VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

GENERÁTOR ZEFEKTIVŇUJÍCÍ TVORBU A UDRŽOVATELNOST SINGLE-PAGE APLIKACÍ

SINGLE-PAGE APPLICATION GENERATOR FOR IMPROVING MAINTAINABILTY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. NORBERT ĎURČANSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. LIBOR POLČÁK, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Ďurčanský Norbert, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Generátor zefektivňující tvorbu a udržovatelnost single-page aplikací**
Single-Page Application Generator for Improving Maintainabilty
Kategorie: Informační systémy
Zadání:

1. Popište aktuálně používané frameworky pro tvorbu single-page aplikací, zaměřte se na spolupráci s backendem Spring Boot.
2. Nastudujte nástroje usnadňující tvorbu, testování, udržování a nasazování single-page aplikací.
3. Identifikuje problémová místa, která mohou zpomalit zahájení vývoje takové aplikace a navrhněte způsob, který povede k zefektivnění tvorby těchto aplikací.
4. Implementujte generátor aplikací využívající navržené postupy.
5. Vytvořte ukázkovou aplikaci využivající generátor. Zaměřte se na pochopitelnost a možnost využití při výuce v předmětu WAP - Internetové aplikace.
6. Zhodnoťte vytvořená řešení, identifikujte nedostatky a navrhněte možná vylepšení.

Literatura:

- Ambler, T., Cloud, N.: JavaScript Frameworks for Modern Web Dev. Berkeley, CA: Apress, 2015.
- Spring boot, https://spring.io/projects/spring-boot, [[online]] navštíveno 2018-08-30.
- Pasquali, S.: Mastering Node. js. Packt Publishing Ltd, 2nd edition, 2017. ISBN 978-1785888960.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Polčák Libor, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 23. října 2018

Abstrakt

Táto diplomová práca sa zaoberá tvorbou generátora *single-page* aplikacií. Pred jeho vytvorením bolo potrebné identifikovať problémové miesta, ktoré spomaľujú vývoj, a popísať nástroje, ktoré uľahčujú tvorbu, testovanie, udržovanie a nasadzovanie single-page aplikácií. Na základe uvedených informácií je navrhnutý a implementovaný generátor *Create Sbspa*, ktorý zefektívňuje tvorbu single-page aplikácií a pomáha odstraňovať problémové miesta vývoja. K tomu využíva generovanie konfigurácií a kódu pomocou sémantických šablón. Generátor je dostupný pomocou uživateľkého rozhrania, ktoré rozdeľuje šablóny do skupín podľa použitia. Pri návrhu nástroja je kladený dôraz na jeho prehľadnosť a jednoduchú rozšíriteľnosť o nové funkcionality. Z tohto dôvodu je súčasťou práce taktiež návrh a implementácia vzorovej aplikácie, ktorá slúži ako ukážka vlastností a výhod generátora.

Abstract

This diploma thesis deals with developing generator for single-page applications. Before developing the application it was necessary to identify problem areas that prevent the development and describe tools that make it easy to create, test, maintain, and deploy single-page applications. Based on the obtained information, the generator *Create Sbspa* is designed and implemented to efficiently create single-page applications and help to eliminate development problems. It generates configuration and code from semantic templates. The generator is available through a user interface that splits the templates into the groups by applicability. The generator was designed with the need for simplicity and clarity to enable efficient integration with new features. This work also includes design and implementation of the example app which shows features and benefits of the generator.

Kľúčové slová

single-page aplikácie, SPA, generovanie aplikácií, problémové miesta vývoja, škálovanie aplikácií, testovanie aplikácií, nasadzovanie aplikácií, udržovanie aplikácií, automatizácia zostavenia, kontrola kódu, sémantické šablóny, Gradle, React, GraphQL, Spring Boot, Docker, Typescript, CI

Keywords

single-page applications, SPA, application generator, development problems, application scaling, application build, application testing, application deployment, maintenance, continuous integration, code analyzer, semantic templates, Gradle, React, GraphQL, Spring Boot, Docker, Typescript, CI

Citácia

ĎURČANSKÝ, Norbert. *Generátor zefektivňující tvorbu a udržovatelnost single-page aplikací*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Libor Polčák, Ph.D.

