Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Specifikace server-side technologií, jejich implementace a porovnání

Diplomová práce

Autor: Eva Kozáková
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové duben 2019

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a s použitím uvedené literatury.

V Hradci Králové dne 16. 4. 2019 Eva Kozáková

Poděkování:
Děkuji vedoucímu diplomové práce doc. Ing. Filipu Malému, Ph.D. za metodické vedení práce a cenné připomínky v průběhu její tvorby.

Anotace

Předložená diplomová práce nejprve pojednává o přístupu k vývoji webových aplikací se zaměřením na serverovou část. V úvodních kapitolách je popsána tato část aplikace z pohledu dílčích oblastí vývoje. Dále jsou představeny technologie pro tvorbu právě serverové části. Tyto technologie jsou Spring, Django, Node.js, Rails a Vaadin. Následuje implementace ukázkové aplikace s cílem předvést přístup jednotlivých frameworků. Jednotlivé oblasti vývoje jsou v práci popsány společně s ukázkami kódu. V dalších kapitolách jsou frameworky porovnány z obecnějšího hlediska. V závěrečné kapitole práce autorka shrnuje výhody a nevýhody každého frameworku. Na základě shrnutí je doporučen typ aplikace, na kterou je vhodné jednotlivý framework vybrat jako vývojový nástroj, a naopak, na jaký druh aplikace není vhodné vzhledem k nevýhodám framework použít.

Klíčová slova

Server-side, backend, framework, Spring, Django, Node.js, Rails, Vaadin

Annotation

Title: Specification of Server-side Technologies, their Implementation and Comparison

The beginning of this diploma thesis deals with web application development with focus on the server-side. This side of application is described from the point of partial development areas. This work introduces several technologies for server-side development. These technologies are Spring, Django, Node.js, Rails and Vaadin. This introduction is followed by implementation of example application for a purpose to show different approaches to development within each framework. Each area of implementation is described together with code examples. Then the frameworks are compared from the general point of view. Finally, the author summarizes main advantages and disadvantages of each framework including recommendation on what type of application is given framework appropriate to use.

Key words

Server-side, backend, framework, Spring, Django, Node.js, Rails, Vaadin

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Seznam použitých zkratek

API ..... Application Programming Interface
CoC ..... Convention over Configuration
CORS ..... Cross Origin Resource Sharing
CRUD ..... Create, Remove, Update, Delete
DRF ..... Djagno Rest Framework
DRY ..... Don’t Repeat Yourself
HTTP ..... Hypertext Transfer Protocol
IoC ..... Inversion of Control
MVC ..... Model–View–Controller
MVCS ..... Model–View–Controller–Service
MVT ..... Model–View–Template
MVP ..... Model–View–Presenter
ORM ..... Objectional Relational Mapping
REST ..... Representational State Transfer
RIA ..... Rich Internet Application
URI ..... Uniform Resource Identifier
URL ..... Uniform Resource Locator

1 Úvod a cíl práce

Vývoj webových aplikací prošel za poslední léta mnoha změnami. Základem ale stále zůstává n-vrstvá architektura typicky rozdělující aplikace na klientskou část (frontend) a serverovou část (backend). Pro vývoj webové aplikace existuje velké množství nástrojů a frameworků. Z toho důvodu může být občas pro vývojáře problém vybrat vhodný nástroj pro tvorbu konkrétní aplikace.

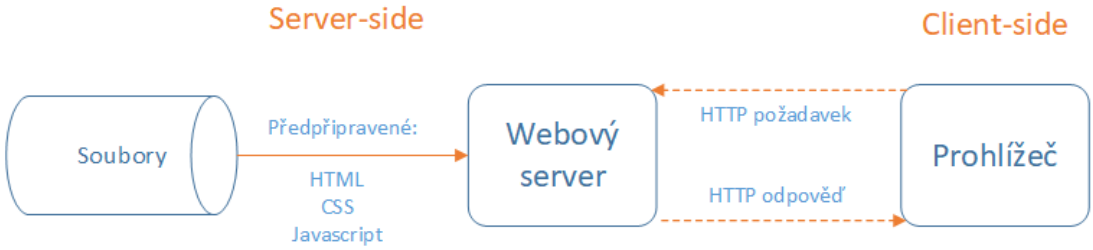
Tato diplomová práce si klade za cíl představit čtenáři oblasti vývoje serverové části aplikace a představit vybrané frameworky pro její tvorbu. Frameworky byly vybrány s ohledem na různorodosti přístupu k vývoji a architektuře. Jako hlavní architektura byl sice zvolen návrh MVC, ale každý z frameworků má jeho vlastní modifikaci. Jedná se o frameworky: Spring v jazyce Java, Django v jazyce Python, Node.js (Express.js) v jazyce JavaScript, Ruby on Rails v jazyce Ruby a Vaadin v jazyce Java. Dále bude implementována ukázková aplikace v daných frameworcích za účelem představit přístupy jednotlivých frameworků. Za tímto účelem bude vytvořena i frontendová aplikace, se kterou bude většina backendových aplikací komunikovat přes rozhraní REST API. Výjimkou je framework Vaadin, který ze své podstaty nepodporuje vývoj pomocí REST API a celý vývoj aplikace probíhá na straně serveru.

Následovat bude srovnání frameworků z hlediska obecnějších charakteristik, než je implementace. Mezi porovnávané oblastí patří dokumentace a podpora, rychlost učení, testování, zabezpečení a rozšiřitelnost.

V poslední kapitole práce budou zdůrazněny výhody a nevýhody jednotlivých frameworků společně s doporučením, na jaký druh aplikace je a není vhodné určitý framework vybrat jako vývojový nástroj.

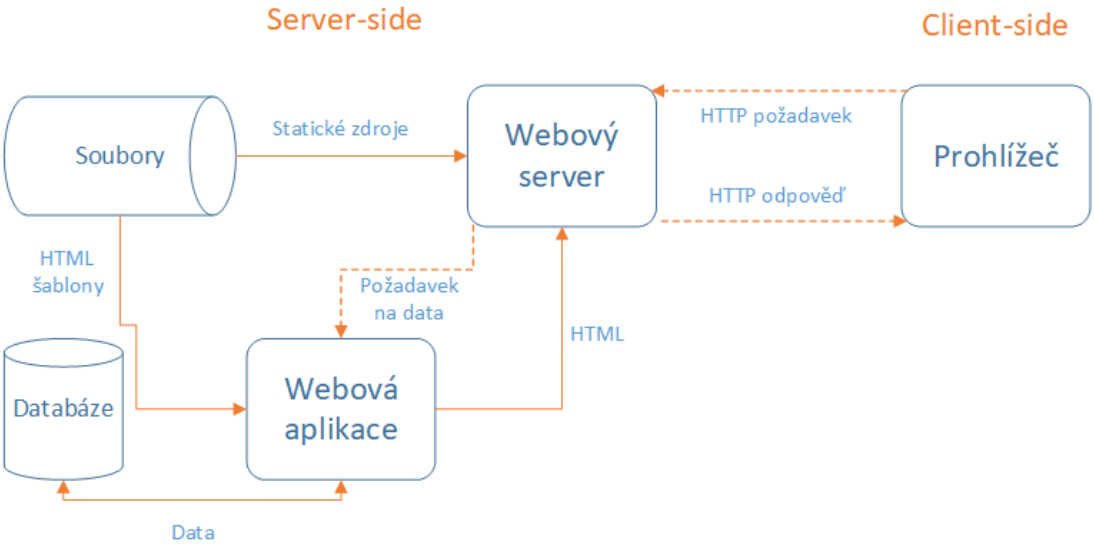
2 Vývoj webové aplikace

Z historického hlediska dělíme weby na dvě kategorie, weby statické a dynamické. Statický web se skládá z jednotlivých neměnných (statických) stránek, přesněji HTML souborů. Server pokaždé, kdy je o danou stránku požádáno, pouze vrátí příslušnou stránku, včetně ve stránce odkazovaných CSS stylů, skriptů, obrázků apod. Prohlížeč následně soubory zpracuje a požadovanou stránku vykreslí.



Obrázek 1: Architektura statického webu
Zdroj: vlastní zpracování

Oproti tomu dynamický web, resp. webová aplikace, generuje obsah v případě potřeby (dynamicky). Ve webové aplikaci jsou výsledné HTML stránky typicky vytvářeny pomocí načítání dat z databáze a jejich následného vkládání do připravených HTML šablon. Dynamická stránka může vracet různorodá data na základě požadavků uživatele. Většina kódu zodpovědného za chod dynamického webu musí běžet na serveru. Prohlížeč pošle požadavek na server, ten požadavek zpracuje a následně vrátí odpovídající HTTP odpověď. Podrobněji je tento proces popsán v kapitole 5 [1].



Obrázek 2: Architektura dynamického webu
Zdroj: vlastní zpracování

Moderní webové aplikace jsou vyvíjeny za použití různorodých technologií a principů. Dnes používané objektově orientované programování je přístup, pomocí kterého jsou neperzistentní data vhodně strukturována. Databázové systémy slouží ke strukturování perzistentních dat a webové aplikační frameworky pomáhají ke značnému zefektivnění samotného vývoje aplikací [2].

2.1 Backend vs. Frontend

Frontend, neboli kód na straně klienta, je kód běžící v prohlížeči a primárně slouží k vylepšení vzhledu a chování vykreslované webové stránky. Naproti tomu backend, neboli kód na straně serveru, zahrnuje rozhodování, jaký obsah se má v prohlížeči zobrazit. Backend zpracovává úkoly jako je validace přijatých dat a požadavků, využívá databázi k ukládání a načítání dat a v neposlední řadě se stará o posílání dat klientovi dle požadavku.

Frontend je typicky vyvíjen za pomocí HTML, CSS a JavaScriptu. Tyto technologie běží uvnitř webového prohlížeče a mají velmi malý nebo vůbec žádný přístup k operačnímu systému.

Backend může být vyvíjen v různých technologiích, které budou podrobněji popsány v další části této práce. Kód na straně serveru má plný přístup k operačnímu systému serveru. Typické pro vývoj backendu je využití webových frameworků. Jedná se o kolekci funkcí, objektů, pravidel a jiných konstrukcí vytvořených k řešení běžných problémů, k urychlení vývoje, či ke zjednodušení různých typů úloh vyskytujících se v určitých doménách [1].

2.2 Vícevrstvá architektura

Typická webová aplikace se dělí na tři základní vrstvy, kde každá vrstva má svoji specifickou zodpovědnost.

Prezentační vrstva umožňuje uživateli přístup k datům, především pomocí grafického uživatelského rozhraní. Stará se o uživatelské vstupy a zároveň zprostředkovává prezentaci dat směrem k uživatelům.

Vrstva aplikační logiky, též nazývána jako businessová vrstva, má za úkol zpracovávat uživatelské příkazy a logická rozhodnutí. Tato vrstva je stěžejní vrstvou aplikace, implementující její základní funkcionality.

Datová vrstva využívá ke své činnosti databázový systém. Jedná se o nejnižší vrstvu architektury a zajišťuje perzistentní ukládání dat, případně jejich vyhledávání a načítání [3].

2.2.1 Návrhové vzory

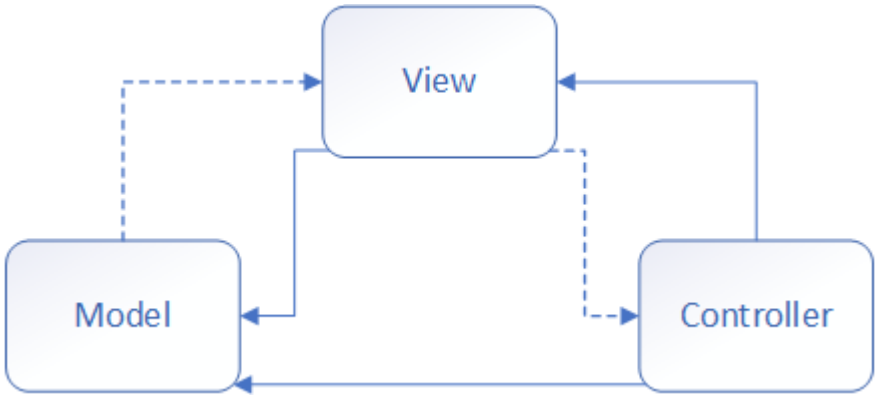
Na principu popsaném výše fungují mnohé vzory pro návrh architektury webových aplikací. Nedrží se ovšem striktně jen tří vrstev, ale často se jednotlivé vrstvy dále dělí. Mezi nejpoužívanější vzory patří Model-View-Controller a Model-View-Presenter. Tato práce se bude zabývat i návrhovým vzorem Model-View-Template. Každá ze zmíněných architektur prakticky vychází z MVC, ale každá architektura má odlišné rozdělení závislostí a zodpovědností.

2.2.2 Model-View-Controller

Návrhový vzor MVC dělí aplikaci na tři hlavní komponenty.

- Model – datová struktura a business logika
- View – zobrazení výstupu
- Controller – reaguje na události

Základní myšlenka je taková, že požadavky od uživatele jsou směrovány na Controller, který je zodpovědný za spolupráci s Modelem za účelem zpracovat uživatelské akce a případně vrátit výsledky dotazů. Controller navíc vybírá, jaké View bude uživateli zobrazeno spolu s příslušnými daty poskytnutými Modelem. Obecně platí, že View nemá přímou vazbu na Controller. Model reprezentuje datovou strukturu a zároveň slouží jako doménový model implementující business logiku. Často se mezi Controller a Model implementuje servisní vrstva sloužící k napojení na externí komponenty nebo systémy. V takovém případě se jedná o architekturu nazývanou MVCS.
Diagram níže ukazuje komponenty v jejich vzájemné spolupráci [4].



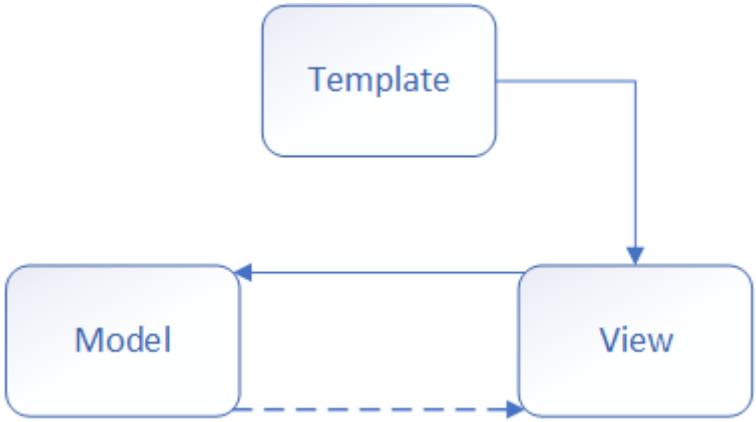
Obrázek 3: Architektura MVC
Zdroj: vlastní zpracování

2.2.3 Model-View-Template

Architektura MVT vychází z MVC, částečně se však liší funkcemi jednotlivých komponent. Ty jsou následující:

- Model – datová struktura
- View – business logika
- Template – zobrazení výstupu

V MVT návrhu platí, že Model reprezentuje vrstvu starající se o přístup k datům, podobně jako v MVC architektuře. Odlišná implementace nastává v komponentách View a Template. View obsahuje logiku, která se stará o přístup k Modelu a zároveň směruje přístup ke vhodnému Template. Template v MVT architektuře zastává prezentační vrstvu a určuje způsob, jakým bude obsah zobrazen na webové stránce. Tedy má podobné funkce jako View v návrhovém vzoru MVC [5].



