí. Toto rozhodnutí bylo zvoleno pro zjednodušení implementace a zachování flexibility databáze.

2.4 Implementace

Tato kapitola se zaměřuje na implementaci navrženého systému, přičemž bude postupně popsán vývoj jednotlivých komponent, počínaje backendovou službou, a následně se přesune k

frontendovým aplikacím. Celý a plně funkční zdrojový kód projektu je k dispozici v příloze A této diplomové práce pro podrobnější nahlédnutí a ověření. Pro efektivní správu a vývoj více nezávislých služeb, které sdílejí JavaScript/TypeScript kód, byla zvolena a využita struktura monorepa. Tento přístup umožňuje konsistentní správu závislostí, zjednodušené sdílení kódu mezi projekty a jednotnou konfiguraci vývojových a build nástrojů. Je důležité zmínit, že pro účely vývoje a prezentace všech kódových ukázek a postupu implementace bude využíváno prostředí operačního systému Linux. Toto rozhodnutí koresponduje s volbou cílového prostředí pro nasazení systému – Kubernetes, které rovněž efektivně funguje a vyžaduje Linuxové prostředí pro své spuštění a optimální provoz. Jako nutné předpoklady pro spuštění a testování systému je třeba mít nainstalovaný Docker pro kontejnerizaci služeb a nástroj k6 pro provádění výkonových a zátěžových testů.

2.4.1 Implementace databáze

Implementace začíná přípravou Javascriptového runtime prostředí Bun.js. Podle oficiálních instalačních pokynů stačí spustit příslušný instalační skript, který stáhne a nakonfiguruje Bun.js do systémové cesty. Po úspěšné instalaci runtime máme připravené prostředí pro spuštění serverové aplikace i správu balíčků.

Dále se připraví základní struktura projektu tak, aby jednotlivé části byly přehledně rozčleněny podle své funkce. Adresářové hierarchie:

- adresář *database* pro všeobecná nastavení a nástroje související s databází,
- adresář *server* pro samotnou backendovou aplikaci,
- adresář *frontend* pro uživatelské rozhraní,
- adresář *deployment* pro konfigurační soubory a skripty určené k nasazení systému.

V adresáři *database* je vytvořen konfigurační soubor *drizzle.config.js*, který definuje cestu ke schématu a parametry připojení k databázovému serveru. Součástí tohoto adresáře jsou také soubory *schema.ts* (definice tabulek a typů) a *seed.ts* (skript pro naplnění databáze testovacími daty).

Ve stejném adresáři jsou pomocí příkazu *bun add drizzle-kit drizzle-orm* nainstalovány knihovny pro práci s databází, které umožňují generování a aplikaci migrací. Spuštěním příkazu *bun run db:push*, který odkazuje na příslušný skript definovaný v *package.json*, se vytvoří

všechny tabulky a struktury ve skutečné databázi podle definovaného schématu. Následně je databáze naplněna testovacími daty pomocí příkazu *bun run db:seed*.

Tím je backendová databázová část připravena k dalšímu vývoji a testování, včetně integrace s frontendovými službami a nasazení do produkčního prostředí.

2.4.2 Implementace backendu

Implementace backendové části začala inicializací projektu pomocí nástroje Bun přes příkaz *bun init -m -y*.

Následně byla vytvořena základní struktura projektu se složku *src,* ve které je soubor *index.ts* obsahující téměř veškerý kód pro backendovou část a složku *middleware* obsahující middleware.

Pro backendovou službu byl použit framework Hono, který umožnil rychlé definování API rozhraní. Do projektu byly přidány potřebné balíčky pomocí příkazu *bun add hono*. Dále byl vytvořen základ kódu projektu, který obsahoval novou instanci Hono a middleware pro CORS.

Postupně byly implementovány jednotlivé API endpointy. První skupina se týká funkcionalit dostupných uživatelům – například získávání informací o objednávce, vytváření požadavku na vrácení zboží nebo zobrazení detailů požadavku. Každý endpoint pracuje s databází, ověřuje vstupní parametry a vrací odpovídající výstupy včetně chybových stavů.

Následovala implementace autentizační vrstvy, která využívá mapu relací *sessions* uchovávanou na straně serveru. Bylo vytvořeno middleware do souboru *auth.ts* ve složce *src/middleware* pro ověřování identity uživatele, které je použito u chráněných endpointů. Tím se zajistilo, že k určitým částem systému mají přístup pouze přihlášení uživatelé.