Generátor zefektivňující tvorbu a udržovatelnost single-page aplikací

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Libora Polčáka, Ph.D.. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Norbert Ďurčanský
27. apríla 2019

Poďakovanie

Týmto by som chcel poďakovať môjmu vedúcemu Ing. Liborovi Polčákovi, Ph.D. za jeho ochotu, profesionálny prístup a odbornú pomoc pri vedení tejto diplomovej práce.

Obsah

Kapitola 1

Úvod

Väčšina služieb, kvôli ktorým sme museli chodiť na úrady, do banky či do obchodu sa pomaly presúva na Internet a webové aplikácie, ako ich dnes poznáme, sa stali neoddeliteľnou súčasťou nášho života. Služby ako online bankovníctvo, nakupovanie, hranie hier, rezervácia leteniek sú využívané ľuďmi po celom svete. Rovnako zabezpečujú komunikáciu s firmami, ktoré si uvedomujú, že pre zaistenie spokojnosti zákazníkov a úspešného podnikania je potrebné služby[1] vytvárať jednoduché a pre užívateľa intuitívne. Webové aplikácie sa preto stávajú veľmi rozšírené. Uľahčujú nám život a poskytujú nástroje pre efektívnejšiu prácu. Tento trend pomáha nárastu dopytu po vývoji webových aplikácií, ale taktiež vedie k zvyšovaniu tlaku na rýchle doručenie, kvalitu, prenositeľnosť a bezpečnosť softvérového produktu. Pre vytvorenie vysoko kvalitného kódu, sa musia vývojári usilovať zaistiť jeho udržateľnosť, prehľadnosť, testovateľnosť a často krát aj adaptáciu na zmeny.

Jedným z rozšírených spôsobov implementácie moderných webových aplikácií sú *single-page* aplikácie (SPA). Na väčšine stránok sa nachádza veľké množstvo obsahu, ktoré sa nemení. Môžu to byť rôzne hlavičky, pätičky alebo dokonca celé šablóny. *Single-page* aplikácia využíva vlastnosť, že sa obsah na stránke opakuje a nahrádza len časti, ktoré sa zmenili. Takto je celá aplikácia rýchlejšia a užívateľské rozhranie výrazne plynulejšie.

Cieľom tejto diplomovej práce je zoznámiť sa s fungovaním *single-page* aplikácií a naštudovať vybrané platformy, ktoré sa používajú pri tvorbe webových služieb. Podstatnou časťou je ich spolupráca s platformou *Spring Boot*, ktorá umožňuje jednoducho vytvoriť samostatnú produkčnú *Spring*[2] aplikáciu, ktorá nevyžaduje zložité konfigurácie a je spustiteľná okamžite. Taktiež je dôležitou úlohou identifikovať a analyzovať problémové miesta vývoja, ktoré spôsobujú spomalenie začiatku a vedú k postupnej degradácií celého projektu. Keďže veľkú časť z nich je možné redukovať, projekt popisuje nástroje, ktoré uľahčujú tvorbu, testovanie, správu a nasadzovanie *single-page* aplikacií.

Na základe získaných informácií je navrhnutý generátor *Create Sbspa*, ktorý pomáha zefektívniť tvorbu *single-page* aplikácií. Výsledné riešenie je implementované v podobe knižnice pre *Node.js*, ktorá je dostupná pomocou nástroja NPM[3]. Umožňuje vygenerovať štruktúru projektu a poskytnúť nástroje pre jeho efektívnejší vývoj. Veľký dôraz, ktorý je pri návrhu kladený, spočíva v pochopiteľnosti, rozšíriteľnosti a jednoduchosti aplikácie, ktorá nevyžaduje zložité rozhranie a konfiguráciu.

[1] Napríklad bankové služby, emailové služby či sociálne siete.

[2] https://spring.io/

[3] https://www.npmjs.com/package/create-sbspa

Create Sbspa je použitý pri návrhu a implementácií vzorovej aplikácie, ktorá ho použila na odstránenie identifikovaných problémových miest vývoja a bude použiteľná pre účely projektu TARZAN.

Diplomová práca je rozdelená do niekoľkých kapitol. V kapitole č. 2 sú popísané *single-page* aplikácie a ich prínosy do moderného vývoja webových aplikácií. Veľká časť je zameraná popisu dvoch aktuálne používaných platforiem a popisu možností spolupráce s frameworkom *Spring Boot*.