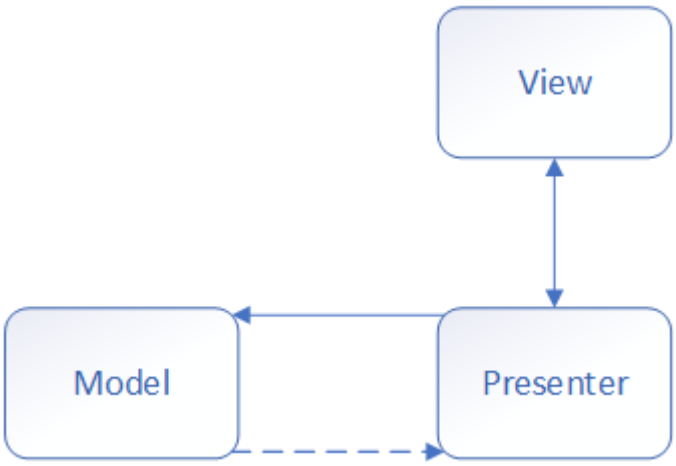
Obrázek 4: Architektura MVT
Zdroj: vlastní zpracování

2.2.4 Model-View-Presenter

Návrhový vzor Model-View-Presenter také vychází z MVC architektury. Komponenty jsou následující:

- Model – datová struktura a business logika
- View – zpracování uživatelského vstupu
- Presenter – aplikační a prezentační logika

Místo Controlleru je zde implementován Presenter, který řídí View skrze své rozhraní. View má přímou vazbu na Presenter. Ten obsahuje aplikační a prezentační logiku. View přímo s Modelem nekomunikuje. Vstupní data jsou směrována nejdříve na View, což je hlavní rozdíl oproti MVC. View však pouze deleguje uživatelské akce na Presenter. Tímto způsobem dochází k přesnějšímu oddělní zodpovědností a umožňuje lehčí testovatelnost Presenteru a Modelu [6].



Obrázek 5: Architektura MVP
Zdroj: vlastní zpracování

3 Server

Server je zařízení nebo počítačový program přijímající nebo odpovídající na požadavky vysílané jiným programem, známým jako klient. Existují dva druhy serverů, webový server a aplikační server. Jejich činnost nemá striktně vymezené hranice a společně jsou zodpovědné za správu síťových zdrojů a za běh programů nebo softwaru poskytujícího služby [7].

3.1 Webový server

Webový prohlížeč komunikuje s webovým serverem pomocí HTTP protokolu. Pokaždé, když uživatel klikne na odkaz, potvrdí formulář nebo provádí vyhledávání, je odeslán HTTP požadavek (request) z prohlížeče na server.

Požadavek obsahuje URL adresu, která identifikuje zdroj, a metodu, co definuje požadovanou akci (například GET, DELETE, POST) a může obsahovat dodatečné informace zakódované v URL parametrech.

Webový server čeká na zmíněné požadavky, zpracovává je a odpovídá prohlížeči pomocí HTTP odpovědí (response). Odpověď obsahuje stav indikující, zda požadavek proběhl úspěšně. Seznam stavů je znázorněn v tabulce 1.

Tělo úspěšné odpovědi obsahuje požadovaný zdroj (například nová HTML stránka), který může být zobrazen prohlížečem [1].

KATEGORIE	POPIS
1xx: Informace	
2xx: Úspěch	Požadavek klienta byl úspěšně přijat.
3xx: Přesměrování	Je potřeba další klientské akce k dokončení požadavku.
4xx: Chyba	Indikuje chybu na straně klienta.
5xx: Chyba	Indikuje chybu na straně serveru.

Tabulka 1: HTTP kódy
Zdroj: [8]

3.2 Aplikační server

Aplikační server v typické třívrstvé architektuře systému zajišťuje chod vrstvy mezi operačním systémem a aplikací. Poskytuje základní funkce programům, podobně

jako operační systém (například správa procesů nebo přístup k souborovému systému).

3.3 Jak spolu servery pracují?

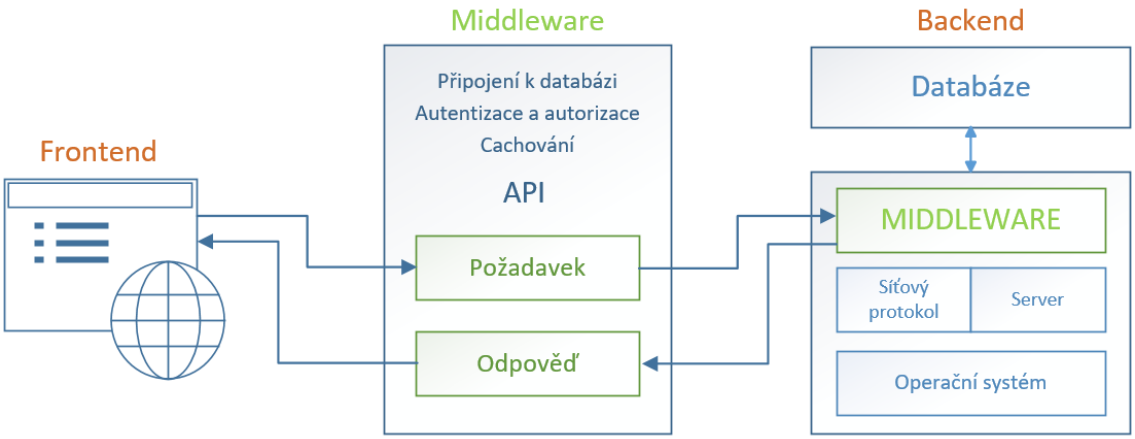
V typickém případě nasazení webové aplikace je webový server využíván pro generování statického obsahu a aplikační server pro generování dynamického obsahu. Programy zajišťující správné směrování toku dat mezi servery jsou reverzní proxy server a load balancer.

- Reverzní proxy server – Přijímá požadavek od klienta, předá ho aplikačnímu serveru a vrátí odpověď klientovi. Poskytuje dodatečnou úroveň abstrakce zajišťující plynulý chod síťového provozu mezi klientem a serverem.
- Load balancer – Distribuuje požadavky klienta mezi skupinu aplikačních serverů a pro každý server vrátí odpověď vhodnému klientovi.

Funkcionalita těchto komponent může na první pohled vypadat podobně, obě se nacházejí mezi klientem a serverem. Od prvního vracejí požadavky a posílají odpověď druhému. Rozdíl je v jejich podmínkách nasazení. Load balancer se nasazuje v případě, kdy aplikace potřebuje pro svůj provoz několik serverů, protože množství požadavků je příliš velké na to, aby jej jeden server zvládl zpracovat efektivně. Load balancer navíc dokáže detekovat přetížení serveru a přesměrovat požadavek na méně vytížený server. Oproti tomu je reverzní proxy server vhodné nasazovat v případě potřeby pouze jednoho webového nebo aplikačního serveru. Výhodou je například rychlejší vyřízení požadavku mezi klientem a serverem [7].

4 Middleware

Middleware je označení pro tu část aplikace, která se stará o komunikaci a správu dat distribuovaných aplikací. Jedná se v podstatě o skrytou vrstvu, často označovanou jako „*potrubí*“ (pipeline) nebo „*lepidlo*“ (glue). Důvodem je využití této vrstvy pro spojení jakékoliv komunikace (požadavky a odpovědi) mezi samotnou aplikací a serverem či databázovým systémem. Na obrázku 6 je v části backend znázorněno postavení této vrstvy mezi ostatními komponentami. Uprostřed obrázku je pak v detailu zobrazena podrobnější funkcionalita.



Obrázek 6: Middleware workflow
Zdroj: vlastní zpracování

Samotný middleware může být rozdělen do více vrstev. Důležitou vrstvou je web API, které tvoří „most“ mezi aplikační a prezentační vrstvou.

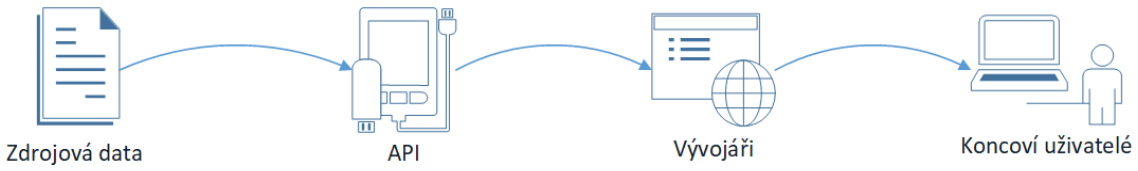
4.1 API

S tím, jak stále narůstá počet uživatelů a aplikací, stoupá i poptávka po větších datových úložištích, schopných pojmout zvyšující se objem dat. Zdrojem těchto dat mohou být například obrázky, videa, historie komunikace jako například emaily nebo informace získané z uživatelských profilů.

S takovým rozvojem se stala populární takzvaná cloudová úložiště. Data uživatelů jsou uložena na serveru, který je přístupný přes internet.

Je běžné, že uživatelé zpřístupní svá data jiným uživatelům. Různí uživatelé tak můžu prohlížet, sledovat nebo dokonce modifikovat stejný zdroj dat. Zjednodušeně lze říci, že API (Application Programming Interface) je rozhraní sloužící ke sdílení a

ochraně dat a informací. Díky API spolu mohou aplikace vzájemně komunikovat a sdílet informace.



Obrázek 7: API workflow
Zdroj: vlastní zpracování

Klasická webová aplikace používá URL adresu k zasílání požadavku na server a jako odpověď se vrací webová stránka, která se zobrazí v prohlížeči. API tento proces zjednodušuje. API umožňuje vývojářům či aplikacím přístup do databáze a služeb. V podstatě se chová jako konektor poskytující standardní sadu instrukcí k přístupu k datům [9].

4.1.1 REST API

Representational State Transfer (REST) je styl pro návrh architektury distribuovaných systémů. Poprvé byl zmíněn v disertační práci Roye Fieldinga z roku 2000 [10].

Aby byl návrh považovaný za RESTful, měl by splňovat 6 následujících podmínek [11]:

- **Client-Server:** Oddělení klientské a serverové části, čímž je umožněna přenositelnost mezi platformami a lepší škálovatelnost.
- **Stateless:** Každý klientem odeslaný požadavek musí obsahovat všechny informace k porozumění požadavku. Nemělo by se tedy využívat dat uložených na serveru.
- **Cacheable:** Je možnost explicitně označit, zda data obsažená v odpovědi lze nebo nelze cachovat.
- **Uniform interface:** Jednotné rozhraní je hlavní klíč ke zjednodušení a zefektivnění komunikace mezi klientem a serverem pomocí standardizovaných postupů.
- **Layered system**: Architektura je rozdělena do několika samostatných vrstev.

- **Code on demand**: Možnost rozšířit funkcionalitu na straně klienta o kód zaslaný serverem.

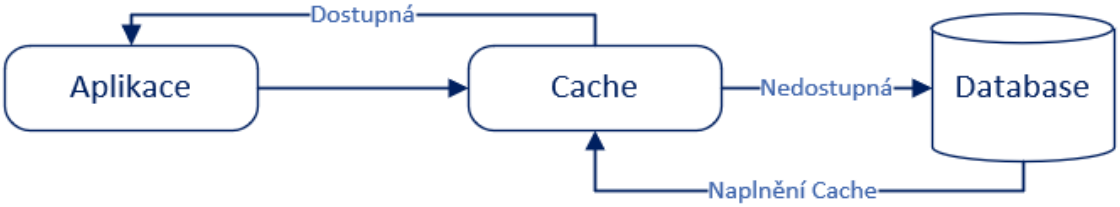
ZDROJ

REST je používaný pro jednotný a snadný přístup ke zdrojům (resources). Zdrojem může být prakticky cokoliv, co dokážeme pojmenovat. Například dokument, obrázek nebo dočasná služba. Reprezentace zdroje se stává z dat, metadat a odkazů na jiná hypermedia.

Datový formát pro hypermedia je typicky JSON, případně HTML nebo XML, ale obecně se může jednat o libovolný formát, jakému je počítač schopen porozumět [11]. Dalším důležitým faktorem spojovaným s REST návrhem jsou metody používané pro vykonání požadovaného úkolu. I přesto, že se Roy Fielding [10] ve své disertační práci nezmiňuje o konkrétních metodách, pro mnoho vývojářů se staly standardem HTTP metody sloužící k mapovaní CRUD operací na HTTP požadavky. Jedná se o metody GET, POST, PUT, DELETE. [12]

4.2 Cachování

Moderní webové aplikace spotřebují obrovské množství dat. Díky cachování je možné zefektivnit výkon tím, že se do dočasné paměti uloží data, která jsou požadována častěji. Díky tomu se zredukují dotazy nad databází. Cache má roli dočasného úložiště a slouží jako první místo, kam se aplikace podívá pří hledání dat. Fungování tohoto procesu je graficky znázorněno na následujícím schématu:



Obrázek 8: Cache workflow
Zdroj: vlastní zpracování

Pokud cache existuje, aplikace vyhledá data právě ve vyrovnávací paměti a ta vrátí data aplikaci. Pokud cache neexistuje, aplikace vyhledá data v databázi a těmito novými daty se naplní vyrovnávací paměť [13].

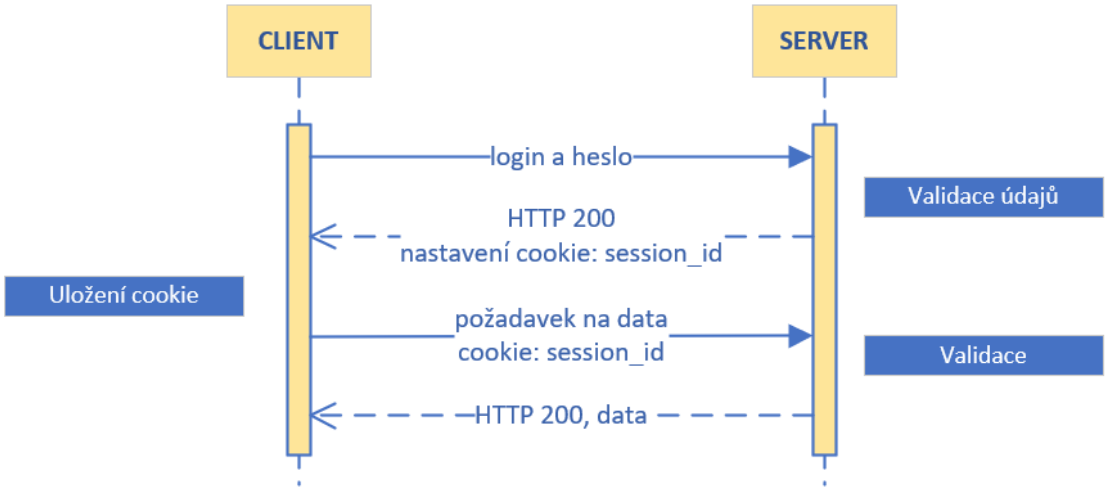
4.3 Autentizace a autorizace

Při vývoji webových aplikací je bezpečnost jednou z klíčových oblastí. Základem je pak autentizace a autorizace uživatelů. Autentizace může být definována jako proces ověřování identity na základě poskytnutých údajů (typicky uživatelské jméno, email a heslo). Autorizace je proces povolení přístupu autentizovaného uživatele k určitým datům. Na straně serveru existuje několik způsobů autentizace.