Dále byly implementovány endpointy pro přihlášení a odhlášení uživatele. Při úspěšném přihlášení se vytvoří nová session, která se přiřadí danému uživateli a uloží se do mapy relací *sessions* a do cookie, která obsahuje ID relace a je zaslána uživateli. Odhlášení pak spočívá ve zneplatnění této relace v sessions a zničení cookie.

Pro správu požadavků na vrácení zboží byla vytvořena část API určená pro administrátory. Tato část obsahuje endpointy pro zobrazení všech přijatých požadavků a pro změnu jejich stavu. Změna stavu se provádí přidáním nové události do časové osy daného požadavku.

Tím byla dokončena základní implementace backendu, který zajišťuje jak funkcionalitu pro koncové uživatele, tak i administrativní správu celého systému. Backend je připraven k nasazení a dalšímu testování ve spojení s frontendovou částí.

2.4.2.1 Jednotkové testy

Jednotkové testy byly implementovány pouze pro backendovou část systému. Slouží k ověření správného chování jednotlivých funkcí bez závislosti na databázi nebo externích službách. Testy se zaměřují zejména na validaci vstupních dat, logiku zpracování požadavků a práci s interními strukturami. Testovací soubory jsou umístěny ve složce */server/test*, odděleně od aplikační logiky. Pro psaní a spouštění testů je využit vestavěný nástroj testování runtime Bun, který umožňuje spustit všechny testy jednoduchým příkazem bun test.

2.4.3 Implementace frontednu

Frontendová část systému je rozdělena do dvou samostatných aplikací – zákaznické ve složce *frontend/zakaznik* a administrátorské ve složce *frontend/admin*. Obě aplikace sdílejí některé komponenty a závislosti, proto mají společný adresář *frontend* pro sdílené části.

Pro vývoj frontendu byl použit Vite, který umožňuje rychlou inicializaci projektů založených na Reactu s typescriptem. Každá z aplikací (zákaznická i administrátorská) byla vytvořena jako samostatná Vite šablona přes příkaz *bun create vite@latest*, ale díky využití pracovních prostorů (workspaces) mají společný soubor se závislostmi *package.json* ve složce *frontend*, čímž se zajišťuje konzistence balíčků a jednodušší správa.

V obou aplikacích byl extenzivně použit framework Tailwind CSS pro stylování a knihovna shadcn/ui pro předpřipravené komponenty. Kromě toho byly do projektu přidány i další knihovny, jako jsou React Router pro klientské routvání a React Query pro práci s daty z API. Konfigurace těchto knihoven byla sjednocena tak, aby bylo možné snadno sdílet logiku mezi oběma částmi systému.

2.4.3.1 Zákaznická část

Zákaznická část umožňuje uživatelům zadat požadavek na vrácení zboží. Aplikace obsahuje dvě hlavní stránky

První je formulář, který rozdělen do dvou kroků: V prvním kroku uživatel zadává číslo objednávky a email. Po odeslání dotazu na backend přes API dojde k ověření údajů a načtení příslušné objednávky včetně seznamu produktů. Uživatel si následně vybere, které produkty chce vrátit, a pro každý z nich zvolí důvod vrácení ze zadaného seznamu. Vnitřní stav komponenty určuje, která fáze se má zobrazit. Přechod mezi kroky se děje pomocí *useState* – správy stavu v Reactu. Po odeslání druhého kroku je požadavek odeslán na API, které vytvoří nový záznam v databázi a vrátí identifikátor požadavku.

Po vytvoření požadavku je uživatel přesměrován na stránku s detailem požadavku, kde vidí produkty, které byly do požadavku zahrnuty, a také časovou osu událostí, které s požadavkem souvisejí (například: „Přijato", „Čeká na balík", „Zpracováno"). Tyto údaje jsou načítány z backendu pomocí dotazu, který využívá React Query. Vizualizace událostí je řešena komponentou, která dynamicky zobrazuje ikony, barvy a popisy podle typu události.

2.4.3.2 Administrátorská část

Administrátorská část aplikace je chráněná autentizací a slouží k přehledu a správě všech vrácených požadavků.

Administrátor má přístup ke stránce pro přihlášení, kde zadává své přihlašovací údaje. Po jejich ověření je uživatel autentizován a na straně serveru se vytvoří session, která je na klientovi předána a uložena prostřednictvím cookie. Tato informace se odesílá s každým následujícím požadavkem. Stav přihlášení je udržován v kontextu Reactu, který je dostupný napříč celou aplikací. Při každém načtení se provádí kontrola, zda je uživatel přihlášen – v případě, že není, je uživatel přesměrován zpět na přihlašovací stránku.