Kapitola č. 3 sa zaoberá samotným vývojom *single-page* aplikácií. Zameraná je predovšetkým na identifikáciu problémových miest v rôznych častiach vývojového procesu softwarového produktu. Na základe týchto informácií sú uvedené nástroje zjednodušujúce vývoj a následnú správu webovej aplikácie.

V kapitole č. 4 je špecifikovaný návrh nástroja pre zefektívnenie tvorby *single-page* aplikácií. Podstatnú časť tvorí popis štruktúry a použitých nástrojov pri tvorbe *backend* a *frontend*. Ďalej sú v nej definované požiadavky na samotnú aplikáciu, možnosti použitia a rozšíriteľnosti.

Na základe návrhu je v kapitole č. 5 popísaná samotná implementácia generátora *Create Sbspa*. Sú v nej uvedené implementačné detaily navrhnutých častí, kľúčové triedy a vzťahy medzi nimi. Zameraná je prevažne na popis najdôležitejších súčastí, špecifikovaných v návrhu, ktoré zeefektívňujú proces vývoja a udržateľnosti *single-page* aplikácií. Taktiež sú v nej uvedené nástroje potrebné k behu a štruktúra generovaných častí programu. V závere kapitoly sú popísané vzorové materiály, ktoré boli vytvorené pre účely možnosti použitia pri vyuke v predmete WAP.

Kapitola č. 6 je venovaná návrhu a implementácií vzorovej aplikácie, ktorá bude použiteľná pre účely projektu TARZAN a využije dostupné nástroje *Create Sbspa*, ktoré zefektívňujú vývoj.

Kapitola č. 7 sa zaoberá testovaním implementovaného nástroja. Rozdelená je do dvoch častí. V prvej časti je testovaná funkčnosť vygenerovanej *single-page* aplikácie, ktorá je porovnávaná s aplikáciami vygenerovanými inými nástrojmi. Druhá časť je zameraná na porovnanie funkcionalít a zhodnotenie prínosov vybraných generátorov do vývoja *single-page* aplikácií. V závere kapitoly sú následne uvedené výsledky a porovnanie vybraných nástrojov.

Kapitola 2

Popis vybraných nástrojov pre tvorbu single-page aplikácií

Webové aplikácie pomaly nahrádzajú staré desktopové aplikácie. Sú pohodlnejšie na použitie, ľahko sa aktualizujú a nie sú viazané na jedno zariadenie. Aj keď sa užívatelia často krát presúvajú z webových aplikácií založených na prehliadači na mobilné aplikácie, dopyt po komplexných aplikáciách je obrovský a stále rastie. Tento trend pomáha nárastu *single-page* aplikácií, ktorých počet sa neustále zvyšuje. Väčšina z nás ich už pravdepodobne využíva, či už sú to emailové služby (Google) alebo sociálne siete (Facebook) [20].

Vytvorenie webovej aplikácie zahŕňa použitie jedného z dvoch hlavných vzorov. Vzor *multi-page* alebo *single-page* aplikácie. Samozrejme, že oba modely majú svoje výhody a nevýhody. Táto práca sa zaoberá vzorom *single-page* aplikácií, ktorých spoločnou vlastnosťou pri prechode medzi stránkami je, že sa stránka celá neobnoví. Dôjde len k načítaniu nových dát a veľké množstvo obsahu zostane nezmenené. Načíta sa prvotným dotazom zaslaným na server. Klient prijme HTML stránku a prehliadač sa postará o vykreslenie užívateľského rozhrania (UI). Ďalšie dotazy na server slúžia prevažne na získanie nových dát a ako reakcia na akcie užívateľa, prekreslením UI [27, 20].

V tejto kapitole sú popísané aktuálne používané frameworky pre tvorbu *single-page* aplikácií. Zameraná je prevažne na ich princípy a prínosy do moderného vývoja. Dôležitou časťou je popis spolupráce s platformou *Spring Boot*, ktorá bude použitá pri návrhu aplikácie.