Autentizace pomocí cookies

Tento typ autentizace je považován za stavový (stateful). To znamená, že největší část se odehrává na straně serveru. Server udržuje detaily o aktivní *session* a na frontendu se vytvoří *cookie*, která drží ID této *session*. Celý proces probíhá následovně [14]:

- Uživatel zadá přihlašovací údaje
- Server ověří dané údaje, vytvoří session a uloží ji do databáze.
- Na straně klienta se vytvoří cookie držící session ID
- Pro následující požadavky je session ID certifikována proti databázi
- Session je zničena zároveň z backendu i frontendu v případě, že se uživatel odhlásí

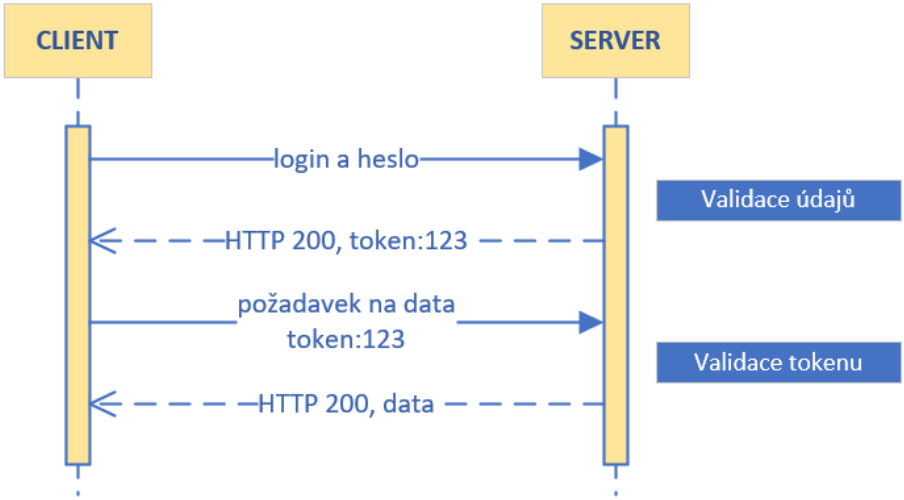


Obrázek 9: Autentizace pomocí cookie
Zdroj: vlastní zpracování

Autentizace pomocí tokenu

Oproti autentizaci pomocí cookie je autentizace pomocí tokenu považována za bezstavovou (stateless). To znamená, že server neuchovává žádné informace o přihlášeném uživateli. Ty jsou uloženy na straně klienta. Proces autentizace pomocí tokenu probíhá následovně [14]

- Uživatel zadá přihlašovací údaje
- Server ověří dané údaje a vrátí podepsaný token
- Token je uložen na frontendu
- Následující požadavek bude odeslán s tokenem v HTTP hlavičce
- Server ověří token a vrátí požadovaná data
- Token je zničen v případě, že se uživatel odhlásí



Obrázek 10: Autentizace pomocí tokenu
Zdroj: vlastní zpracování

5 Databázový systém

Dnes se databázový systém definuje jako software schopný spravovat data, která jsou objemná (počtem i velikostí), sdílená a perzistentní. Pomocí databázového systému je také zajištěna bezpečnost dat. Stejně jako jiné systémy, i databázový systém by měl být výkonný a efektivní. Samotná databáze je kolekce dat spravovaných právě databázovým systémem [15].

5.1 SQL vs. NoSQL databáze

Nejpoužívanějšími databázovými systémy jsou databáze relační (dále SQL) a nerelační (NoSQL). Database Management System (DBMS) se zabývá tím, jak jsou data uložena, jak je s nimi manipulováno a jak jsou data udržována.

Relační databáze se vyznačuje tím, že data jsou reprezentována pomocí tabulek, sloupců a řádků. Tabulka (relace) je chápána jako kolekce objektů stejného typu (řádky). Relační databáze používají jazyk SQL (Structured Query Language) pro práci se samotnými daty [15].

Mezi nejznámější relační databáze patří [16]:

- MySQL
- MS SQL Server
- PostgreSQL

NoSQL znamená „not only SQL“ (nejen SQL) nebo přímo „non SQL“ (bez SQL) ve smyslu nerelační databáze. NoSQL databáze jsou rozmanitější – na rozdíl od SQL databází nemají zcela jasnou specifikaci. Poprvé se NoSQL databáze začaly využívat ke konci 20. století a oblibu získaly zvláště díky tomu, že přinesly jednodušší škálovatelnost a vyšší výkon oproti tradičnějším SQL databázím. Díky těmto vlastnostem jsou NoSQL databáze vhodné pro práci s velkými objemy dat, především s Big Data [17]. Další velkou výhodou NoSQL databází je poměrně snadné horizontální škálování (na rozdíl od SQL databází, kde se využívá primárně vertikální škálování).

Nejznámější kategorie NoSQL databází mohou být klasifikovány podle použitého datového modelu [16]:

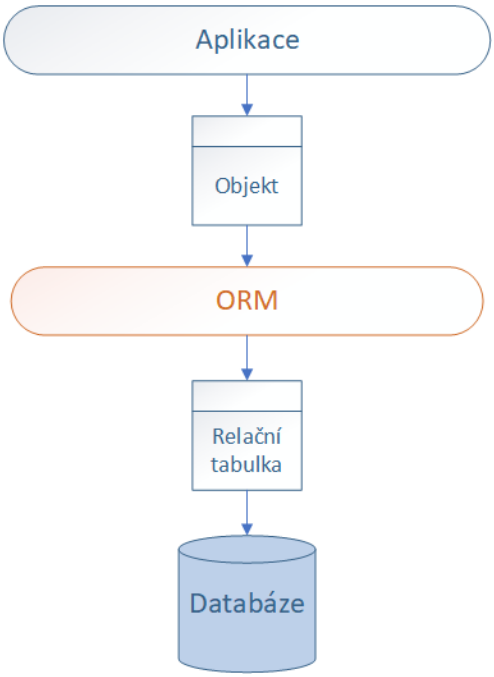
- Ukládání podle principu klíč-hodnota (Např. Redis)
- Ukládání do sloupců (Např. CASSANDRA)
- Ukládání do dokumentu (Např. MongoDB)

V této práci bude pro ukázkovou aplikaci použit MySQL server jako relační databázový systém.

5.2 Objektově relační mapování

Základem každé aplikace programované v objektově orientovaných jazycích jsou objekty. Mezi těmito objekty a „objekty" v relačních databázích je však rozdíl. Mají rozdílné datové typy a odlišné organizační principy.

Jako řešení je možné vytvořit v aplikaci rozhraní mezi těmito paradigmaty. Takové rozhraní se v rámci webových frameworků nazývá objektově relační mapování (ORM). ORM mapuje data z databáze na objekty vytvořené v objektově orientovaném jazyce. Tím se z databázového dotazu vytvoří zjednodušený a výkonný dotaz bez toho, aniž by byl nutný zásah do kódu aplikace. [2]



Obrázek 11: ORM v rámci webové aplikace
Zdroj: vlastní zpracování

6 Jazyky a frameworky

Server-side frameworky jsou nástroje ulehčující psaní, údržbu a škálovatelnost webových aplikací. Poskytují knihovny, které zjednodušují běžné úkoly. Ty zahrnují například interakci s databází, uživatelskou autentizaci a autorizaci, formátování výstupu (například JSON, XML) a v neposlední řadě zajišťují bezpečnost webových aplikací [18].
Během návrhu aplikace je potřeba se rozhodnout, jestli pro tvorbu aplikace použít kompilovaný nebo interpretovaný jazyk.

6.1 Kompilované

Zdrojový kód psaný v jazycích jako jsou například Java nebo C je překládán skrze kompilátor. Tím je dosaženo velké efektivity kódu. Samotný překlad probíhá pouze jednou, při opakovaném spuštění již zdrojový kód nemusí být překládán, pouze se načte a spustí [19].

6.1.1 Java

Poprvé vyšel objektově orientovaný programovací jazyk Java v roce 1996. Od té doby není jen programovacím jazykem, ale i obrovskou platformou se značným množstvím knihoven a prostředím, které poskytuje služby jakou jsou bezpečnost, přenositelnost mezi různými operačními systémy a automatický garbage collector [20] .

Spring

Spring framework (a jeho první verze Spring 0.9) vychází z knihy *Expert One-on-One: J2EE Design and Development* od Roda Johnsona [21]. Za poslední dekádu se Spring dramaticky změnil jak ve funkcionalitách, tak v podpoře komunity. Poslední vydanou verzí je Spring Framework 5.0 [22]

Vaadin

Vaadin je relativně nový webový framework implementující architekturu Model-View-Presenter. Jedná se o framework převážně se zaměřující na vývoj na straně serveru, a to i z pohledu uživatelského rozhraní. Vaadin totiž umožňuje psát webové

komponenty na straně serveru, takže programátor nemusí nutně znát frontendové technologie. Poslední verze frameworku je Vaadin 13 (Vaadin Flow) z roku 2018 [6].

6.2 Skriptovací

Zdrojový kód interpretovaných jazyků jako například JavaScript, Python nebo Ruby, musí být parsován, interpretován a spuštěn pokaždé, když se spustí program. Tímto se značně zvýší cena za spuštění (běh) programu. Z tohoto důvodu jsou skriptovací jazyky většinou méně efektivní než kompilované jazyky [19].

6.2.1 Python

Python je interpretovaný, interaktivní a objektově orientovaný programovací jazyk. Kombinuje vysoký výkon a velice jasný syntax. Python je, stejně jako Java, přenosný mezi operačními systémy. Od verze 2.1 je spravován Python Software Foundation [23].

Django

Django je open-source webový framework, který následuje principy MVT (Model-View-Template) architektury. Primárním cílem frameworku Django je vytváření komplexních webových aplikací řízených databázemi, jejichž vývoj se vyznačuje především znuvupoužitelností, jednoduchostí kódu a nízkou závislostí. Django klade důraz na princip DRY. Tato zkratka stojí za frází „Don't repeat yourself", česky „Neopakuj se" [24].

6.2.2 Ruby

Ruby jako programovací jazyk byl poprvé vydán v roce 1995. Ruby se snaží kombinovat prvky funkcionálního a imperativního programování. Je objektově orientovaný – vše v Ruby je chápáno jako objekt. Zatím poslední stabilní verze Ruby je 2.5.3 [25].

Ruby On Rails

Rails je aplikační framework jehož první verze vznikla v roce 2004 (verze 1.0). Filozofie frameworku Rails je postavena na dvou základních principech: DRY a Convention over Configuration, volně přeloženo jako „konvence je důležitější než

konfigurace“. To znamená, že Rails sám ví, jaký způsob je nejlepší pro implementaci dané části webové aplikace a z těchto konvencí vychází například při přístupu do databáze. Vývojář je tak oproštěn od mnohdy zbytečného nastavování konfiguračních souborů. Poslední verze Rails je verze 5.2.1 z roku 2018 [26].

6.2.3 JavaScript

JavaScript (často se používá zkratka JS) je interpretovaný, objektově orientovaný programovací jazyk. Nejznámější je pro své využití jako skriptovací jazyk pro frontend, nicméně s příchodem Node.js je i hojně používaný pro tvorbu serverové části webových aplikací. Standardem pro JavaScript je ECMAScript. Od roku 2012 plně podporují všechny prohlížeče verzi 5.1. Od roku 2015 je ECMAScript pravidelně vydáván v majoritních verzích každý rok. Nejnovější plná verze ECMAScriptu je ECMAScript 2018 [27].

Node.js

Node.js je platforma postavena na Chrome JavaScript runtime. Pomocí Node.js je možné vytvořit rychlé a škálovatelné aplikace. Node.js využívá model událostí a asynchronní I/O model pro efektivní vývoj především serverové části webových aplikací [28].

7 Implementace

V následující části bude implementována webová aplikace pomocí technologií zmiňovaných v předchozím textu. Celkem bude vytvořeno pět backendových aplikací a jedna frontendová aplikace. Komunikace mezi backendem a frontendem bude probíhat na základě REST API. Výjimkou bude implementace ve frameworku Vaadin, kde vzhledem k jeho vlastnostem bude vyvíjena aplikace specifickým způsobem a nebude využívat zvlášť vyvinutou klientskou část, která je jinak společná pro ostatní backendové aplikace. Jako databáze byla vybrána MySQL databáze.
V rámci implementace budou znázorněny postupy v následujících oblastech:

- **Instalace a základní konfigurace** – vytvoření základní kostry aplikace a nastavení základní konfigurace
- **Připojení do databáze** – nastavení připojení k MySQL databázi a tvorba schématu
- **Architektura** – popsání principů, podle kterých daný framework funguje a zpracovává uživatelské požadavky
- **Zabezpečení** – ukázka autentizace uživatele
- **Cachování** – ukázka možnosti cachování v daném frameworku
- **Testování** – ukázka možností testování v daném frameworku

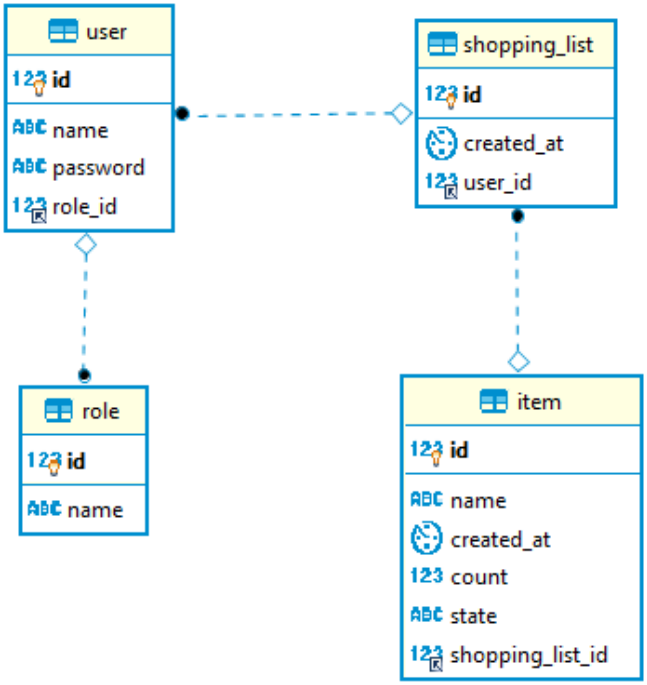
Frontend

Klientská část bude implementována v jazyce React, což je javascriptová knihovna pro vytváření uživatelského rozhraní pomocí znovupoužitelných UI komponent. V obecné MV* architektuře lze chápat aplikaci v Reactu jako View, tedy v podstatě prezentační vrstvu [29].

Návrh aplikace

Jako ukázková aplikace bude implementován nákupní lístek. Doménový model obsahuje 4 hlavní entity: uživatel, role, nákupní lístek a položka. Uživatel má možnost vytvořit si vlastní účet. Poté si může vytvořit jednotlivé položky nákupního

lístku a následně může lístek uložit. Na hlavní stránce se uživateli zobrazují pouze nákupní lístky jím vytvořené.



Obrázek 12: Databázové schéma
Zdroj: vlastní zpracování

Mezi rolí a uživatelem panuje vztah 1:N ze strany role. Jeden uživatel může vytvořit několik nákupních lístků, zatímco konkrétní lístek může patřit právě jednomu uživateli, mezi těmito entitami tedy panuje vztah také 1:N ze strany uživatele. Podobný vztah funguje mezi nákupním lístkem a položkami, kde je také vztah 1:N ze strany nákupního lístku.

Tato aplikace byla navržena za účelem porovnání technologií k tvorbě backendu. V následující kapitole budou popsány vybrané aspekty implementace v rámci každé technologie.

7.1 Spring

Backendová aplikace vytvořená ve frameworku Spring (verze 4.3) je psaná v jazyce Java (verze 8) a využívá principy MVC architektury s využitím REST API pro komunikaci s frontendem. Jako vývojové prostředí byl vybrán software IntelliJ IDEA od společnosti JetBrains.

7.1.1 Instalace a základní konfigurace

K založení nové aplikace byl použit nástroj Spring Boot. Jedná se o nástroj pro rychlé a jednoduché vytvoření základního modulu springovské aplikace. Na stránce https://start.spring.io/ si programátor může přednastavit nástroj pro řízení a správu buildů aplikací (Maven), jazyk (Java) a verzi Spring Boot (v době vytváření aplikace 1.5.8) Dále může zvolit základní závislosti na moduly jako je například Spring MVC nebo Spring Security a projekt jednoduše vygenerovat. Spring Boot v základu využívá aplikační server Apache Tomcat. Spouštěcí třída vypadá následovně:

```
@SpringBootApplication
public class ShoppingListApplication {

   public static void main(String[] args) {
      SpringApplication.run(ShoppingListApplication.class, args);
   }

}
```

Ukázka kódu 1: Spring Boot Application
Zdroj: vlastní zpracování

Spring Boot je v podstatě založený na konfiguraci pomocí anotací. Například anotace *@SpringBootApplication* je ekvivalent pro použití tří základních konfigurací:

- *@Enable Autoconfiguration* – zajišťuje auto konfiguraci na základě jar závislostí
- *@ComponentScan* – zajišťuje automatické rozpoznávání springovských komponent
- *@Configuration* – zajišťuje dodatečné přidání Beans nebo konfiguračních tříd v rámci daného kontextu [30]

7.1.2 Balíčková Struktura

```
ShoppingList/
    src/
     main/
        java/
           configuration
           controller
           model
           repository
           service
           ShoppingListApplication.java
        resources/
           application.properties
        test
    pom.xml
```

Ukázka kódu 2: Spring – Balíčková struktura
Zdroj: vlastní zpracování

- balíček `java/` obsahuje všechny soubory zdrojového kódu rozděleného do dalších balíčků podle funkcionality
- balíček `resource/s` obsahuje soubor *application.properties*, který obsahuje informace potřebné k připojení do databáze
- balíček test obsahuje testovací soubory
- soubor *pom.xml* je konfigurační soubor Mavenu obsahující informace o závislostech na jednotlivých modulech projektu. Závislosti jsou reprezentovány ve formátu XML.