Hlavní stránka je dashboard, který zobrazuje tabulku se všemi aktuálními požadavky. Každý řádek reprezentuje jeden požadavek, obsahuje informace o zákazníkovi, datu vytvoření a aktuálním stavu. Součástí každého záznamu je možnost změnit stav požadavku – například přidat novou událost typu „Balík doručen" nebo „Vyřízeno". Výběr nové události automaticky aktualizuje databázi a vizuálně odráží nový stav.

Administrátor může zobrazit detail každého požadavku kliknutím na příslušný řádek. Na stránce detailu se zobrazují podrobnosti o všech produktech zahrnutých v požadavku a časová osa událostí, podobně jako v zákaznické části. Odtud je také možné upravovat stav požadavku.

2.4.4 implementace nasazení

Pro každou komponentu systému byl v příslušných složkách vytvořen samostatný Dockerfile, který definuje, jak má výsledný image vypadat. U frontendových částí je proces dvoufázový: nejprve probíhá build pomocí Bun.js, a poté jsou statické soubory přeneseny do NGINX image, který slouží jako server. Backendový kontejner je jednodušší – obsahuje pouze serverové skripty, nainstaluje závislosti a spustí server pomocí Bun.js. U databáze je použit oficiální image PostgreSQL, který je dále rozšířen o prostředí Bun.js, aby bylo možné automaticky generovat migrace a naplnit databázi testovacími daty při startu kontejneru.

Pro jednodušší správu a sestavení všech image byl v kořenové složce vytvořen konfigurační soubor *docker-compose.yml,* který definuje jednotlivé komponenty systému a kam se image pro danou část systému má nahrát. Image jsou sestaveny pomocí *docker-compose build* a poté nahrány do lokální Docker registry, která byla spuštěna na vývojovém stroji, přes *docker-compose push*. Tento registry slouží jako zdroj pro následné nasazení image do Kubernetes, bez nutnosti využití cloudu.

Nasazení do Kubernetes je řešeno pomocí YAML konfiguračních souborů, které jsou rozděleny podle jednotlivých komponent systému ve složce *deployment*. Pro každou část existují soubory typu Deployment, Service a Ingress, pro databázi je vytvořen PersistentVolumeClaim (PVC) místo Ingressu, jelikož databáze nebude veřejně dostupná.

- Deployment definuje, kolik instancí komponenty má běžet, jaký image se má použít, jaké jsou požadované prostředky (paměť, CPU), jaké porty budou dostupné a zda je třeba připojit trvalé úložiště.
- Service definuje přístup k běžícím kontejnerům přes interní síť Kubernetes.
- Ingress specifikuje, jakým způsobem budou jednotlivé služby vystaveny navenek – včetně domén, cest a pravidel směrování.
- PersistentVolumeClaim (PVC), který zajišťuje trvalé úložiště dat i při restartu nebo opětovném nasazení služby.

Pro účely této práce, jak už bylo zmíněno, byla použitá lehká distribuce Kubernetes, K3s, která byla nainstalována lokálně dle oficiální dokumentace. Po přípravě všech image, definic a instalaci K3s bylo nasazení provedeno příkazem k3s kubectl apply -f ./deployment/{slozka}, kde {slozka} je složka s definicemi pro danou část systému. Tento příkaz vytvořil všechny potřebné služby a automaticky spustil kontejnery jednotlivých komponent.

3 Testování

Testování softwaru je soubor aktivit zaměřených na odhalování chyb, hodnocení jeho kvality a snižování rizika selhání při provozu systému [62]. Existuje několik typů testů, které můžeme provést. V rámci implementace byly vytvořeny funkční testy, které se zaměřují na testování jednotlivých funkcionalit a správného chování systému dle definovaných funkčních požadavků. Do této kategorie patří například unit testy a integrační testy, které byly použity pro ověření funkčních požadavků [62].

Nefunkční požadavky se zaměřují na to, „jak dobře systém funguje" a zahrnují charakteristiky jako výkonnost, spolehlivost nebo udržovatelnost. [62]. Ačkoli navržený systém zahrnuje i frontendovou část, testování výkonu pro účely této práce se zaměří na backendovou vrstvu, která zpracovává API požadavky a aplikační logiku. Je důležité zdůraznit, že toto testování zahrnuje interakci backendu s databází.