2.1 AngularJS

Veľký rozdiel pri riešení vývojových výziev *single-page* aplikácií od ostatných alternatív prináša platforma *AnglularJS*. Jednoduchý framework s Javascript kódom pre správu stavu aplikácie a rýchlu odozvu na interakcie užívateľa. *AngularJS* sa stal populárnym medzi webovými i UI vývojármi. Vďaka možnosti modifikácie HTML tagov a implementácií metód pre ich obsluhu je úprava webového rozhrania jednoduchá. Navyše vývojárom umožňuje *test-driven development*[1], čo ich pomáha motivovať ku písaniu testov s vysokou spoľahlivosťou a rozsiahlou platnosťou. Hlavnou charakteristikou, ktorou sa odlišuje od iných frameworkov je spôsob, ktorým umožňuje vývojárom tvoriť webové aplikácie. Namiesto imperatívneho spôsobu vývoja, na ktorý sú vývojári zvyknutí, využíva deklaratívny spôsob, ktorý prináša množstvo benefitov [18, 19].

[1] https://www.pluralsight.com/guides/introduction-to-angular-test-driven-development

Pod slovíčkom programovanie si väčšina ľudí predstaví **imperatívny** prístup. Vývojár zadáva počítaču inštrukcie, ako má niečo vykonať. Výsledok je potom úspešné prevedenie inštrukcií. **Deklaratívny** prístup sa na druhú stranu sústredí na vyjadrenie ako má výsledok vyzerať a prevedenie jednotlivých krokov necháva na počítač [19, s. 156-157].

Ak je webová aplikácia vytvorená pomocou deklaratívneho prístupu, zodpovednosť kontroly toku dát v rámci aplikácie prevezme jej rozhranie. To je možné vďaka *Angular* direktívam, ktoré obohacujú DOM elementy alebo existujúce inštancie komponent o nové funkcionality [2]. Existujú 3 rôzne druhy:

- komponenty,
- štrukturálne direktíva,
- atribútové direktíva.

Komponenty sú špeciálny druh direktív, ktorý používa jednoduchšiu konfiguráciu vhodnú pre popis štruktúry aplikačných komponent. Vďaka jednoduchej štruktúre sú častokrát použité na architektúru postavenú z komponent. Na druhú stranu ich však nie je možné použiť na volanie akcií alebo reagovanie na udalosti. **Štrukturálne direktíva** sú zodpovedné za rozmiestnenie HTML elementov a zmenu DOM štruktúry, zvyčajne pridaním, odstránením alebo manipuláciou elementov. **Atribútové direktíva** umožňujú meniť vzhľad alebo správanie elementov, komponent alebo iných direktív [2, 14].

Správa dát a závislostí vo frameworku *AngularJS* eliminuje množstvo kódu, ktoré musí vývojár napísať, čo ho robí spojením s HTML významným deklaratívnym jazykom pre tvorbu webových aplikácií.

Komplexné aplikácie prestávajú byť komplexné ak sa k nim nepristupuje ako k jedinej entite, ale prostredníctvom kolekcie malých komponent, ktoré spolu spolupracujú. *Angular* moduly sú základný stavebný kameň *Angular* projektov a poskytujú praktický vzor štruktúry aplikácií. Každá aplikácia je samostatný modul a každý modul môže špecifikovať ďalšie v podobe závislostí. Takto je potom možné členiť aplikáciu na menšie, vzájomne nezávislé, moduly [16, 19].