7.1.3 Připojení do databáze

Spring Boot má v základu nastavené připojení do defaultní databáze H2. Proto je potřeba nakonfigurovat vlastní databázové připojení pomocí souboru *application.properties*

```
spring.datasource.url=jdbc:mysql://localhost:3306/shopping_list
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
```

Ukázka kódu 3: Spring – application.properties
Zdroj: vlastní zpracování

K mapování objektů aplikace na relační tabulky byl vybrán nástroj *Spring Data JPA,* který vnitřně využívá ORM framework *Hibernate*. Kromě závislosti do *pom.xml* je potřeba vytvořit modelovou třídu s anotací *@Entity*, pomocí které je automaticky třída přeložena jako databázová tabulka. *@Table* určuje název tabulky a jméno databázového schéma, do kterého bude tabulka zapsána. *@Id* a *@GeneratedValue* adresuje primární klíč, který se bude automaticky inkrementovat po přidání dalšího záznamu do tabulky.

```
@Entity
@Indexed
@Table (name = "items", schema = "demo")
public class Item {

    @Id
    @GeneratedValue
    private int id;
    ...
```

Ukázka kódu 4: Spring – Definování entity
Zdroj: vlastní zpracování

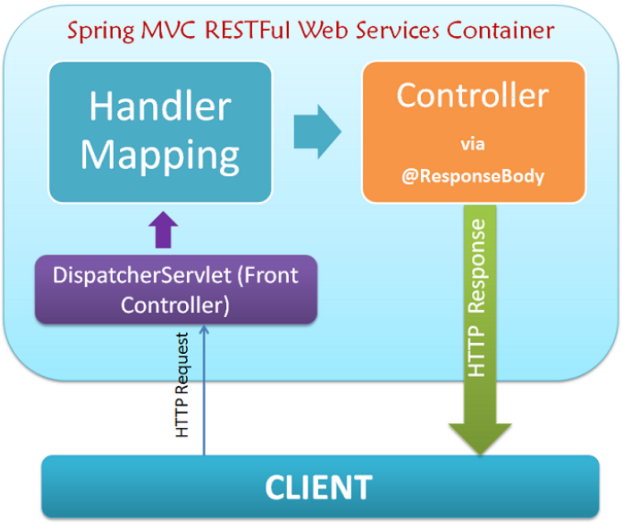
Dalším krokem je vytvoření vrstvy, která bude s databází přímo komunikovat. Tato vrstva je anotována pomocí *@Repository*. V rámci Spring Data JPA stačí implementovat *JpaRepository*. Díky tomuto kroku již není potřeba vlastní implementace dotazů do databáze, protože framework si sám odvodí, jakou operaci provést. Buď z defaultních metod (*save, delete*) nebo s vlastních metod (*findAllByShoppingListId*), kde je v názvu definován atribut, podle kterého je položka vyhledána [22].

```
@Repository
public interface ItemRepository extends JpaRepository<Item, Long> {
    Optional<Item> findById(int id);

    List<Item> findAllByShoppingListId(int id);

    void delete(Item item);

    Item save(Item item);
}
```

Ukázka kódu 5: Spring – Definování repository
Zdroj: vlastní zpracování

7.1.4 Architektura

Spring Framework obsahuje modul Spring MVC, který poskytuje rozsáhlou podporu MVC návrhu pro tvorbu webových aplikací. Na obrázku 13 je zobrazena architektura Spring REST MVC.



Obrázek 13: Spring MVC architektura
Zdroj: [31]

Každý požadavek prochází filtrem. *Dispatcher Servlet* poté požadavek zanalyzuje a předá ho Controlleru k dalšímu zpracování. Dále se dostává na řadu *Handler Mapping*, který mapuje přicházející požadavky na *Handler* (metoda, která je součástí Controlleru a stará se o samotné zpracování požadavku). V případě REST služby vrací Controller přímo data (např. ve formátu JSON) pomocí anotace *@ResponseBody* [22].

```
@RestController
public class ItemListController {

    private final ItemService itemService;

    @Autowired
    public ItemListController(ItemService itemService) {
        this.itemService = itemService;
    }
    @CrossOrigin
    @PostMapping(value = "/getItems")
    public List<Item> showList(@RequestBody ShoppingList shoppingList) {
        return itemService.findItems(shoppingList);
    }
}
```

Ukázka kódu 6: Spring – Definování controlleru
Zdroj: vlastní zpracování

Anotace *@RestController* v sobě obsahuje jak anotaci *@Controller*, tak anotaci *@ResponseBody*, tu již tedy není potřeba definovat. Pomocí této kombinace anotací rozpozná framework, že se jedná o komponentu starající se o zpracování požadavků od klienta. Anotace *@Autowired* slouží k injektování závislosti na servisní třídu. *@CrossOrigin* zajišťuje *Cross-origin resource sharing* čili sdílení zdrojů pro aplikaci na jiné doméně. *@PostMapping* je zkrácená verze pro *@RequestMapping(method = RequestMethod.POST)* a takto anotovaná metoda zpracovává HTTP POST požadavek na dané URI (*value*). Anotace *@RequestBody* automaticky deserializuje příchozí JSON na Java objekt. V takovém případě musí příchozí JSON korespondovat s typem, který je touto anotací obalen [32].

7.1.5 Zabezpečení

Standardem pro zabezpečení springovských aplikací je modul Spring Security poskytující širokou paletu služeb pro vytvoření zabezpečené webové aplikace. Pro použití modulu je potřeba vytvořit závislost v *pom.xml.*

Autentizace

Nedílnou součástí Spring Security jsou služby poskytující nástroje pro autentizaci a autorizaci uživatele. Nejdříve je potřeba nakonfigurovat potřebné služby v konfigurační třídě anotovanou pomocí *@EnableWebSecurity* a implementující *WebSecurityConfigurerAdapter*.

Hlavním rozhraním pro zajištění autentizace je *AuthenticationManager*, který vybírá *AuthenticationProvider* podle konfigurace. V rámci tohoto provideru je možné definovat *PasswordEncoder* zajišťující hashování hesla.

Základním objektem je *SecurityContextHolder*, kde jsou uloženy detaily současného kontextu aplikace obsahující atributy uživatele, který je právě přihlášen. K reprezentaci těchto informací je používán objekt *Authentication.* V následující ukázce kódu je zobrazen způsob, jak získat objekt reprezentující přihlášeného uživatele.

```
MyUserPrincipal myUserPrincipal = (MyUserPrincipal)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

Ukázka kódu 7: Spring – získání přihlášeného uživatele

Zdroj: vlastní zpracování

Objekt *MyUserPrincipal* je vlastní implementace defaultního objektu *UserDetails*. Jedná se o rozhraní mezi vlastním objektem *User* v databázi a mezi objektem, který potřebuje *SecurityContextHolder*. Pro práci s *UserDetails* je potřeba implementovat *UserDetailsService* pro zajištění komunikace mezi kontextem a uživatelskými detaily [33].

7.1.6 Cachování

Spring Boot automaticky nakonfiguruje cachovací logiku v případě, že je cache povolena pomocí anotace *@EnableCachin*g v inicializační třídě aplikace. Pomocí anotace *@Cacheable* u metody v konkrétní repository je možné určit databázovou tabulku, podle které se budou data ukládat do mezipaměti [34].

```
@Override
@Cacheable("item")
public List<Item> findAllByListId(ShoppingList shoppingList) {
// . . .
}
```

Ukázka kódu 8: Spring – konfigurace cache v metodě
Zdroj: vlastní zpracování

7.1.7 Testování

Spring v základu poskytuje zabudované moduly pro podporu jednotkového a integračního testování. V ukázce kódu 9 je znázorněna konfigurace integračního testu. Pomocí anotace *@RunWith* pozná Spring testovací případ. Anotace *@SpringBootTest* zajistí spuštění aplikačního kontextu tím, že vyhledá hlavní konfigurační třídu (tu s anotací *@SpringBootApplication*)

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class UserServiceTest {

@Autowired
UserService userService;
// . . .
```

Ukázka kódu 9: Spring – konfigurace integračního testu
Zdroj: vlastní zpracování

@Autowired je rozpoznána Springem a daná servisní třída je injektována do kontextu ještě před samotným testem [35].

7.2 Django

Backendová aplikace vytvořená ve frameworku Django (verze 2.0) je psaná v jazyce Python (verze 3.7) a následuje principy MVT architektury se zaměření na Model a View. Aplikace využívá REST API pro komunikaci s frontendem. Jako vývojové prostředí byl vybrán software PyCharm od společnosti JetBrains.

7.2.1 Instalace a základní konfigurace

Pro vytvoření Django aplikace je potřeba mít nainstalovaný jazyk Python a framework Django. V IDE je nutné nastavit virtuální prostředí a Python interpreta. Hlavním skriptem pro práci s Djangem je *django-admin*. Samotná aplikace se vytvoří příkazem *django-admin start-project nazev-aplikace.* Tím se vytvoří základní balíčková struktura se základní konfigurací projektu (*settings.py*) týkající se například zabezpečení (INSTALLED_APPS). Projekt obsahuje vlastní webový server určený pro vývoj.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Ukázka kódu 10: Django – základní konfigurace
Zdroj: vlastní zpracování

Některá z konfigurací využívá databázové tabulky, ty se vytvoří automaticky příkazem *python manage.py migrate.* Vygenerovaný projekt obsahuje i spustitelnou administrátorskou část, kde je možné v pozdějších fázích projektu manipulovat s databázovými objekty bez toho, aniž by programátor musel tyto funkce programovat separátně [5].

7.2.2 Balíčková Struktura

```
shoppingList_django/
    manage.py
    tests
    __init__.py
    settings.py
    urls.py
    wsgi.py
    app/
      migrations
      _init_.py
      models.py
      serializers.py
      urls.py
      views.py
```

Ukázka kódu 11: Django – balíčková struktura
Zdroj: vlastní zpracování

- Vnější balíček `shoppingList_django` funguje jen jako kontejner pro projekt.
- soubor `manage.py` je třída, která nahrazuje skript *django-admin.*
- Vnitřní `app` balíček slouží jako skutečná složka pro samotnou aplikaci. Jméno složky je potřeba při importu jiných souborů do projektu.
- Soubor `__init__.py` je prázdný soubor, který určuje, že nadřazená složka se chová jako Python balíček.
- Soubor `settings.py` slouží ke konfiguraci projektu, například k připojení do databáze nebo k importu knihoven starající se o zabezpečení. Soubor `urls.py` slouží ke směrování na URL adresy.
- Balíček `migrations` se automaticky vytvoří po migraci databázového modelu
- `models.py` obsahuje definici objektů
- `serializers.py` obsahuje metody pro serializaci na Python objekty
- `views.py` obsahuje třídy a metody pro aplikační logiku

7.2.3 Připojení do databáze

V základu používá Django databázi SQLite. Ukázková aplikace ale potřebuje přístup k MySQL serveru, je tedy potřeba nakonfigurovat připojení do databáze v souboru *settings.py*. Dále je potřeba mít nainstalovaný plugin *mysqlclient*

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'data',
        'USER': 'root',
        'PASSWORD': 'root',
        'HOST': 'localhost',
        'PORT': '3306'
}
```

Ukázka kódu 12: Django – připojení do databáze
Zdroj: vlastní zpracování

Dále je potřeba definovat model aplikace v souboru *models.py*. Níže je ukázka vytvoření objektu *ShoppingList* s cizím klíčem vázaným na tabulku *User.*

```
class ShoppingList(models.Model):
    createdAt = models.DateTimeField(default=datetime.now)
    user = models.ForeignKey(User, on_delete=models.CASCADE, null=True,
default=None)
```

Ukázka kódu 13: Django – model
Zdroj: vlastní zpracování

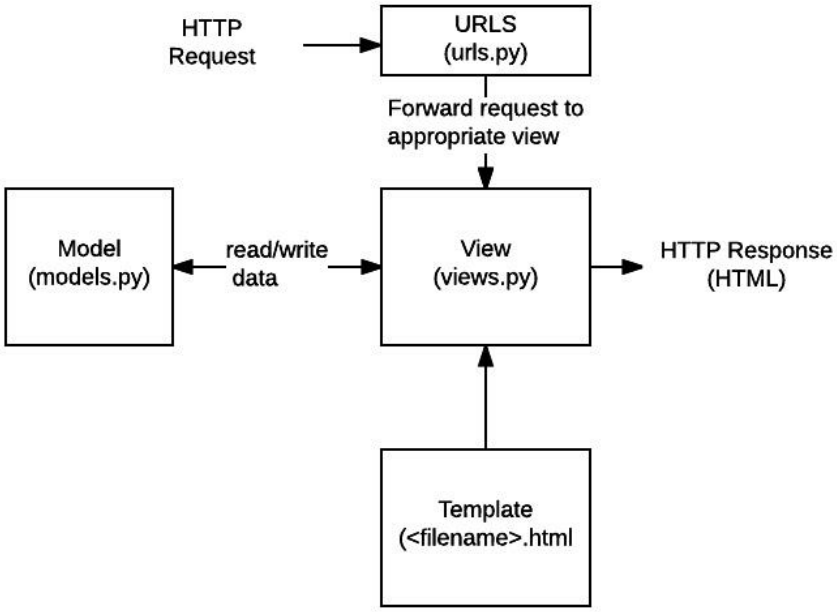
Následuje provedení migrace. Příkaz *python manage.py makemigrations app* říká frameworku, že v modelu nastala nějaká změna (v tomto případě vytvoření nových modelů). Následné změny jsou propsány do databáze a zároveň jsou uloženy v migraci (balíček `migrations`). V základu Django implementuje vrstvu pro ORM, není tedy potřeba další definice ohledně objektově relačního mapování. Django pracuje pouze s metodami a objekty Pythonu, ty ORM automaticky transformuje na dotazy a zároveň se je snaží sám optimalizovat. Pro složitější dotazy je tu i samozřejmě možnost psát dotazy přímo pomocí SQL. Práce s objekty je však velice intuitivní a vyžaduje minimální množství kódu [5]. Například pro výpis objektů filtrované podle určité hodnoty je následující:

```
ShoppingList.objects.filter(user=user)
```

Ukázka kódu 14: Django – filtrování objektu
Zdroj: vlastní zpracování

7.2.4 Architektura

Přestože Django využívá principy MVC, pro určité aspekty implementace používá vlastní logiku. Vzhledem k tomu, že práci Controlleru v obvyklém MVC návrhu zde zastává samotný Django Framework, se Django často označuje jako MVT framework.



Obrázek 14: Django MVT architektura
Zdroj: [36]

V MVT návrhu platí, že Model reprezentuje vrstvu starající se o přístup k datům, stejně jako v MVC architektuře. Odlišná implementace nastává v komponentách View a Template. V rámci ukázkové aplikace není důležitý Template vzhledem k tomu, že v aplikaci je implementován Django Rest Framework, kde není obsah renderován do Template, ale data jsou posílána klientovi přes REST API. View obsahuje logiku, která se stará o přístup k Modelu a zároveň zachycuje požadavek pocházející od klienta a vrací odpověď s potřebnými daty [5].