3.1 Metodika testování

V rámci testování výkonu byly použity tři základní typy testů, které poskytují komplexní pohled na chování systému pod různými zátěžovými profily:

- Load test: Hlavním cílem je ověřit chování systému pod očekávanou nebo mírně zvýšenou běžnou zátěží. Měří se výkonnostní metriky, jako je při cílové úrovni zátěže. Tím se ověřuje, zda systém zvládá základní stanovená očekávání pro běžný provoz.
- Stress Test: Cílem je určit limity systému tím, že je zatěžován nad rámec běžné provozní kapacity, často postupným zvyšováním zátěže, dokud aplikace nezačne selhávat nebo vykazovat výraznou degradaci výkonu. Měří se především maximální dosažitelná propustnost před selháním a chování systému blízko jeho limitů. Tento test pomáhá pochopit odolnost systému vůči přetížení.
- Spike Test: Tento test slouží k pozorování chování systému, když zátěž náhle a krátkodobě vyskočí na velmi vysokou úroveň. Měří se především doba odezvy a chybovost bezprostředně po nárazu. Test ověřuje schopnost systému zvládnout neočekávané špičky provozu.

Jelikož testování výkonu pro účely této práce se zaměřuje na backendovou vrstvu včetně interakce s databází, byla pro praktické testování vybrána omezená sada endpointů, které reprezentují klíčové operace a zatěžují jak backendovou logiku, tak databázi. Pro tento účel

jsem tedy zvolil 2 endpointy, které provádí čtecí operace z databáze (`GET /api/order`), a endpoint, který provádí zápisové operace do databáze (`POST /api/request`). Na základě definovaných typů testů a vybraných endpointů byly navrženy následující testovací scénáře, které jsou navázány na nefunkční požadavky:

- TC01 – Load Test: Tento scénář bude měř it stanovená očekávání pro běžnou zátěž. Běžnou úroveň zátěže jsme pro tuto práci stanovili na 10 req/s, což odpovídá 20x průměru (0,5 req/s) získanému z dat Alza.cz (17,9 mil. objednávek/rok [57], 10 % návratnost [56] → ~1,79 mil. vratek/rok → ~4 900/den → ~612/hod → ~10/min). Každý proces vrácení nejspíš patří 1 uživateli, který generuje průměrně 3 API volání, tedy celkově máme ~30 volání/min (0,5 req/s). Pro simulaci mírné špičky využíváme 10násobek tohoto průměru, tj. 5 req/s od 10 uživatelů.
- TC02 – Stress Test: Cílem je zjistit maximální propustnost a limity jedné instance backendu postupným zvyšováním zátěže až do bodu, kdy systém začne nesplňovat naše požadavky. Výchozí zátěž je opět 10 req/s, zvýšená až na 50násobek. Sledujeme okamžik, kdy latence či chybovost překročí definované prahy
- TC03 – Testování škálovatelnosti pomocí Stress Testu: Stejné provedení jako TC02, avšak s různým počtem instancí backendové služby. Cílem je vyhodnotit, jak se mění maximální propustnost a limity systému při přidávání instancí (NP03).
- TC04 – Spike Test: Testuje schopnost jedné instance backendu zvládnout náhlý skok zátěže. Maximální zátěž je nastavena na 50násobek běžné hodnoty, avšak dosaženo ve velmi krátkém časovém úseku (NP02).
- TC05 – Testování nárazovou zátěží při škálování: Stejné jako TC04, ale s proměnným počtem instancí backendu. Cílem je zjistit, jak škálování ovlivňuje odolnost vůči nárazové zátěži a rychlost zotavení.

Měřené metriky jsou ve všech scénářích stejné a mají následující cílové limity:

- Propustnost (req/s, req/min): Na základě stanovené očekávané zátěže
  - Pro TC01: ≥ 5 req/s (300 req/min)
  - Pro TC02 – TC05: ≥ 50 req/s (3000 req/min)
- Podle Jakob Nielsenova „Response Times: The Three Important Limits“ je 0,1 s (100 ms) hranicí, pod kterou uživatel vnímá odezvu jako okamžitou [58].
  - p95 ≤ 100ms
  - p99 ≤ 250ms
- Chybovost (error rate)

- Pod 1%

V neposlední řadě je důležité zmínit prostředí, na kterém byly testy prováděny. Pro účely této práce bylo testování realizováno na kombinaci osobního zařízení a virtuálního privátního serveru (VPS). Konkrétní specifikace použitého testovacího prostředí jsou následující:

- Osobní zařízení: Testy probíhaly na Notebooku, který disponuje procesorem Intel Core i7-10510U se 4 jádry, 16 GB operační paměti a rychlým NVMe diskem. Jako operační systém sloužil Windows 11 s využitím WSL2 (Debian 12) pro běh Linuxového prostředí.
- VPS server: Dodatečně byl využit VPS server s konfigurací 4 virtuální CPU jádra, 6 GB RAM a SSD disk, běžící na operačním systému Debian 12.