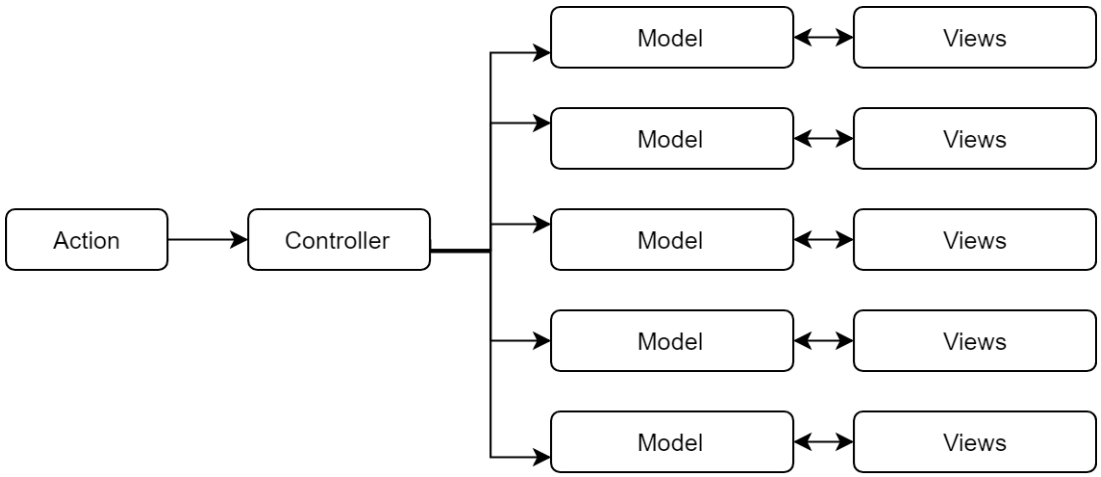
MVC

AngularJS v porovnaní s doposiaľ používanými imperatívnymi frameworkami[2] pre tvorbu webových aplikácií, je všestranný, klientsky orientovaný MVC[3] framework. Poskytuje implementáciu pre *controllers*, *views* a model. Za model aplikácie sú považované JSON objekty[4], šablóna (respektíve výstup generovania) predstavuje *view* a Angular komponent, ktorý je napojený na DOM element, spája model a pohľad v roly *controller*. Pomocou konštrukčných prvkov sú vytvárané *single-page* aplikácie spoločne so smerovaním, šablónovaním, dátovým modelom, aplikačnou konfiguráciou a logikou pre komunikáciu s externými službami prostredníctvom AJAX a REST [18, 30].

MVC framework uľahčuje samotný proces vývoja užívateľského rozhrania. Množstvo kódu, ktoré je potrebné napísať sa výrazne zredukuje, pretože kód pre synchronizáciu medzi modelom a pohľadom už nie je potrebný[5]. Navyše prináša lepšiu čitateľnosť a udrža-

[2] Napríklad jQuery https://jquery.com/.
[3] https://www.futurehosting.com/blog/what-is-an-mvc-framework/
[4] https://www.json.org/json-cz.html
[5] *AngularJS* ho vygeneruje.

teľnosť kódu, vďaka jasnej separácií modelu od pohľadu [1]. Na obrázku 2.1 je zobrazená architektúra MVC.



Obr. 2.1: Architektúra *MVC* (prevzaté z [12]).

Modely a *views* spolu komunikujú v oboch smeroch, preto sa jedna o **obojsmerný tok dát**.

2.2 React

AngularJS bol vytvorený v začiatkoch vývoja webových aplikácií, kedy sa model DOM považoval za základný stavebný kameň aplikácie. Programátori písali HTML, CSS kód a funkcionalita sa pridávala pomocou Javascript. S myšlienkou nového prístupu prišiel framework *React*. Pôvodne bol vytvorený vývojármi vo firme Facebook, so zámerom vyriešiť problémy spojené s vývojom komplexných užívateľským rozhraním a dátami, ktoré sa dynamicky môžu meniť [21].

Aplikácie, ako sú tieto, zvyčajne pozostávajú z obojsmerného toku dát a šablón. *React* zmenil spôsob, akým sú aplikácie vytvárané a nastavil nové konvencie vývoja *frontend* [21]. Masívnym posunom paradigmy je zmena pohľadu na webovú aplikáciu a DOM. Namiesto použitia direktív a šablón HTML elementov, existuje iba Javascript aplikácia, ktorej výstupom je DOM. DOM sa preto stal výstupom a nie nástrojom pre zmeny. Koncept frameworku *React* pôsobí jednoducho a elegantne. Namiesto overovania všetkých komponent na zmeny (*AngularJS*) si *React* udržuje stav DOM ako veľký Javascript objekt. Ak užívateľ vykoná akciu, *React* vytvorí nový DOM objekt, ktorý porovná so starým a všetky zmeny vykreslí do UI. Taktiež poskytuje možnosť vytvárať zapuzdrené komponenty, ktoré ovládajú svoj vlastný stav a skladať ich do komplexných užívateľských rozhraní. Keďže logika komponent je namiesto šablón písaná výlučne v Javascript, je možné predávať dáta skrz aplikáciu a separovať tak stav od DOM [12].

React vďaka deklaratívnemu prístupu poskytuje možnosť jednoduchej tvorby užívateľských rozhraní. To znamená, že namiesto písania postupu krokov rozhrania, vývojár definuje rozhranie z hľadiska koncového stavu. Ak nastane transakcia s týmto stavom, React sa postará o aktualizáciu UI. Navyše umožňuje návrh nezávislých *views* pre rôzne stavy aplikácie, ktoré pri zmene dát aktualizujú iba zmenené komponenty. Deklaratívne *views* robia kód predvídateľnejším, čitateľnejším a jednoduchším na ladenie [12].