```
// ursl.py

path('api/new', views.CreateItem.as_view())

// views.py

class CreateItem(APIView):
    def post(self, request):
        serializer = ItemSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors,status=status.HTTP_400_BAD_REQUEST)
```

Ukázka kódu 15: Django – APIView

Zdroj: vlastní zpracování

V ukázce kódu 15 je znázorněna implementace *APIView*. Jedná se přímo o DRF komponentu, ve které je možné definovat zachycení HTTP metody, v tomto případě metody POST. Metoda je zachycena podle definice v souboru *urls.py*. Pomocí *serializeru* se data v příchozím požadavku transformují na Python objekt, se kterým je možné dále pracovat [37].

7.2.5 Zabezpečení

Django i DRF poskytují základní služby pro zabezpečení webové aplikace.

Pro minimální zabezpečení je potřeba nadefinovat následující backend v souboru *settings.py.*

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
]
```

Ukázka kódu 16: Django – konfigurace zabezpečení

Zdroj: vlastní zpracování

Autentizace

Výhodou tohoto frameworku je původní *User model*, který je možné použít v rámci autentizace. Při vytváření uživatele pomocí *User.objects.create_user(username=name, password=password)* je zajištěno uložení nového uživatele do databáze s bezpečně zahashovaným heslem.

DRF poskytuje tři základní typy autentizace: *BasicAuthentication*, *TokenAuthentication* a *SessionAuthentication*. Pro ukázku byla v aplikaci implementována *SessionAutehntication*, cože je defaultní typ autentizace, pokud

není v konfiguraci nastaveno jinak. V případě úspěšné autentizace je možné získat přihlášeného uživatele přes *request.user* [38]. Způsob konfigurace použité autentizace je znázorněn v ukázce kódu 17.

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.SessionAuthentication',
    )
}
```

Ukázka kódu 17: Django – konfigurace autentizace
Zdroj: vlastní zpracování

7.2.6 Cachování

Django poskytuje celkem robustní cachovací systém. Typickým využitím je uložení výstupu View. Nastavení cache v rámci Djanga není složité. Nejprve je potřeba nastavit, jakým způsobem chceme data ukládat – přímo do databáze, filesystemu nebo přímo do paměti. V rámci ukázkové aplikace bylo využito ukládání přímo do paměti pomocí *Loc-memory caching.*

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
    }
}
```

Ukázka kódu 18: Django – konfigurace Loc-memory caching
Zdroj: vlastní zpracování

Jedná se o globální nastavení, při každém požadavku tedy proběhne ukládání dat do mezipaměti [39].

7.2.7 Testování

Django k testování využívá malou kolekci tříd postavenou na Python knihovně *unittest*. Knihovna podporuje jak jednotkové, tak integrační testování. Každá testovací třída musí dědit z třídy *TestCase*, která testy spouští v rámci jednotlivých transakcí, čímž zajistí jejich oddělení.

```
class CreateItemTestCase(TestCase):

    def setUp(self):
        self.item = Item.objects.create(name='TestName',
createdAt=datetime.now(), shoppingList_id="", state=False )

    def testGetItems(self):
        self.assertEquals(Item.objects.get(name='TestName').name, 'TestName')
```

Ukázka kódu 19: Django – ukázka testu

Zdroj: vlastní zpracování

Na ukázce kódu 20 je testován případ vytvoření nové položky. Metoda *setUp* je volána vždy jako první. Django automaticky vytvoří testovací databázi v případě, že je v rámci testu potřeba vstup do databáze [40].

7.3 Node.js

Backendová aplikace postavena na platformě Node.js (verze 10.13.0) je naprogramována ve frameworku Express.js (verze 4.16). Framework implementuje architekturu MVC. Přes REST API jsou data poslána na frontend. Jako vývojové prostředí byl použit software WebStorm od společnosti JetBrains.

7.3.1 Instalace a základní konfigurace

Součástí instalace Node.js je i balíčkový ekosystém *npm*. Díky tomuto balíčku lze instalovat a spravovat dodatečné knihovny potřebné k vývoji aplikace. Samotný projekt lze vytvořit buď v IDE (framework Express.js je součástí standardní instalace WebStorm) nebo jej lze vygenerovat pomocí nástroje *express-generator*. Nejdůležitějšími vygenerovanými prvky jsou soubory *www* a *app.js*. Tyto soubory obsahují základní konfiguraci aplikace a její samotné spuštění. Node.js má vestavěný *http modul*, který se kompletně stará o HTTP komunikaci. Tento modul tudíž slouží jako webový server [41].

```
var app = require('../app');
var http = require('http');

var port = normalizePort(process.env.PORT || '8080');
app.set('port', port);

var server = http.createServer(app);
server.listen(port);
```

Ukázka kódu 20: Node.js – základní konfigurace
Zdroj: vlastní zpracování

7.3.2 Balíčková Struktura

```
ShoppingList/
    bin/
      www
    model/
    node_modules/
    routes/
    app.js
    package.json
```

Ukázka kódu 21: Node.js – balíčková struktura
Zdroj: vlastní zpracování

- Adresář `bin/` obsahuje javascriptový soubor `www`, který definuje základní nastavení aplikace
- V adresáři `model/` jsou soubory definující připojení do databáze a tvorbu databázových entit
- Adresář `npm_modules` obsahuje několik tisíc podadresářů a souborů obsahující vygenerované knihovny
- Adresář `routes/`, také označován jako `controllers/`, obsahuje definici směrování požadavků
- Soubor `app.js` je hlavní spouštěcí soubor aplikace
- Soubor `package.json` obsahuje seznam závislostí

7.3.3 Připojení do databáze

Aplikace ve frameworku Express.js mohou využít jakoukoliv databázi, kterou podporuje Node.js. Vzhledem k tomu, že jako databáze pro aplikaci byla vybrána MySQL, byl použit ORM Framework *Sequelize* pro práci s perzistentními daty. Jelikož se jedná o framework psaný v JavaScriptu, je *Sequelize* také založen na práci s *promisy*. Pro připojení k databázi je potřeba nainstalovat dvě knihovny pomocí příkazů *npm install --save sequelize* a *npm install --save mysql2.*

```
const Sequelize = require('sequelize')
const dbConnection = new Sequelize('demo', 'root', 'root', {
    host: 'localhost',
    dialect: 'mysql'
})
```

Ukázka kódu 22: Sequelize – připojení do databáze
Zdroj: vlastní zpracování

V ukázce kódu 22 je znázorněno připojení do databáze pomocí ORM. Definování entity je následně velice jednoduché (ukázka kódu 23) – pomocí příkazu *define* zavolaným nad vytvořeném připojení *dbConnection*. Přes metodu *sync()* je provedena synchronizace s databázovým schématem. Tedy pokud tabulka v databázi neexistuje, vytvoří se nová tabulka dle definice [42].

```
const ShoppingList = dbConnection.define('shopping_list', {
        id: {
            type: Sequelize.INTEGER,
            primaryKey: true,
            autoIncrement: true
        },
        created_at: new Date(),
    },
    {tableName: 'shopping_list'});

dbConnection.sync();
```

Ukázka kódu 23: Sequelize – vytvoření entity
Zdroj: vlastní zpracování

Následné provedení jednotlivých operací nad objekty může být provedeno opět pomocí *Sequelize*, jak je znázorněno v ukázce kódu 24.

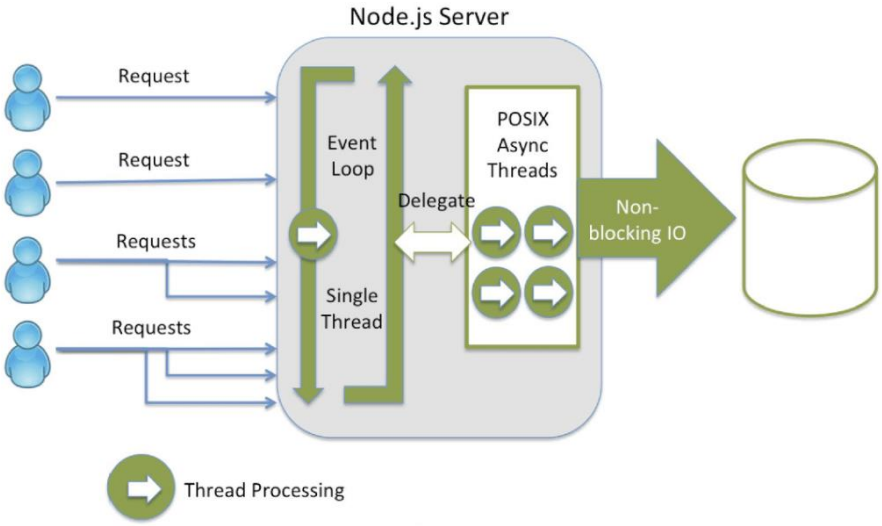
```
Item.findAll({where: {
    shopping_list_id: req.body.id
    }}).then(items => {
    res.send(items)
});
```

Ukázka kódu 24: Sequelize – získání položek podle id
Zdroj: vlastní zpracování

7.3.4 Architektura

Typickou vlastností architektury webových frameworků je zpracovávání požadavků pomocí více vláken. Node.js ovšem používá *Single Threaded Event Loop* architekturu. Tato architektura je založena na modelu řízení pomocí událostí s mechanismem *callbacku*, což je pro JavaScript typické. Výhodou této architektury je schopnost zpracovávat několik klientských požadavků zároveň v jednom vlákně.

Každá akce provedená v rámci aplikace je chápána jako událost. Základem celého procesu je smyčka událostí (*event loop*). Smyčka přijímá všechny uživatelské požadavky a přiřadí je jednomu vláknu (*single thread*). Další prováděné operace jsou řešeny pomocí neblokujícího I/O modelu. To znamená, že například HTTP požadavky musí být asynchronně zpracovávány, vzhledem k tomu, že se neustále pracuje pouze v jednom vlákně. Každý nově příchozí požadavek je posílán opět smyčce událostí. Webový server tedy není blokován pokaždé, když přijde nový požadavek [43].



Obrázek 15: Node.js architektura
Zdroj: [43]

Na ukázce kódu 25 je znázorněno zpracování požadavku POST. Pomocí *express()* se načte modul frameworku. Nad ním se zavolá metoda *post*, která přijímá anonymní funkci jako parametr. Funkce obsahuje minimálně další dva parametry – *req* jako požadavek a *res* jako odpověď. Parametr *req* mimo jiné obsahuje atribut *body* definující příchozí objekt, který je následně zpracován a uložen do databáze jako *Item*.

```
var app = express();
app.post("/api/newItem", (req, res, next) => {
    let newItem = {
        count: req.body.count,
        created_at: new Date().toLocaleString(),
        name: req.body.name,
        state: req.body.state
    }

// volani Sequelize

});
```

Ukázka kódu 25: Node.js – Zpracování requestu
Zdroj: vlastní zpracování

7.3.5 Zabezpečení

Vzhledem k nutnosti importovat spoustu balíčků je aplikace v Node.js náchylnější k bezpečnostním hrozbám, proto je potřeba věnovat zvýšenou pozornost při výběru a instalaci knihoven.

Autentizace

Node.js sám o sobě neimplementuje žádnou autentizační strategii, je tedy potřeba využít jednu z knihoven. V rámci projektu byla použita knihovna *Passport.js*. Ta poskytuje mnoho strategiií pro autentizaci uživatele. V projektu byla použita strategie *passport-local*, která ověřuje uživatele klasickým způsobem pomocí uživatelského jména a hesla. *Passport.js* automaticky nehashuje hesla při registraci uživatele, z toho důvodu bude využita externí knihovna *bcrypt-nodejs.*

Při použití *passport-local* je potřeba definovat autentizační strategii, autentizační middleware a případně *session*, pokud je potřeba pamatovat si přihlášeného uživatele i při následných požadavcích po úspěšné autentizaci. Referenci na *passport-local* lze získat pomoc *require*. Při samotném vytváření strategie je důležité implementovat *verifikační callbacky* zajišťující vyhledání uživatele, který odpovídá příchozím přihlašovacím údajům.

```
const LocalStrategy = require('passport-local').Strategy;
passport.use('local', new LocalStrategy(
    {
        usernameField : 'name',
        passwordField : 'password'
    },
    function(username, password, done) {
        User.findOne( {
                name: username
            }).then(user => {
                return done(null, user);
        });
    }
));
```

Ukázka kódu 26: Passport.js – autentizační strategie
Zdroj: vlastní zpracování

Při správné konfiguraci se při každé autentizaci požadavku použije strategie definovaná na ukázce kódu 26. Nejprve je parsováno příchozí uživatelské jméno a heslo a údaje jsou následně poslány jako argumenty do verifikačního callbacku. Pokud jsou údaje validní (v tomto případě uživatelské jméno), je zavolán parametr *done*, který dále distribuuje přihlášeného uživatele.

```
passport.serializeUser(function(user, done) {
    done(null, user.id);
});

passport.deserializeUser(function(id, done) {
    User.findById(id).then(function(user) {
        done(null, user);
    }).catch(function(err) {
        if (err) {
            throw err;
        }
    });
});
```

Ukázka kódu 27: Passport.js – serializace uživatele
Zdroj: vlastní zpracování

Passport.js podporuje držení přihlášeného uživatele v session pomocí metod *serializeUser* a *deserializeUser.* Na ukázce kódu 27 je znázorněna implementace těchto metod. Uživatel je serializován do session pouze pomocí jeho unikátního ID, tím se minimalizuje počet dat, které se v session uchovávají. Při autentizaci požadavku je toto ID použito k vyhledaní uživatele, kterého je poté možno získat přes *req.user*. Odhlášení uživatele lze poté jednoduše provést pomocí *req.logout()* [44].

7.3.6 Cachování

Jako základní cachovací techniku lze použít knihovnu *memory-cache*. Jedná se o jednoduchou knihovnu pro Node.js, instalace se provádí pomocí příkazu *npm install memory-cache.*

Parametr *duration* reprezentuje údaj, jak dlouho budou hodnoty uchovány v mezipaměti. V definici *cacheMiddleware* se vytvoří unikátní klíč obsahující url requestu. Dále je provedena kontrola, jestli už je hodnota podle daného klíče uložena v mezipaměti. Pokud ano, je hodnota vrácena a request je dále zpracován. Pokud hodnota v mezipaměti není, je do ní vložena pomocí metody *put()* [45].

```
let cacheMiddleware = (duration) => {
    return (req, res, next) => {
        let key = 'myKey' + req.originalUrl || req.url
        let cacheContent = memCache.get(key);
        if(cacheContent){
            // . . .
        }else{
            // . . .
        }
    }
}
// . . .
app.post("/api/getItems",  cacheMiddleware(30),(req, res, next) => {
// . . .
});
```

Ukázka kódu 28: Node.js – memory-cache
Zdroj: vlastní zpracování

7.3.7 Testování

Pro testovací účely je v projektu vytvořen adresář *test*, se souborem *test.js* a vlastním souborem *package.json*. Typickou knihovnou pro testování Node.js je knihovna *Mocha.js*. Základem každého testu jsou klíčová slova *describe* a *it*.

```
describe('Testing Shopping Lists', function() {
    describe('GET Lists', function() {
        it('should return shopping lists', function(done){
            // do the testing
        });
    });
});
```

Ukázka kódu 29: Mocha – ukázka testování v Node.js
Zdroj: vlastní zpracování

Metoda *describe()* slouží pro seskupování souvisejících testů. Jako první argument se definuje název celé skupiny a druhý argument je *callback*. Metoda *it()* je používána v rámci jednotlivých testovacích případů a jasně se v ní definuje účel daného testu [46]. V ukázce kódu 29 je znázorněna definice testovací metody pro získání všech nákupních lístků.

7.4 Ruby on Rails

Backendová aplikace postavena na frameworku Ruby on Rails (verze 5.2.0) je naprogramována v jazyce Ruby (verze 2.4.4). Framework implementuje architekturu MVC. Přes REST API jsou data posílána na frontend. Jako vývojové prostředí byl použit software RubyMine od společnosti JetBrains.