Toto kombinované prostředí sloužilo jako platforma pro provoz Kubernetes clusteru (k3s) a všech nasazených komponent systému, což umožnilo simulovat distribuované nasazení a provést potřebné výkonnostní testy.

3.2 Výsledky testování

Tabulka 17: Výsledky testování – Zdroj: vlastní

	TC01	TC02	TC03	TC04	TC05
Propustnost	15,00 req/s	54,29 req/s	54,29 req/s	51,23 req/s	51,23 req/s
Laten. p95	4,26 ms	71,45 ms	56,60 ms	62,07 ms	60,09 ms
Laten. p99	180,78 ms	281 ms	222,31 ms	192,8 ms	227,72 ms
Chybovost	0 %	0 %	0 %	0 %	0 %

Celkově testování systému ukazuje vysokou odolnost vůči běžné, zvýšené i nárazové zátěži – ve všech scénářích jsme udrželi požadovanou propustnost, p95 latence se pohybovala mezi 4ms a 62ms (vše hluboko pod limitem 100ms) a chybovost zůstala na 0 %. P99 latence se při maximálním zatížení jediné instance mírně překročila stanovenou hranici, nicméně již při horizontálním škálování klesla pod 223ms. Spike testy potvrdily, že náhlé špičky až na 50× průměr zvládá aplikační vrstva bez degradace služeb a s p99 pod 250ms.

Pro identifikaci dalších úzkých míst je vhodné aktivovat logování pomalých dotazů a provést profilování aplikace, čímž se přesněji určují kódem nebo databázovou vrstvou způsobené degradace výkonu. Latenci lze následně snížit prostřednictvím optimalizace SQL dotazů a indexů, zavedením cache vrstvy pro opakovaně čtená data [60] a doladěním politik auto-škálování pro navyšování či snižování kapacity podle aktuální zátěže [59].

Závěr

Navržený systém představuje základní, avšak funkční rámec, který je možné dále rozšiřovat podle potřeb produkčního nasazení. Díky použití kontejnerizace a orchestrace pomocí Kubernetes je systém připraven na horizontální škálování i provoz ve více instancích. Mezi hlavní směry dalšího rozvoje patří integrace s externími službami (platební brány, dopravci), rozšíření o notifikační systém, migrace na výkonnější databázovou platformu, implementace bezpečnostních mechanismů a pokročilých autentizačních metod.

Ačkoliv aplikace v současné podobě neobsahuje veškerou funkcionalitu plnohodnotného systému pro reálný provoz, slouží jako důkaz konceptu a demonstruje využití moderních principů při návrhu webových systémů. Díky zvolené architektuře a technologickému stacku je systém dobře připraven na budoucí úpravy a rozšíření.

Všechny hlavní cíle práce byly naplněny: podařilo se navrhnout architekturu škálovatelného systému, zvolit a zdůvodnit použití moderních technologií a zrealizovat funkční prototyp, který zvládá definované uživatelské scénáře. Testování potvrdilo, že systém je stabilní, s rychlou odezvou a připraven na provozní zatížení. Práce zároveň poskytuje přehledný metodický rámec, který může sloužit jako inspirace pro vývoj obdobných aplikací i v jiných doménách.