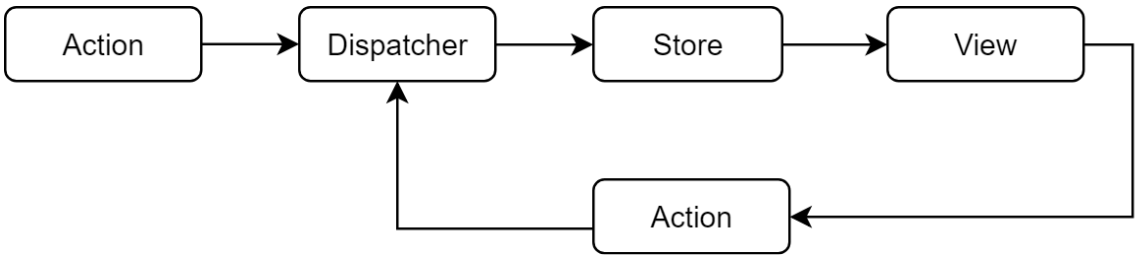
Užívateľské rozhrania sú tvorené pomocou komponent, ktoré sú definované ako funkcie alebo triedy. Ich vstupom sú vlastnosti a stav. Výstup je popis UI (podobný HTML zápisu). Je ich možné opätovne použiť a skladať z nich väčšie a komplexnejšie komponenty. Na rozdiel od funkcií, triedy môžu mať svoj vlastný stav pre uchovávanie údajov, ktoré sa dokážu v priebehu času meniť. Používajú sa väčšinou na miestach, kde je potrebné meniť stav aplikácie a reagovať na akcie užívateľa [12].

Flux

React je výlučne zameraný na vrstvu *view*. Práca s vrstvou modelu a *controller* preto vyžaduje použitie návrhového vzoru. Najznámejšia je architektúra *Flux* použitá firmou Facebook [29].

Flux úzko súvisí s frameworkom *React*. Je to aplikačná architektúra, ktorá popisuje, ako získavať dáta a ako komunikovať s *React* komponentami zmysluplným spôsobom. *Flux* nie je MVC model, pretože nevyužíva obojsmerný tok dát. Na rozdiel od toho obohacuje *React* komponenty prostredníctvom **jednosmerného toku dát**, čím pomáha lepšej správe stavu aplikácie [29, 21]. Na obrázku 2.2 je zobrazená architektúra *Flux*, ktorá je zložená zo 4 hlavných komponent:

- dispatcher,
- store,
- view,
- action.



Obr. 2.2: Architektúra *Flux* (prevzaté z [12]).

Dispatcher je možné pochopiť ako centrálny rozbočovač aplikácie. Na vstupe prijíma akcie a preposiela ich spoločne s dátami registrovaným príjemcom. Podstatnou časťou je možnosť implementácie funkcionality pre zaistenie správneho poradia odosielania, prípadne čakania na vykonanie určitej akcie. *Dispatcher* pri výskyte akcie na vstupe pošle dáta (tzv. *payload*) do **store**, ktorý je registrovaný pre konkrétnu akciu. Jeho hlavnou úlohou je aktualizácia *views*, správa business logiky a aktualizácia dát. **Views** sú bežné *React* komponenty, ktoré reagujú na zmeny aplikačného stavu prichádzajúce zo *store* a propagujú ich pomocou parametrov do komponent na nižšej úrovni. Veľkú úlohu pri predávaní dát zohrávajú **actions**. Sú to kolekcie funkcií, ktoré sú volané v rámci *views* alebo na inom mieste, kde sú vytvárané akcie zo vstupných parametrov. K akciám sa priradí typ a odošle sa do modulu *Dispatcher* [29, 33].

Jednosmernosť toku dát je zabezpečená vďaka nezávislosti uzlov modelu a presnej špecifikácií vstupov a výstupov všetkých komponent.