7.4.1 Instalace

Nejdříve je potřeba stáhnout a nainstalovat jazyk Ruby společně s vývojářskými nástroji. Ruby má nástroj *RubyGems* pro správu a distribuci programů a knihoven (*gems*). Například framework Rails se tedy nainstaluje pomocí příkazu *gem install rails*. Rails poskytuje různé skripty (generátory) pro vytvoření základního kódu a konfigurace při běžných úkonech. Jedním z generátorů je příkaz *new*. Pomocí *rails new shopping_list* je vytvořena základní kostra a konfigurace aplikace. Jako defaultní webový server je používán server Puma [26].

7.4.2 Balíčková Struktura

```
ShoppingList/
    app/
      assests/
      channels/
      controllers/
      helpers/
      jobs/
      mailers/
      models/
    bin/
    config/
    db/
    test/
    tmp/
    vendor/
```

Ukázka kódu 30: Rails – balíčková struktura

Zdroj: vlastní zpracování

- Adresář `app/` sdružuje hlavní aplikační komponenty v podadresářích
- Adresář `app/controllers` obsahuje controllery, kde jsou obsluhovány požadavky
- Adresář `app/models` obsahuje třídy definující data
- V adresáři `config/` je obsažena veškerá konfigurace potřebná k běhu aplikace

- Do adresáře `db/` se ukládají skripty pro práci s relační databází

7.4.3 Připojení do databáze

Pro připojení do databáze je třeba mít přidat gem *mysql2* do souboru *Gemfile.* Poté je potřeba nastavit konfiguraci v souboru *database.yml* (ukázka kódu 31).

```
development:
    adapter: mysql2
    encoding: utf8
    database: demo
    pool: 5
    username: root
    password: root
    socket: /tmp/mysql.sock
```

Ukázka kódu 31: Rails – připojení do databáze
Zdroj: vlastní zpracování

V Rails existuje *Active Record*. Ten se stará o vytváření a použití objektů, jejichž data vyžadují perzistentní ukládání. To znamená, že *Active Record* zde funguje i jako samotný ORM framework. Nejjednodušší způsob, jak vytvořit samotný objekt je přes terminál příkazem `$ rails generate model ShoppingList`. Po tomto příkazu se vytvoří soubor *create_shopping_list.rb* v adresáři `db/migrate/` a samotný objekt v adresáři `app/models`. V modelu se případně můžou dodefinovat cizí klíče.

```
class CreateShoppingLists < ActiveRecord::Migration[5.2]
  def change
    create_table :shopping_lists do |t|
      t.timestamps
    end
  end
end
```

Ukázka kódu 32: Rails – vytváření tabulky
Zdroj: vlastní zpracování

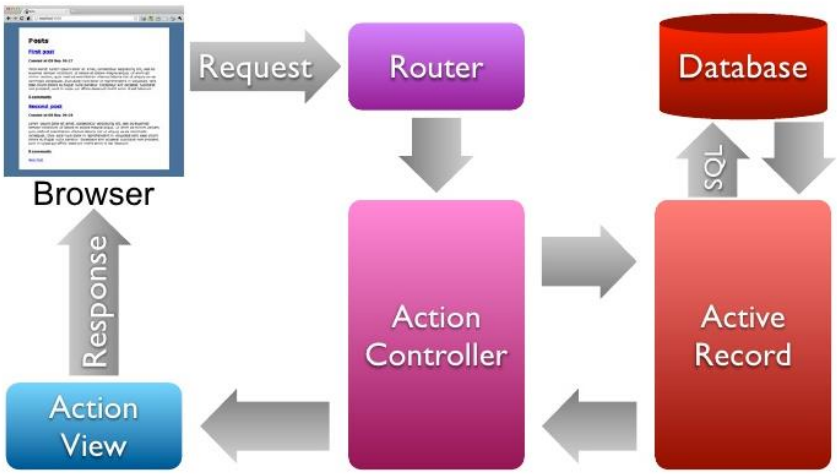
Není potřeba definovat primární klíč, framework si jej v databázi definuje sám jako sloupec *id*. Dalším předdefinovaným atributem je *t.timestamps*, díky němuž se v databázové tabulce automaticky vytvoří sloupce *created_at* a *updated_at*. Aby byla tabulka vytvořena, je potřeba zavolat příkaz v terminálu *rails db:migrate*. Rails je silně založen na principu Convention over configuration, což se v rámci migrace projevuje nejen předdefinovanými atributy, ale i v názvu tabulky [47]. Nově vytvořený objekt se tedy vytvoří jako tabulka *shopping_lists*. Další výhodou tohoto přístupu je automatické mapování objektových tříd na databázové tabulky, není tedy třeba dalšího kódu v rámci ORM.

```ruby
def getItems
  list_id = params[:id]
  items = Item.where(shopping_list_id: list_id)
  // . . .
end
```

Ukázka kódu 33: Rails – získání položek podle id
Zdroj: vlastní zpracování

7.4.4 Architektura

Implementace MVC má svá určitá specifika. Model je zde implementován pomocí objektu *Active Record*, o němž již byla zmínka v předchozí kapitole. Controller je zde implementován jako *Action Controller*. Tak, jako v každé obvyklé REST aplikaci, slouží Controller k přijímání požadavků a k případné úpravě modelu. *Action Controller* je společně s *Action Dispatch* součástí *Action Packu*. *Action Dispatch* parsuje informace o požadavku, obsahuje router starající se o směrování požadavků a stará se o další logiku týkající se například *cookies* nebo *sessions*.



Obrázek 16: Rails architektura
Zdroj: [48]

Každý nově vytvořený Controller dědí z třídy *ActionController*. V ukázce kódu 34 je znázorněn Controller a definice metody. Ve frameworku Rails je velmi důležitá jmenná konvence, a to z důvodu směrování požadavků na Controllery a jejich metody [49].

```
class ItemsController < ApplicationController
  def newItem
    itemNew = Item.create(name: params[:name], count: params[:count], state:
params[:state])
    respond_to do |format|
      format.any { render json: itemNew, content_type: 'application/json' }
    end
  end
end
```

Ukázka kódu 34: Rails – definice controlleru
Zdroj: vlastní zpracování

V souboru *routes.db* je definována cesta jako *post '/api/newItem' => 'items#newItem'*. Framework tedy ví, že na dané cestě má hledat metodu *newItem* v Controlleru *ItemsController*. Konkrétní požadavek není pro vývojáře viditelný, ale může k jeho parametrům přistoupit přes *params[]*. Pomocí metody *respond_to* je klientovi vrácena odpověď s daty v určitém formátu. Součástí odpovědi je automaticky i HTTP status.

7.4.5 Zabezpečení

Co se týče autentizace uživatele, nabízí Rails několik možností. Například gem *Devise* nabízí mnoho nástrojů a metod pro autentizaci uživatele. Pro potřeby ukázkové aplikace je ale zbytečné využít takto komplexní řešení. Byla tedy implementována vlastní autentizační strategie na podobných principech.

Autentizace

Nejprve je potřeba vytvořit model *User* pomocí *rails g scaffold User email:uniq password:digest.* Tento příkaz vytvoří kostru jak modelu v databázi, tak pro Controller. Příkaz *digest* zajišťuje validaci hesla. V modelu se vytvoří metoda *has_secure_password* zajišťující hashování hesla pomocí funkce *bcrypt* (potřebný gem musí být definován v *Gemfile*) a dále poskytuje metodu *authenticate* pro autentizaci uživatele na základě jména a hesla. Po každém přihlášení se autentizovaný uživatel uloží do *session* a následně je možné k němu přistoupit v Controlleru pomocí *current_user* [50].

```
user = User.find_by_name(params[:name])
if user && user.authenticate(params[:password])
  session[:user_id] = user.id
```

Ukázka kódu 35: Rails – autentizace uživatele
Zdroj: vlastní zpracování

7.4.6 Cachování

Rails v základu poskytuje tři typy strategií pro cachování bez toho, aniž by musel být instalován dodatečný plugin. Jedná se o *Page Caching*, *Action Caching* a *Fragment Caching.* V ukázce kódu 36 je znázorněna implementace strategie *Action Caching*. V tomto případě je cachována celá odpověď, ale na rozdíl od *Page Caching*, prochází požadavek přes *Action Pack*. Výhodou tohoto přístupu je to, že filtrování probíhá ještě před samotným cachováním, což umožňuje použití cache i při autentizaci či jiných omezeních [51].

```
class ItemsController < ApplicationController
  before_action :authenticate, except: :index

  caches_page   :index
  caches_action :getItems
```

Ukázka kódu 36: Rails – Action Caching
Zdroj: vlastní zpracování

Na ukázce kódu 36 je znázorněno povolení cachováni v ItemsController. Metoda index nepotřebuje autentizaci, tudíž je zde vhodnější použít rychlejší *Page Caching*. Na všechny ostatní metody Controlleru se již vztahuje autentizace, pomocí *caches_action: getItems* se tedy definuje konkrétní metoda, na kterou bude cachování aplikováno.

7.4.7 Testování

Testovací nástroje jsou součástí základního balíčku Rails. Při vytváření aplikace se vytvoří i složka `test/`, která má podobnou strukturu jako hlavní aplikace. Pro účely testování využívá Rails zvlášť testovací prostředí. To je konfigurovatelné v souboru *test.rb*. Při vytváření modelu se automaticky vytvoří i kostra testovacího kódu, takzvaný *Minitest*.

```
require 'test_helper'
class ItemTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

Ukázka kódu 37: Rails – Minitest
Zdroj: vlastní zpracování

Pomocí *test_helper.rb* je načtena konfigurace potřebná pro běh testu. Testovací třída dědí z *ActiveSupport::TestCase*, čímž je k dispozici velké množství testovacích metod. Metoda *test* definuje jméno testu. Rails před jméno automaticky přidá prefix *test_*, čímž se jasně definuje její účel.

```
test "should save item " do
  item = Item.new(name: "myItem", state: false, count: 1)
  assert item.save
end
```

Ukázka kódu 38: Rails – implementace testu
Zdroj: vlastní zpracování

Rails používá testovací databázi pro testy vyžadující interakci s databází. Do testovací databáze se při každém spuštění testu nahrají testovací data definované v adresáři `test/fixtures/`. Tyto data jsou opět automaticky vytvořena při generování modelu [52]. V ukázce kódu 38 je znázorněna implementace jednoduchého testu na vytvoření nové položky.

7.5 Vaadin

Aplikace vytvořena pomocí frameworku Vaadin (verze 13) je naprogramovaná odlišným způsobem, než tomu bylo u předchozích frameworků. Vzhledem k podstatě frameworku není aplikace striktně rozdělena na frontend a backend, respektive celkový vývoj aplikace probíhá na straně serveru. Vzhledem k tomu, že v tomto přístupu není vhodná komunikace přes REST API, komunikace a předávání dat probíhá pomocí Java UI Component API.

7.5.1 Instalace

Vaadin poskytuje několik startovacích projektů na stránce https://vaadin.com/start/latest/project-base. Pro účely této práce byla vybrána demo aplikace s propojením na Spring. Jako defaultní aplikační server byl použit Jetty Server [6].

7.5.2 Balíčková Struktura

```
ShoppingList/
    src/
     main/
        java/
           configuration/
           presenter/
           model/
           repository/
           service/
           view/
           Application.java
        resources/
           application.properties
        test/
        pom.xml
```

Ukázka kódu 39: Vaadin – balíčková struktura
Zdroj: vlastní zpracování

- balíček `java/` obsahuje všechny soubory zdrojového kódu rozděleného do dalších balíčků podle funkcionality
- balíček `configuration/` obsahuje konfigurační třídy
- balíčky `model/`, `repository/` a `service/` obsahují třídy pro definování entit a komunikaci s databází
- balíček `view/` obsahuje třídy pro definici GUI a zachycování událostí

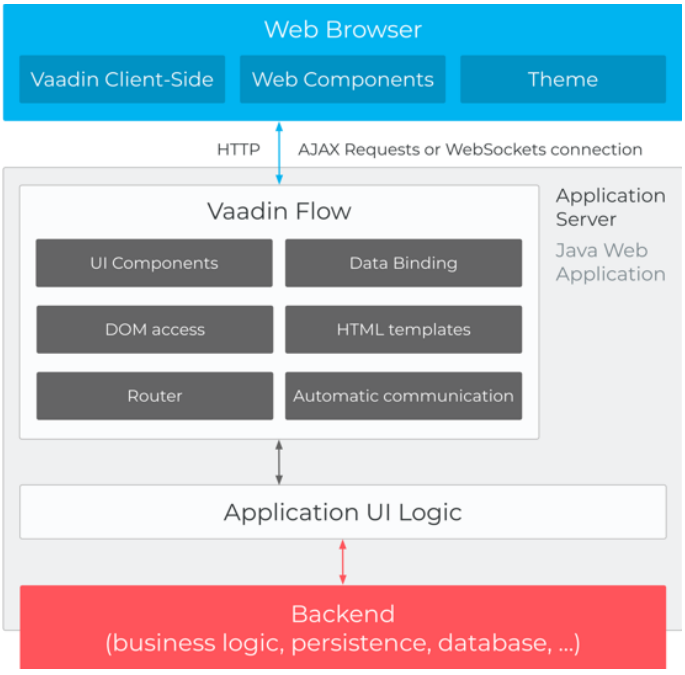
- balíček `presenter/` obsahuje třídy pro obsluhu událostí
- balíček `resource/` obsahuje soubor *application.properties*, který obsahuje informace potřebné k připojení do databáze
- balíček test obsahuje testovací soubory
- soubor *pom.xml* je konfigurační soubor Mavenu obsahující informace o závislostech na jednotlivých modulech projektu. Závislosti jsou reprezentovány ve formátu XML.

7.5.3 Připojení do databáze

Připojení do databáze framework Vaadin neřeší. K tomuto účelu je tedy potřeba využít jiný framework, v tomto případě Spring. Problém připojení do databáze pomocí Springu byl již popsán v kapitole 7.1.3.

7.5.4 Architektura

Nejsilnější stránkou Vaadinu je jeho server-side API Vaadin Flow. Ten je součástí standardní Vaadin Platform. Pomocí Vaadin Flow probíhá komunikace mezi backendem (typický psaným v Javě) a frontendem tak, že UI komponenty jsou renderovány na serveru a interakce uživatele s webovými komponenty jsou zachyceny taktéž na serveru pomocí listenerů.



Obrázek 17: Vaadin Flow – architektura
Zdroj: [53]

Oproti přístupu RESTové architektury má backendový vývojář plný přístup k webovým komponentám přes Javu na straně serveru. Programování webové aplikace takovým způsobem spíš připomíná vývoj desktopové aplikace.