Použitá literatura

1. ROGERS, Dale S.; TIBBEN-LEMBKE, Ronald S. An examination of reverse logistics practices [online]. *Journal of Business Logistics*, 2001, roč. 22, č. 2, s. 129–148 [cit. 19. 4. 2025]. ISSN 0735-3766.
2. EVROPSKÝ PARLAMENT a RADA EU. Směrnice 2011/83/EU o právech spotřebitelů [online]. *Úřední věstník Evropské unie*, 2011, L 304, s. 64–88 [cit. 19. 4. 2025]. Dostupné z: https://eur-lex.europa.eu/legal-content/CS/TXT/?uri=CELEX:32011L0083
3. JÖNSSON, Patrick; LINDVALL, Magnus. Impact analysis. In: AURUM, Andreas; WOHLIN, Claes, eds. *Engineering and managing software requirements*. Berlin: Springer Science & Business Media, 2006, s. 117–142. ISBN 978-3-540-28244-0.
4. ARLOW, Jim; NEUSTADT, Ila. *UML 2 and the unified process: practical object-oriented analysis and design*. 2. vyd. Upper Saddle River: Addison-Wesley, 2005. ISBN 978-0-321-32127-5.
5. CZECHCRUNCH. Nový standard ve vrácení zboží. České Retino nabídne e-shopům zdarma své řešení pro pohodlné a expresní vratky [online]. 2020 [cit. 28. 7. 2024]. Dostupné z: https://cc.cz/novy-standard-ve-vraceni-zbozi-ceske-retino-nabidne-e-shopum-zdarma-sve-reseni-pro-pohodlne-a-expresni-vratky/
6. AMAZON WEB SERVICES. What is a web application? [online]. Seattle: Amazon Web Services [cit. 19. 4. 2025]. Dostupné z: https://aws.amazon.com/what-is/web-application/
7. MDN WEB DOCS. An overview of HTTP [online]. Mozilla, 2024 [cit. 19. 4. 2025]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
8. ABRIATA, Luciano A.; RODRIGUES, João P. G. L. M.; SALATHÉ, Marcel; PATINY, Luc. Augmenting research, education, and outreach with client-side web programming [online]. *Trends in Biotechnology*, 2018, roč. 36, č. 5, s. 473–476 [cit. 20. 4. 2025]. ISSN 0167-7799. Dostupné z: https://doi.org/10.1016/j.tibtech.2017.11.009
9. CARNIATO, Ryan. Building JavaScript frameworks to conquer eCommerce [online]. DEV Community, 14. 7. 2021 [cit. 19. 4. 2025]. Dostupné z: https://dev.to/this-is-learning/building-javascript-frameworks-to-conquer-ecommerce-3glc
10. CARNIATO, Ryan. JavaScript frameworks – heading into 2025 [online]. DEV Community, 2025 [cit. 19. 4. 2025]. Dostupné z: https://dev.to/this-is-learning/javascript-frameworks-heading-into-2025-hkb
11. TECHELOPMENT. *Web Application Architecture: Front-end, Middleware and Back-end* [online]. DEV Community, 30. 1. 2025. Dostupné z: https://dev.to/techelopment/web-application-architecture-front-end-middleware-and-back-end-2ld7 [cit. 19. 4. 2025].
12. IBM. Web server versus application server: what's the difference? [online]. 14. 10. 2021 [cit. 20. 4. 2025]. Dostupné z: https://www.ibm.com/think/topics/web-server-application-server
13. MDN WEB DOCS. Frameworks and libraries – learn web development [online]. Mozilla, 2023 [cit. 19. 4. 2025]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries
14. MDN WEB DOCS. Introduction to front-end frameworks – learn web development [online]. Mozilla Developer Network [cit. 19. 4. 2025]. Dostupné z:

https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries/Introduction

15. MESBAH, Ali; VAN DEURSEN, Arie. Migrating multi-page web applications to single-page AJAX interfaces [online]. arXiv preprint arXiv:cs/0610094, 2006 [cit. 20. 4. 2025]. Dostupné z: https://arxiv.org/abs/cs/0610094
16. SIRETEANU, Napoleon-Alexandru; HOMOCIANU, Daniel. Front-end frameworks for development of SPA and MPA web applications [online]. SSRN, 2021 [cit. 20. 4. 2025]. Dostupné z: https://ssrn.com/abstract=3987838
17. STRAPI. Client-side rendering vs server-side rendering (2025 guide) [online]. Strapi Blog, 2025 [cit. 21. 4. 2025]. Dostupné z: https://strapi.io/blog/client-side-rendering-vs-server-side-rendering
18. CARNIATO, Ryan. JavaScript frameworks – heading into 2024 [online]. DEV Community, 21. 12. 2023 [cit. 21. 4. 2025]. Dostupné z: https://dev.to/this-is-learning/javascript-frameworks-heading-into-2024-i3l
19. DEVIGRAPHICS. State of JavaScript 2024 – about [online]. 2024 [cit. 20. 4. 2025]. Dostupné z: https://2024.stateofjs.com/en-US/about/
20. RAMBEAU, Michael. 2024 JavaScript rising stars [online]. 2024 [cit. 20. 4. 2025]. Dostupné z: https://risingstars.js.org/2024/en/
21. DEVIGRAPHICS. State of CSS 2023 – about [online]. 2023 [cit. 20. 4. 2025]. Dostupné z: https://2023.stateofcss.com/en-US/about/
22. AL SALMI, H. Comparative CSS frameworks. *Multi-Knowledge Electronic Comprehensive Journal for Education and Science Publications*, 2023, roč. 66. ISSN 2616-9185
23. BELTON, Valerie; STEWART, Theodor J. Multiple criteria decision analysis: an integrated approach. Boston: Kluwer Academic Publishers, 2002. ISBN 978-1-4615-1495-4.
24. ISHIZAKA, Alessio; NEMERY, Philippe. Multi-criteria decision analysis: methods and software. Chichester: John Wiley & Sons, Ltd., 2013. ISBN 978-1-118-64489-8.
25. SAATY, Thomas L. The analytic hierarchy process: planning, priority setting, resource allocation. New York; London: McGraw-Hill International Book Co., 1980. ISBN 0-07-054371-2.
26. KRAUSE, Stefan. js-framework-benchmark [online]. GitHub, 2025 [cit. 25. 4. 2025]. Dostupné z: https://github.com/krausest/js-framework-benchmark
27. STACK OVERFLOW. 2024 Stack Overflow Developer Survey [online]. New York: Stack Exchange Inc., 2024 [cit. 26. 4. 2025]. Dostupné z: https://survey.stackoverflow.co/2024/
28. JETBRAINS. State of Developer Ecosystem Report 2024 [online]. Prague: JetBrains s.r.o., 2024 [cit. 26. 4. 2025]. Dostupné z: https://www.jetbrains.com/lp/devecosystem-2024/
29. THE BENCHMARKER. Web Frameworks Benchmark – result 2024-04-17 [online]. [s.l.]: The Benchmarker, 2024 [cit. 26. 4. 2025]. Dostupné z: https://web-frameworks-benchmark.netlify.app/result
30. TECHEMPOWER. TechEmpower Framework Benchmarks – round 23 (Fortune test) [online]. El Segundo: TechEmpower, 24. 2. 2025 [cit. 26. 4. 2025]. Dostupné z: https://www.techempower.com/benchmarks/#section=data-r23&test=fortune
31. REACT. Creating a React app [online]. Menlo Park: Meta Platforms, 2024 [cit. 27. 4. 2025]. Dostupné z: https://react.dev/learn/creating-a-react-app