2.3 Spolupráca s frameworkom Spring Boot

V predchádzajúcich sekciách boli popísané princípy aktuálne používaných frameworkov pri tvorbe užívateľského rozhrania *single-page* aplikácií. Väčšina zmysluplných interakcií však vyžaduje server, ktorý môže zastávať viacero rolí. Najčastejšie obsluhuje statický obsah, vykresľovanie dynamického HTML, autentifikáciu užívateľov a v neposlednom rade interakciu s Javascript v prehliadači pomocou HTTP a JSON (niekedy nazývaný aj ako REST API). REST (*Representational State Transfer*) služby sa preto častokrát používajú pri tvorbe *single-page* aplikácií. Jedným z hlavných dôvodov tvorby je ich malá závislosť na klientovi. Všetko, čo je potrebné je HTTP klient, ktorý je bežnou súčasťou frameworkov a knižníc jazykov. To robí REST služby ideálne pre tvorbu aplikácií s veľkým i malým rozsahom a širokou platformnou podporou [22, s. 97].

Spring stále patril medzi populárne technológie pri implementácii *backend* a s príchodom *Spring Boot* spravil implementáciu ešte jednoduchšou. Java vývojárom poskytuje platformu pre vývoj aplikácií s minimálnou potrebnou konfiguráciou. Spojenie tejto platformy s Javascript frameworkom umožňuje vytvoriť *single-page* aplikáciu s možnosťou pokročilej interakcie a využitia služieb serveru[6] [13, 31]. Príklad architektúry klienta so serverom je zobrazený na obrázku č. 2.3.

Obr. 2.3: Príklad architektúry single-page aplikácie s *backend Spring Boot*.

Klient sa odkazuje na server pre získanie prístupu k zabezpečeným zdrojom. Server požiadavky filtruje, prípadne propaguje vo forme databázových dotazov. Databáza slúži na uloženie trvalých dát. Benefity, ktoré je možné získať využitím *Spring Boot* je mnoho. Prístupy ku chráneným zdrojom alebo do databázy sú pred klientom skryté a prístup k nim má len cez zabezpečené API. To poskytuje ďalšie možnosti monitorovania a auditu.

[6] Napríklad prístup do databázy, autentifikácia, autorizácia, monitorovanie a pod.

Kapitola 3

Vývoj single-page aplikácií

Znižovanie rozpočtu a nedostatočné plánovanie sa stáva bežnou súčasťou vývoja softvérových produktov [25]. Trhové podmienky a konkurenčné tlaky si vyžadujú od spoločností skrátenie doby vývoja, zníženie nákladov a zlepšenie kvality. V dôsledku toho vzniká negatívny vplyv na výkonnosť a efektívne riadenie vývoja softvéru, čo vedie k vývojárskym chybám a písaniu tzv. *legacy* kódu[1]. Riešenie problému spočíva v určení softvérového procesu, štandardov a pravidiel vývoja webových aplikácií [25].

Táto kapitola sa zaoberá vývojom *Single-page* aplikácií. Zameraná je predovšetkým na problémové miesta, ktoré majú významný dopad na kvalitu softvérového produktu a ich riešenie je častokrát náročné a výrazne spomaľuje začiatok vývoja. Na základe uvedených problémov je významná časť venovaná popisu vybraných nástrojov pre zjednodušenie vývoja, nasadzovania, testovania a správy aplikácií. Informácie uvedené v tejto kapitole sú dôležité pre návrh a rozšíriteľnosť výslednej aplikácie.

3.1 Problémové miesta vývoja

Softvérový priemysel čelí rýchlym zmenám v technológiách. Akonáhle sa systémy presunú do produkcie, často krát zlyhávajú alebo nie sú spôsobilé plniť účel, na ktorí boli navrhnuté. To všetko je zvyčajne spôsobené zlými implementačnými a prevádzkovými aspektami systém. V minulosti spoločnostiam stačila jednoduchá stránka so statickým obsahom a minimálnou logikou. Avšak dnes sa očakáva od webových aplikácií viac ako len web dizajn [25].

Definícia implementácie predstavuje nasadenie systému do činnosti. Štádium vývoja systémov, v ktorom je vyvinutý hardvér, softvér a nainštalovaný systém je otestovaný a zdokumentovaný [25]. Dôležité oblasti vývoja, ktoré môžu byť častokrát zdrojom problémov, je možné rozdeliť do niekoľkých oblastí:

- písanie kódu,
- testovanie,
- výkonnosť,
- zálohovanie,
- aktualizácie platformy,
- bezpečnosť.