```
//ItemsView
@Route("createItems")
public class ItemsView extends VerticalLayout {
// . . .

btnSaveItemsList.addClickListener(e ->{
   presenter.saveShoppingList(user, editor.currentItems);
});


//ItemsViewPresenter
public void saveShoppingList(User user, List<Item> items) {
    shoppingListService.createList(user, items);
}
```

Ukázka kódu 40: Vaadin – obsluha událostí
Zdroj: vlastní zpracování

O veškeré detaily komunikace mezi webovým prohlížečem a aplikačním serverem se stará samotný framework a vývojáři zůstávají tyto detaily skryty. Ve Vaadinu je implementována architektura MVP. V tomto případě je Model řešen pomocí Spring frameworku. View je třída, která implementuje Layout a tvoří UI komponenty. Presenter je třída obsluhující uživatelské události a plní roli prostředníka mezi Modelem a View [6].

7.5.5 Zabezpečení

Pro zabezpečení webové aplikace je možné využít framework Spring a jeho modul Spring Security. Tato problematika již byla popsána v kapitole 7.1.5.

7.5.6 Cachování

Pro cachování webové aplikace je také možné využít možnosti frameworku Spring popsané v kapitole 7.1.6.

7.5.7 Testování

Pro testování aplikace psané ve Vaadinu je možné využít možnosti frameworku Spring popsané v kapitole 7.1.7.

8 Srovnání frameworků

V následující kapitole budou na základě předchozí rešerše a implementace popsány v rámci každé technologie jejich možnosti týkajících se dokumentace a podpory, rychlosti učení, testování, zabezpečení a rozšiřitelnosti.

8.1 Dokumentace a podpora

Co se týče dokumentace, má toho Spring Framework spoustu co nabídnout. Na oficiálním github účtu [54] lze nalézt mnoho ukázkových projektů řešících typické příklady a problémy, či integraci s jinými technologiemi. Velkou výhodou frameworku Django je popularita jazyka Python. Django nabízí dokumentaci Django Project popisující celkový vývoj aplikace od úplných základů. Dokumentace je dostupná i v českém jazyce. Co se týče Node.js (respektive Express.js) obsahuje oficiální dokumentace pouze vybrané aspekty vývoje aplikace. Rails, stejně jako Django, poskytuje kompletní tutoriál vývoje aplikace, navíc dostupného i v českém jazyce. Vaadin značně zaostává za předchozími frameworky co se týče dokumentační podpory. Vývojáři Vaadinu však dávají k dispozici ukázkové projekty různých úrovní. Pro přístup k těm komplexnějším ke ovšem potřeba zakoupit licenci.

Ukazatelem podpory může být i aktivita komunity na webu. V tabulce 2 je shrnutí počtu vrácených výsledků po zadání dotazu na dokumentaci ve vyhledávači Google. Druhý sloupec ukazuje počet dotazů na platformě StackOverflow. Informace jsou aktuální k dubnu 2019.

Technologie	Počet výsledků Google	Počet dotazů na StackOverflow
Spring	37 milionů	150 tisíc
Django	17 milionů	194 tisíc
Node.js (Express.js)	14 milionů	51 tisíc
Rails	3 miliony	300 tisíc
Vaadin	0,3 milionů	12 tisíc

Tabulka 2: Podpora technologií na webu
Zdroj: www.google.com, www.stackoverflow.com

Podpora IDE a chytrých editorů

Společnost JetBrains poskytuje vývojová prostředí pro všechny zmíněné jazyky a frameworky. Nevýhodou těchto nástrojů je plná podpora těchto frameworků pouze v placených verzích a velká zátěž na paměť a procesor počítače. Volně dostupnou alternativou pro Spring je nástroj Spring Tools 4, díky kterému je možné využít framework v jiných vývojových prostředích a chytrých editorech. Pro Django existuje nástroj PyDev s integrovanou podporou frameworku. Jasnou volně dostupnou alternativou pro Node.js je Visual Studio Code, kde je JavaScript a případně Node.js nativně podporován. Rails je možné použít v jiných nástrojích a editorech pouze pomocí pluginů.

Technologie	JetBrains	VS Code	Ostatní
Spring	IntelliJ IDEA	Plugin	Eclipse
Django	PyCharm	Plugin	Atom, Vim
Node.js (Express.js)	WebStorm	Nativní podpora	Atom, Vim
Rails	RubyMine	Plugin	Atom, Vim
Vaadin	IntelliJ IDEA	Bez oficiální podpory	Eclipse

Tabulka 3: Podpora IDE a chytrých editorů
Zdroj: [56] [57] [58] [59] [60]

8.2 Rychlost učení

Rychlost učení jakéhokoliv jazyka a frameworku záleží na mnoha faktorech a je pro každého programátora jiná. Z tohoto důvodu je tato kapitola zaměřena spíše na rychlost vývoje v rámci jednotlivých frameworků. Rychlost vývoje je založena na složitosti a principech, na kterých je framework postaven.

Složitost projektu

Díky nástroji Spring Boot je vývoj aplikací ve Spring frameworku značně zjednodušen, stále ale přetrvává problém někdy až příliš robustních řešení na poměrně jednoduchý problém. Django preferuje vývoj pomocí principu DRY a nejen díky tomu jsou jeho projekty relativně přehledné, co se množství kódu týče. Navíc Django poskytuje nástroje jako DRF nebo *Django Admin*, což může značně usnadňovat vývoj. Na druhou stranu musí být každá funkcionalita explicitně konfigurována, což může proces vývoje zpomalovat. Node.js umožňuje sdílet codebase na frontendu a backendu, což je velká výhoda pro full-stack vývojáře. Na druhou stranu představuje poměrně složitý koncept programování, takže není nejvhodnější pro začátečníky. Rails se silně drží principů DRY a CoC, což umožňuje vývojářům tvořit webové aplikace s minimálním množstvím kódu a konfigurace. Framework totiž sám předpokládá nejlepší řešení daného problému. Vaadin oficiálně poskytuje startovací demo projekty různých složitostí, za ty komplexnější se ovšem musí platit, stejně jako za oficiální podporu.

Technologie	Principy	Rychlost vývoje
Spring	Dependency Injection	nízká
Django	Don’t Repeat Yourself	vysoká
Node.js (Express.js)	I/O model	střední
Rails	Convention over Configuration	vysoká
Vaadin	Dependency Injection	střední

Tabulka 4: Rychlost učení a principy vývoje
Zdroj: vlastní zpracování

8.3 Testování

Spring poskytuje několik modulů pro podporu integračního a jednotkového testování. Samozřejmostí je podpora assertačních metod a mockování objektů. Veškerá konfigurace testů probíhá pomocí anotací. Konfigurace může někdy být poněkud nepřehledná, což je dáno tím, že Spring nabízí opravdu velké množství způsobů testování a někdy je těžké určit, jaký způsob je nejvhodnější. Nejjednodušším způsobem testování ve frameworku Django je pomocí balíčku unittest, který je základní součástí standardní knihovny Pythonu. Unittest poskytuje funkce jak pro jednotkové, tak pro integrační testování s podporou assertačních metod a mockování objektů. To vše vyžaduje minimální konfiguraci. Node.js nevyužívá k testování žádný vestavěný balíček, je tedy potřeba stáhnout externí knihovnu (například Mocha.js). Ta obsahuje assertační metody a podporu asynchronního a synchronního testování. Rails má vestavěný testovací framework s minimální potřebou konfigurace. Kostra testovací třídy se navíc automaticky vygeneruje spolu s vytvářením modelu. Vaadin kromě řešení pomocí Springu poskytuje k testování knihovnu TestBench sloužící k celkovému testování aplikace od UI komponent až po databázové dotazy. Vzhledem k tomu, že veškerý vývoj aplikace ve Vaadinu probíhá na straně serveru, je TestBench vítanou a užitečnou knihovnou. Bohužel je tento nástroj dostupný pouze pod placenou licencí. V tabulce 4 je shrnutí nástrojů pro testování s informací, jestli podporuje integrační a jednotkové testování. Ani s jedním z frameworků nebyl v tomto ohledu výraznější problém.

Technologie	Nástroj	Integrační testy	Jednotkové testy
Spring	JUnit	ANO	ANO
Django	Unittest	ANO	ANO
Node.js (Express.js)	Mocha.js	ANO	ANO
Rails	Vestavěný	ANO	ANO
Vaadin	TestBench, JUnit	ANO	ANO

Tabulka 5: Možnosti testování
Zdroj: vlastní zpracování

8.4 Zabezpečení

Zabezpečení je jedním z nejdůležitějších aspektů vývoje serverové části webové aplikace. Nejedná se pouze o autentizaci a autorizaci, která již byla zmíněna v rámci implementace, ale důležitá je i ochrana proti různým útokům a ochrana sdílených dat. Dle organizace OWASP [55] jsou právě tyto tři aspekty největší zranitelností dnešních webových aplikací.

CORS

Jedná se o mechanismus zajišťující bezpečné sdílení zdrojů mezi různými doménami pomocí přidaných HTTP hlaviček. U Springu je konfigurace CORS jednoduchá pomocí anotací. U Djanga je potřeba definovat v konfiguračním souboru CORS middleware. Stejně tak je potřeba definovat CORS u Node.js a Rails, k čemuž slouží opět externí knihovny. U Vaadinu není potřeba CORS konfigurovat, protože aplikace běží na jedné doméně. V tabulce 5 je obsažena informace, jakým způsobem je možné povolit CORS v rámci jednotlivých frameworků.

Autentizace

Důležitým aspektem bezpečnosti aplikace je dobře zvládnutá autentizace. Spring poskytuje díky Spring Security komplexní řešení pro autentizaci a autorizaci, které je sice velice spolehlivé, ale oproti ostatním frameworkům vyžaduje složitější konfiguraci. Django poskytuje v základu tři typy autentizací, vyžadující menší konfiguraci. Node.js samo o sobě neposkytuje bezpečnostní prvky, je tedy potřeba stáhnout externí knihovnu pro potřeby autentizace. Rails pomocí *ActionControlleru* poskytuje základní autentizaci, která ale pro většinu projektů není dostačující. Je tedy potřeba stáhnout externí knihovnu nebo vytvořit vlastní řešení. Vaadin využívá pro autentizaci Spring Security. Podrobněji byly konkrétní implementace autentizace popsány v kapitole 7.

Ochrana proti útokům

Jedním z nejčastějších napadení aplikace je útok typu SQL injection. Jedná se o útok skrz SQL dotaz. V případě úspěchu může mít za následek například ztrátu citlivých dat nebo může jiným způsobem modifikovat obsah databáze [55]. Tabulka 6 obsahuje informaci, jakým způsobem se frameworky s danou hrozbou umí vypořádat.

Technologie	CORS	Autentizace	SQL Injection
Spring	Anotace	Spring Security	Spring Data JPA
Django	Knihovna	Auth Middleware	Django ORM
Node.js (Express.js)	Knihovna	Passport.js	Vlastní řešení
Rails	Knihovna	Vestavěná, Devise	Active Record
Vaadin	–	Spring Security	Spring Data JPA

Tabulka 6: Možnosti zabezpečení
Zdroj: vlastní zpracování

8.5 Rozšiřitelnost

Jak již bylo zmíněno, Django těží z popularity Pythonu. Pro jazyk existuje množství knihoven především pro vědecké a vzdělávací účely zaměřených na matematiku, statistiku a v poslední době velmi populární strojového učení. Výhodou je i možnost integrace se softwarem jako je Microsoft Excel. Node.js má za sebou silnou komunitu, která tvoří spoustu balíčků a knihoven využitelných v rámci jakéhokoliv frameworku postaveném nad Node.js. Využití takových knihoven bývá většinou nutnost, protože část funkcionalit frameworky či samotný nástroj Node.js v základu nenabízí. Rails nabízí gemy (knihovny) na široké spektrum problémů vytvořených početnou komunitou zaměřenou výhradně na webový vývoj. Problémem může být nedostatečná dokumentace vzhledem k množství a rychlosti vývoje gemů. Hlavní výhodou Vaadinu je možnost napojení na Spring Framework, což zajistí především přístup do databáze a zabezpečení.

Internacionalizace a lokalizace

Cílem internacionalizace a lokalizace je poskytnout obsah webové aplikace v jazyce a formátu korespondujícím s uživatelovou národností. Mezi běžně lokalizovatelné části aplikace patří možnost automatizovaně překládat textové řetězce, přizpůsobit formát data a času, formát měny a formát telefonních čísel. Tabulka 7 obsahuje informace o tom, jestli daný framework obsahuje nástroj pro zmíněné části lokalizace.

Technologie	Překlad Stringů	Datum a čas	Měna	Tel. čísla
Spring	ANO	ANO	ANO	ANO
Django	ANO	ANO	ANO	ANO
Node.js (Express.js)	npm balíček	ANO	ANO	ANO
Rails	ANO	ANO	gem	gem
Vaadin	ANO	ANO	ANO	ANO

Tabulka 7: Možnosti internacionalizace a lokalizace
Zdroj: [22] [5] [27] [49] [6]

9 Shrnutí výsledků

V této kapitole se nachází souhrn získaných poznatků v průběhu implementace a srovnání frameworků. Jsou zde zmíněny výhody a nevýhody jednotlivých technologií a z toho plynoucí vhodné typy aplikací.

Spring

Obecně by se dalo říci, že díky své rozmanitosti je Spring Framework vhodný na jakýkoliv typ aplikace. To ovšem může být zavádějící, protože jsou typy aplikací, které by zdaleka nevyužili všechny možnosti frameworku. Z níže popsaných výhod a nevýhod by se dalo říci, že Spring se hodí především na enterprise aplikace pracující s velkým množstvím dat. Takové aplikace často potřebují vysoké zabezpečení, což Spring splňuje na výbornou. Využití pro malé aplikace může být díky své komplexnosti z edukativních důvodů.

Výhody

- Nástroj Spring Boot – ulehčení konfigurace projektu
- Zabezpečení aplikace pomocí Spring Security
- Dependency Injection – jednoduché vkládání závislostí komponent
- Konfigurace pomocí anotací
- Komplexní oficiální dokumentace zahrnující široké spektrum problémů
- Testovatelnost

Nevýhody

- Mnoho možných řešení na jeden problém – Spring nabízí mnoho řešení na daný problém a chybí jasná konvence, co by měl vývojář použít
- Na internetu je sice spousta neoficiálních návodů a tutoriálů, ale řešení mohou být nesprávná nebo neaktuální
- Příliš komplikovaný nástroj pro vývoj menších aplikací – nadužívání různých komponent a vrstev vede k redundancím nebo duplicitám v kódu
- Java má poměrně složitou syntaxi
- Z výše popsaných nevýhod vyplývá pozvolná křivka učení

Django

Django může být stejně jako Spring zbytečně komplikované řešení pro malé aplikace, na druhou stranu je to ideální framework pro rychlý vývoj aplikace, která má za cíl být dostatečně zabezpečená a lehce udržovatelná. Vzhledem k popsaným výhodám a nevýhodám je typickým příkladem takové aplikace například CRM systém vyžadující administrátorské rozhraní a zároveň spravující množství citlivých dat. Díky množství knihoven je využití tohoto frameworku vhodné i pro aplikace zabývající se datovou analýzou nebo strojovým učením. Na druhou stranu pro svou povahu čistého monolitu není framework vhodný pro malé aplikace – například mikroslužby.

Výhody

- Jednoduchá syntaxe Pythonu, což napomáhá ke strmě rostoucí křivce učení
- Skvělá dokumentace – navíc je oficiální dokumentace i v českém jazyce
- Django REST Framework – skvělý nástroj na vývoj a správu API (přes GUI)
- Vestavěný ORM framework
- Django Admin – správa backendu přes GUI, není nutné vyvíjet zvlášť administrátorskou část u určitých typů projektů
- Velké možnosti rozšíření díky knihovnám Pythonu – především matematické a vědecké knihovny

Nevýhody

- Složitější počáteční konfigurace Pythonu a Djanga
- Společně s nesprávně zvolenou architekturou a nižším výkonem Pythonu může být aplikace pomalá
- Nutnost konfigurace každé funkcionality, což může zpomalovat vývoj
- Monolitická architektura – nevhodná pro menší projekty nebo mikroslužby

Node.js

Node.js je nástroj vhodný pro vývoj real-time aplikace, kde jsou data neustále sdílena mezi klientem a serverem. Jedná se například o aplikace typu online hry, chat, fórum a obecně se jedná o aplikace neustále vyžadující aktuální data. Nástroj je vhodný i na vývoj aplikace pomocí mikroslužeb. Node.js je i vhodný nástroj pro programování multiplatformních mobilních aplikací. Nástroj naopak není vhodný pro aplikace pracující s citlivými daty kvůli nižšímu zabezpečení. Problémy mohou nastat i u aplikací náročných na výkon CPU. Příkladem mohou být aplikace pro práci s videem nebo grafikou.

Výhody

- Pokud má vývojář předchozí zkušenosti s JavaScriptem, je učící křivka Node.js strmě rostoucí