32. https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/First_steps/Introduction
33. WILLIAMS, Elijah. Choose tech stack for robust and scalable web applications [online]. Medium, 3. 10. 2023 [cit. 29. 4. 2025]. Dostupné z: https://medium.com/@elijah_williams_agc/choose-tech-stack-for-robust-and-scalable-web-applications-42820bdf9d19
34. ATTRACTION CHAOS. Programming Language Benchmark v2 (plb2) [online]. [s.l.]: GitHub, 2020 [cit. 29. 4. 2025]. Dostupné z: https://github.com/attractivechaos/plb2
35. ELITEX SYSTEMS. Full-stack JavaScript development: pros, cons, and future trends [online]. červen 2024 [cit. 29. 4. 2025]. Dostupné z: https://elitex.systems/blog/full-stack-javascript-development-pros-cons-and-future-trends
36. OPENJS FOUNDATION. About Node.js [online]. [s.l.]: Node.js [cit. 29. 4. 2025]. Dostupné z: https://nodejs.org/en/about
37. DENO LAND INC. Deno [online]. [s.l.]: Deno [cit. 29. 4. 2025]. Dostupné z: https://deno.com/
38. OVEN CORP. Bun [online]. [s.l.]: Bun [cit. 29. 4. 2025]. Dostupné z: https://bun.sh
39. SOFTTECO. How to choose a tech stack for a software project [online]. 22. 3. 2022 [cit. 30. 4. 2025]. Dostupné z: https://softteco.com/blog/how-to-choose-a-tech-stack-for-software-project
40. GOOGLE CLOUD. What is a relational database? [online]. Google Cloud [cit. 29. 4. 2025]. Dostupné z: https://cloud.google.com/learn/what-is-a-relational-database
41. INSTACLUSTR. Scaling PostgreSQL: challenges, tools, and best practices [online]. Instaclustr [cit. 15. 5. 2025]. Dostupné z: https://www.instaclustr.com/education/scaling-postgresql-challenges-tools-and-best-practices/
42. AMAZON WEB SERVICES. What is an API? [online]. AWS [cit. 9. 5. 2025]. Dostupné z: https://aws.amazon.com/what-is/api/
43. MOZILLA. REST [online]. MDN Web Docs [cit. 9. 5. 2025]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/REST
44. DRIZZLE TEAM. Overview – Drizzle ORM documentation [online]. Drizzle ORM [cit. 30. 4. 2025]. Dostupné z: https://orm.drizzle.team/docs/overview
45. REACT. Build a React app from scratch [online]. React [cit. 30. 4. 2025]. Dostupné z: https://react.dev/learn/build-a-react-app-from-scratch
46. IEEE COMPUTER SOCIETY. *What is Software Architecture in Software Engineering?* [online]. IEEE Computer Society, n.d. Dostupné z: https://www.computer.org/resources/software-architecture [cit. 30. 4. 2025].
47. RICHARDS, Mark. Software architecture patterns. Sebastopol, CA: O'Reilly Media, Inc., 2015. ISBN 978-1-4493-7332-0.
48. IBM. Load balancing [online]. IBM Think [cit. 30. 4. 2025]. Dostupné z: https://www.ibm.com/think/topics/load-balancing
49. LIANG, Shichao. Chapter 1: Reliable, scalable, and maintainable applications [online]. Shichao's Notes [cit. 30. 4. 2025]. Dostupné z: https://notes.shichao.io/dda/ch1/
50. ALEXEEV, Andrew. nginx. In: BROWN, Amy; WILSON, Greg, eds. *The Architecture of Open Source Applications, Volume II: Structure, Scale, and a Few More Fearless Hacks* [online]. [s.l.]: Lulu.com, 2012, s. 211–223 [cit. 3. 5. 2025]. Dostupné z: https://aosabook.org/en/v2/nginx.html
51. BUN. Start a cluster of HTTP servers with Bun [online]. Guides. San Francisco: Bun, [cit. 3. 5. 2025]. Dostupné z: https://bun.sh/guides/http/cluster

52. POULTON, Nigel. *The Kubernetes Book*. 1. vyd. [S.l.]: s.n., 2021. ISBN 979-8703756065.
53. DOCKER INC. What is a container? [online]. San Francisco: Docker, Inc., [cit. 3. 5. 2025]. Dostupné z: https://www.docker.com/resources/what-container
54. SILES, Raul. Session Management Cheat Sheet. In: OWASP Cheat Sheet Series [online]. [s.l.]: OWASP Foundation, [cit. 3. 5. 2025]. Dostupné z: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html
55. CHARLES, Jamis. What is middleware? A simple explanation [online]. 24. 6. 2018 [cit. 3. 5. 2025]. Dostupné z: https://medium.com/@jamischarles/what-is-middleware-a-simple-explanation-bb22d6b41d01
56. Češi vrací každou desátou objednávku. E-shopy vratky přijímají bez podmínek [online]. *iDNES.cz*, 19. 5. 2021 [cit. 3. 5. 2025]. Dostupné z: https://www.idnes.cz/ekonomika/domaci/eshopy-vratky-nechtene-zbozi-objednavky.A210519_094405_ekonomika_vebe
57. ČTK; FORBES ČESKO. Alza v roce 2021 dosáhla rekordního obratu. Firma vykázala čistý zisk 2,5 miliardy [online]. *Forbes Česko*, 10. 1. 2023 [cit. 3. 5. 2025]. Dostupné z: https://forbes.cz/alza-v-roce-2021-dosahla-rekordniho-obratu-firma-vykazala-cisty-zisk-25-miliardy/
58. NIELSEN, Jakob. Response times: the 3 important limits [online]. NN/g, 1. 1. 1993 [cit. 3. 5. 2025]. Dostupné z: https://www.nngroup.com/articles/response-times-3-important-limits/
59. GOLDENBERG, Raz. Kubernetes HPA: Use Cases, Limitations & Best Practices [online]. ScaleOps, 25. 10. 2024. Dostupné z: https://scaleops.com/blog/kubernetes-hpa/ [cit. 5. 5. 2025].
60. LALCHANDANI, Darshil. *5 ways to reduce latency and improve performance for your SQL queries* [online]. Medium, 18. 8. 2024. Dostupné z: https://darshil-lalchandani.medium.com/5-ways-to-reduce-latency-and-improve-performance-for-your-sql-queries-15c3107c63b9 [cit. 5. 5. 2025].
61. PACHOT, Franck. *Distributed PostgreSQL without sharding constraint for SQL joins* [online]. DEV Community, 17. 12. 2023. Dostupné z: https://dev.to/yugabyte/distributed-postgresql-without-sharding-constraint-for-sql-joins-261i [cit. 5. 5. 2025].
62. INTERNATIONAL SOFTWARE TESTING QUALIFICATIONS BOARD. *Certified Tester – Foundation Level Syllabus v4.0.1* [online]. 15. 9. 2024. Dostupné z: https://www.istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf [cit. 5. 5. 2025].

Seznam příloh

Příloha A: Zdrojový kód projektu (SchwamD_NavrhSystemu_JP_prilohaBP_2025.zip)