- Sdílený codebase frontedu a backendu
- Díky event-driven architektuře je nástroj ideální pro REST API
- Práce s NoSQL databázemi – jednotný formát dat (JSON) v databázi, na backendu i na frontendu
- Díky neblokujícímu I/O modelu je možné zpracovávat více souběžných událostí zároveň
- Ke spuštění aplikace není potřeba vlastní webový server

Nevýhody

- Chabé zabezpečení – Node.js v základu neposkytuje nástroje pro autentizaci a ochranu proti běžným útokům
- Kvůli běhu v jednom vlákně je nástroj nevhodný pro práci s daty jako jsou obrázky a videa
- Nutná znalost asynchronního programování, callbacků a promis – pro vývojáře neznalé JavaScriptu vede tato nutnost naopak k pozvolné učící křivce
- Nízká výkonnost při práci náročné na CPU
- Špatná udržovatelnost projektu díky nekonzistencím nebo neaktuálnosti balíčků

Rails

Rails je ideální volba pro projekty vyžadující nižší rozpočet a zároveň mající hraniční termíny. Typicky se může jednat o eshopy na zakázku. Rychlost vývoje pomocí Rails je pro zkušenější vývojáře velmi rychlý a za krátkou dobu je možné prezentovat plně funkční webovou aplikaci. Naopak je těžké aplikaci udržovat a debuggovat vzhledem k tomu, že se spousta logiky děje „na pozadí“. Framework se tedy nehodí na komplexnější enterprise aplikace.

Výhody

- Jednoduchá syntaxe Ruby, což napomáhá ke strmě rostoucí křivce učení
- Convention over configuraition – většina funkcionalit je předkonfigurovaná a funguje na pozadí – výhoda pro zkušené vývojáře
- Generátory hotových řešení pomocí pár příkazů umožňují opravdu rychlý vývoj webové aplikace
- Vyžaduje malé množství kódu – implementace je na první pohled přehlednější
- Active Record – funguje jako vlastní ORM, stará se o migraci dat a generování modelu a tabulek
- Silná komunita zaměřena na webový vývoj – neustálý vývoj nových gemů

Nevýhody

- Convention over Configuration – tím, že spousta funkcionalit se odehrává na pozadí je potřeba zkušených vývojářů pro porozumění procesu a odhalení chyb
- Málo flexibilní – framework nutí vývojáře do předem hotového řešení, o kterém jsou autoři Rails přesvědčeni, že je správné
- Takové množství abstrakce může vést ke složitějšímu hledání chyb
- Nevhodné pro mobilní aplikace či enterprise aplikace – jazyk a komunita zaměřené výhradně na webový vývoj
- Občasná nekonzistence frameworku a gemů
- V práci s více vlákny se může framework stát náročným na výpočetní výkon a paměť

Vaadin

Vaadin je vhodný pro programátory zvyklé na vývoj desktopových aplikací v Javě. Díky tomu je Vaadin dobrou volbou pro tvorbu RIA (Rich Internet Application). Vaadin je také možné použít jako přidanou komponentu do již existujícího firemního sytému. Naopak framework není vhodné použít na aplikace vyžadující časté sdílení dat a pro aplikace využívající REST API.

Výhody

- Sdílený codebase frontendu a backendu
- Díky vlastní UI Component API je Vaadin vhodný na vývoj RIA (aplikace bohaté na UI komponenty)
- Možnost integrace se Springem – zajištění bezpečnosti aplikace a přístupu do databáze
- Jednoduchý vývoj PWA – Progresivní webová aplikace – lze spouštět z desktopu

Nevýhody

- Díky principu vývoje a architektury je Vaadin nevhodný pro aplikace využívající REST API
- Kvůli renderování komponent na serveru je vyžadován vysoký výkon na server
- Slabá komunita, dokumentace řeší jen dílčí části problémů
- Z toho vyplývá pozvolná křivka učení
- Pro plné využití a podporu je potřeba zakoupit licenci

10 Závěry a doporučení

Cílem této diplomové práce bylo v první části seznámit čtenáře s vývojem serverové části webové aplikace, o čemž pojednávají kapitoly 2–6. V 6. kapitole byly navíc představeny frameworky a nástroje pro tvorbu serverové části aplikace. Pomocí těchto technologií byla naimplementována ukázková aplikace a vybrané aspekty implementace byly popsány společně s ukázkami kódu v kapitole 7. V kapitole 8 jsou nejdříve popsány rozdíly mezi jednotlivými frameworky z obecnějšího hlediska. Jednalo se o dokumentaci a podporu, rychlost učení, zabezpečení, testování a rozšiřitelnost. V poslední kapitole je celkové shrnutí výhod a nevýhod jednotlivých frameworků, včetně informace, na jaký druh aplikace je vhodné použít daný framework, a naopak, na jaké typy aplikací framework vhodné použít není.

Spring Framework se díky své komplexnosti hodí na robustní enterprise aplikace pracující s velkým množstvím dat. Naopak využití pro menší aplikace má význam například ze vzdělávacích důvodů.

Framework Django tvoří pevný základ pro administrátorské aplikace díky připravenému autentizačnímu modelu a administrátorské části. Díky rozšiřitelnosti Pythonu je možné framework použít i pro webové aplikace zaměřené na vědecké výpočty či strojové učení. Kvůli své monolitické architektuře ovšem nelze framework použít pro aplikace založené na mikroslužbách.

Node.js je nástroj vhodný pro menší aplikace s častým sdílením dat. Je ale potřeba zvážit přístup k zabezpečení.

Rails je díky svému přístupu Convention over Configuration ideální volbou pro projekty, které je potřeba vyvíjet v krátkém časovém horizontu.

Vaadin není mezi vývojářskou komunitou příliš rozšířený, ale své využití si najde především při tvorbě RIA aplikací. Naopak framework nelze použít pro dnes rozšířený způsob vývoje pomocí REST API.

11 Seznam použitých zdrojů

[1] Introduction to the server side MDN Web Docs. [online]. [cit. 2018-10-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction.

[2] D. A. A. Eraso a F. A. Isaza, Hibernate and spring – An analysis of maintainability against performance, Revista Facultad de Ingeniería, 2016, roč. 0, č. 80, s. 97–108.

[3] B. Ciubotaru a G.-M. Muntean, Advanced Network Programming – Principles and Techniques: Network Application Programming with Java. London: Springer-Verlag, 2013.

[4] Přehled ASP.NET Core MVC. [online]. [cit. 2018-10-02]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/mvc/overview.

[5] Django documentation. [online]. [cit. 2019-03-08]. Dostupné z: https://docs.djangoproject.com/en/2.0/.

[6] Vaadin Flow, Vaadin. [online]. [cit. 2019-03-20]. Dostupné z: https://vaadin.com/docs/flow/Overview.html.

[7] What Is a Web Server vs. an Application Server, NGINX. [online]. [cit. 2018-10-02]. Dostupné z: https://www.nginx.com/resources/glossary/application-server-vs-web-server/.

[8] HTTP Response Status Codes, REST API Tutorial. [online]. [cit. 2018-11-11]. Dostupné z: https://restfulapi.net/http-status-codes/.

[9] C. Wodehouse, The API Economy. Upwork, 2016 [online]. [cit 2018-10-10]. Dostupné z: https://pages.upwork.com/rs/518-rkl-392/images/16-0916-api-for-marketers%20final.pdf

[10] Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST). [online]. [cit. 2018-10-10]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[11] What is REST – Learn to create timeless RESTful APIs. [online]. [cit. 2018-10-10]. Dostupné z: https://restfulapi.net/.

[12] Understanding : REST. [online]. [cit. 2018-10-10]. Dostupné z: https://spring.io/understanding/REST.

[13] Caching in Web Applications, Code by Amir. [online]. [cit. 2018-10-02]. Dostupné z: https://www.codebyamir.com/blog/caching-in-web-applications.

[14] The Most Common Authentication Methods in Web Application Development, Medium. [online]. [cit. 2018-10-10]. Dostupné z: https://medium.com/@pavithraranathunga/the-most-common-authentication-methods-in-web-application-development-35845ecb1da0

[15] A Relational Database Overview, The Java Tutorials. [online]. [cit. 2018-10-15]. Dostupné z: https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html.

[16] SQL vs. NoSQL: What's the difference?,Upwork. [online]. [cit. 2019-03-20]. Dostupné z: https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/.

[17] Pokorný J. (2018) Integration of Relational and NoSQL Databases. In: Intelligent Information and Database Systems. ACIIDS 2018. Lecture Notes in Computer Science, vol 10752. Springer, Cham

[18] Server-side web frameworks, MDN Web Docs. [online]. [cit. 2018-10-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks.

[19] Compiled versus interpreted languages, IBM. [online]. [cit. 2018-10-02]. Dostupné z: https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zappldev/zappldev_85.htm?view=embed.

[20] HORSTMANN, Cay S. Core Java. Eleventh edition. Boston: Pearson, 2019. ISBN 0-13-516630-6.

[21] JOHNSON, Rod. Expert one-on-one J2EE design and development. Indianapolis, IN: Wrox, c2003. ISBN 978-0764543852

[22] Pro spring 5: an in-depth guide to the spring framework and its tools. New York, NY: Springer Science+Business Media, 2017. ISBN 978-1484228074.

[23] Python Software Foundation, Python.org. [online]. [cit. 2018-10-02]. Dostupné z: https://www.python.org/psf/.

[24] Django Overview, The Django Book, [online]. [cit. 2018-10-02]. Dostupné z: https://djangobook.com/.

[25] About Ruby, Ruby. [online]. [cit. 2018-10-30]. Dostupné z: https://www.ruby-lang.org/en/about/.

[26] Getting Started with Rails, Ruby on Rails Guides. [online]. [cit. 2018-10-27]. Dostupné z: https://guides.rubyonrails.org/getting_started.html.

[27] JavaScript, MDN Web Docs. [online]. [cit. 2018-11-09]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[28] Building web apps with Node.js, Web Code Geeks, [online]. [cit. 2018-11-09]. Dostupné z: https://www.webcodegeeks.com/javascript/node-js/building-web-apps-with-node-js/.

[29] React – A JavaScript library for building user interfaces. [online]. [cit. 2019-03-08]. Dostupné z: https://reactjs.org/index.html.

[30] Using the @SpringBootApplication Annotation. [online]. [cit. 2019-03-13]. Dostupné z: https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html.

[31] Spring MVC 4.0 RESTFul Web Services Simple Example, Programming Free [online]. [cit. 2019-11-09]. Dostupné z: http://www.programming-free.com/2014/01/spring-mvc-40-restful-web-services.html.

[32] Building REST services with Spring. [online]. [cit. 2019-03-08]. Dostupné z: https://spring.io/guides/tutorials/rest/.

[33] Spring Security Reference [online]. [cit. 2019-03-08]. Dostupné z: https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#jc-authentication.

[34] Caching [online]. [2019-03-08]. Dostupné z: https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html.

[35] Testing the Web Layer [online]. [cit. 2019-03-08]. Dostupné z: https://spring.io/guides/gs/testing-web/.

[36] Django Web Framework (Python), MDN Web Docs [online]. [cit. 2018-10-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django.

[37] Views, Django REST framework [online]. [cit. 2019-03-08]. Dostupné z: https://www.django-rest-framework.org/api-guide/views/.

[38] Authentication, Django REST framework [online]. [cit. 2019-03-08]. Dostupné z: https://www.django-rest-framework.org/api-guide/authentication/.

[39] Django's cache framework, Django documentation [online]. [cit. 2019-03-08]. Dostupné z: https://docs.djangoproject.com/en/2.0/topics/cache/.

[40] Testing in Django, Django documentation [online]. [cit. 2019-03-08]. Dostupné z: https://docs.djangoproject.com/en/2.0/topics/testing/.

[41] Setting up a Node development environment, MDN Web Docs [online]. [cit. 2019-03-08]. Dostupné z: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/development_environment.

[42] Manual, Sequelize, The node.js ORM for PostgreSQL, MySQL, SQLite and MSSQL [online]. [cit. 2019-03-08]. Dostupné z: http://docs.sequelizejs.com/.

[43] All About Node.Js You Wanted To Know, codeburst [online]. [cit. 2019-03-14]. Dostupné z: https://codeburst.io/all-about-node-js-you-wanted-to-know-25f3374e0be7.

[44] Passport.js, Passport.js. [online]. [cit. 2019-03-08]. Dostupné z: http://www.passportjs.org/.

[45] Memory-cache, npm. [online]. [cit. 2019-03-08]. Dostupné z: https://www.npmjs.com/package/memory-cache.

[46] Mocha.js. [online]. [cit. 2019-03-08]. Dostupné z: https://mochajs.org/.

[47] Active Record Basics, Ruby on Rails Guides [online]. [cit. 2019-03-15]. Dostupné z: https://guides.rubyonrails.org/active_record_basics.html.

[48] Ruby on Rails Beginner Adventures: Simple Blog Application, Medium [online]. [cit. 2019-03-15]. Dostupné z: https://medium.com/@1sherlynn/ruby-on-rails-beginner-adventures-simple-blog-application-6dffd6dcb11a.

[49] Ruby on Rails, Ruby on Rails [online]. [cit. 2019-03-15]. Dostupné z https://github.com/rails/rails.

[50] Flexible authentication solution for Rails with Warden, plataformatec/devise [online]. [cit. 2019-03-20]. Dostupné z: https://github.com/plataformatec/devise

[51] Action caching for Action Pack, Ruby on Rails [online]. [cit. 2019-03-20]. Dostupné z: https://github.com/rails/actionpack-action_caching.

[52] Testing Rails Applications, Ruby on Rails Guides. [online]. [cit. 2019-03-17]. Dostupné z: https://guides.rubyonrails.org/testing.html#the-test-database.

[53] Getting Started With Vaadin, dzone.com. [online]. [cit. 2019-04-05]. Dostupné z: https://dzone.com/refcardz/getting-started-vaadin?chapter=1.

[54] Spring, GitHub. [online]. [cit. 2019-04-07]. Dostupné z: https://github.com/spring-projects.

[55] OWASP, The OWASP Foundation [online]. [cit. 2019-04-07] Dostupné z: https://www.owasp.org/index.php/Main_Page.

[56] JetBrains, Developer Tools for Professionals and Teams, JetBrains [online]. [cit. 2019-04-08]. Dostupné z: https://www.jetbrains.com/.

[57] Visual Studio Code, Code Editing. Redefined [online]. [cit. 2019-04-08] Dostupné z: https://code.visualstudio.com/.

[58] E. F. Inc, The Platform for Open Innovation and Collaboration, The Eclipse Foundation [online]. [cit. 2019-04-08]. Dostupné z: https://www.eclipse.org/.

[59] A hackable text editor for the 21st Century, Atom [online]. [cit. 2019-04-08]. Dostupné z: https://atom.io/.

[60] Vim online, Vim [online] [cit. 2019-04-08]. Dostupné z: https://www.vim.org/.

12 Přílohy

1. CD se zdrojovými kódy

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Kozáková Eva	[redacted]	I1700489

TÉMA ČESKY:

Specifikace server-side technologií, jejich implementace a porovnání

TÉMA ANGLICKY:

Specification of server-side technologies, its implementation and comparation

VEDOUCÍ PRÁCE:

doc. Ing. Filip Malý, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cílem práce je definovat serverovou část webové aplikace a specifikovat používané technologie a postupy. Dále se práce bude věnovat popisu širšímu výběru frameworků pro tvorbu backendu. V praktické části bude implementována aplikace ve vybraných frameworcích. Dalším cílem je porovnání těchto frameworků a následné doporučení, jaký framework je vhodný pro určité druhy aplikací.

1. Úvod
2. Vývoj webové aplikace
3. Server
4. Databázový systém
5. Middleware
6. Jazyky a Frameworky
7. Implementace
8. Porovnání
9. Shrnutí výsledků
10. Závěry a doporučení
11. Seznam použité literatury

SEZNAM DOPORUČENÉ LITERATURY:

Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools (Cosmina, I., Harrop, R., Schaefer, C., Ho, C.)
Beginning Django: Web Application Development and Deployment with Python (Daniel Rubio)

Podpis studenta: [redacted] **Datum:** 9. 10. 2018

Podpis vedoucího práce: [redacted] **Datum:** 9.10.2018