[1] https://hackernoon.com/legacy-code-40027792bf85

Písanie kódu

Pokiaľ nie sú stanovené pravidlá písania kódu, môže byť úprava existujúceho kódu pre vývojára veľmi zložitá. Následok takéhoto vývoja je vznik *legacy* kódu. Je to kód, ktorý bol vytvorený niekým iným alebo ostal ako pozostatok starej aplikácie. Navyše je slabo zdokumentovaný a bez jasnej logiky. Odstránenie *legacy* kódu vylepší samotný produkt. Veľa ráz je tento proces spojený s úpravou (*refactor*) kódu [22, s. 189]. Efektívny spôsob prevencie pri výskyte tohto typu chýb je písanie testov[2] a využívanie praktík *Continuous integration*.

Testovanie

Jednou z najdôležitejších častí vývoja je testovanie. Pomáha overeniu funkčnosti aplikácie a taktiež slúži ako dokumentácia pre vývojárov. Čitateľnosť testov je rovnako dôležitá ako samotná čitateľnosť produkčného kódu. Existujú postupy, ktoré pomáhajú ovplyvniť čitateľnosť kódu a odstrániť tak problémy spojené s vývojom. Dôležité je dodržiavať zmysluplné pomenovanie testov a testovať v jednom súbore maximálne jednu entitu. Niekedy je však zložité rozhodnúť, čo je treba testovať a či je testov dostatok. Existuje viacero metód písania testov [22, s. 174].

Jedným z nich je *test-driven development*. Jeho princípom je písanie *unit* testov skôr než písanie kódu. Preto sa častokrát nazýva aj *test-first development* [22, s. 180-181]. Aplikuje sa v prípade dopredu neznámeho dizajnu produkčného kódu a jeho postup vyzerá takto:

- Napísať jeden nový test, ktorý poukazuje na smer riešenia.
- Pustiť test a overiť, že neprechádza.
- Napísať, čo najmenšie množstvo kódu do tela testu a zaistiť tak jeho prejdenie.
- Upraviť tento kód do metód, tried a vylepšiť tak jeho dizajn.
- Opakovať bod 1. s novým testom.

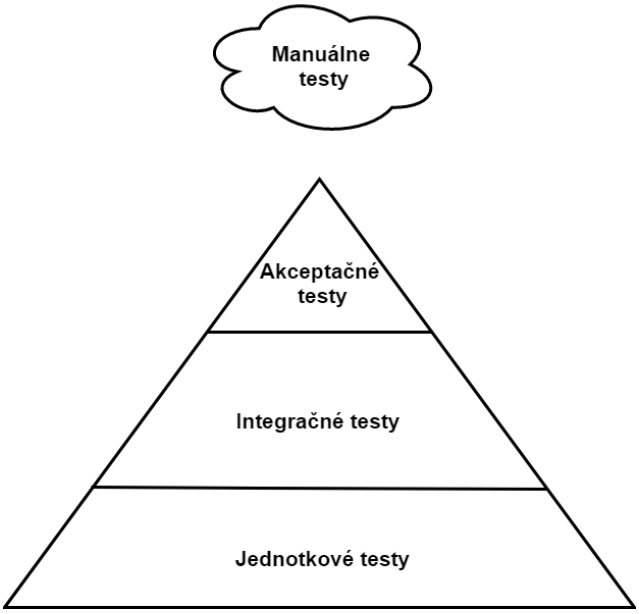
Tieto pravidlá zaisťujú vytvorenie minimálneho dizajnu na prejdenie *unit* testov a pomáhajú riešeniu problému tzv. *over-engineering*[3] [22, s. 179-181].

Unit testy nie sú jediný nástroj na overenie funkčnosti. Je zrejmé že softvér existuje na viacerých úrovniach a *unit* testy pracujú len s tou najnižšou. Ak by neboli použité iné nástroje, bol by validovaný len kód na najnižšej úrovni. Na obrázku 3.1 je zobrazená testovacia pyramída, ktorá zobrazuje relatívne množstvo testov, ktoré je použité na verifikáciu systému. Načrtnutie testovacej pyramídy systému môže byť užitočný nástroj pre detekciu problémov s flexibilitou kódu. Pokiaľ je jej tvar odlišný od pyramídy, je to znak veľkého množstva alebo nedostatku testov na určitej úrovni[4] [22, s. 182-184].

[2] Dôležité sú funkčné testy, ktoré simulujú užívateľskú interakciu v prehliadači.

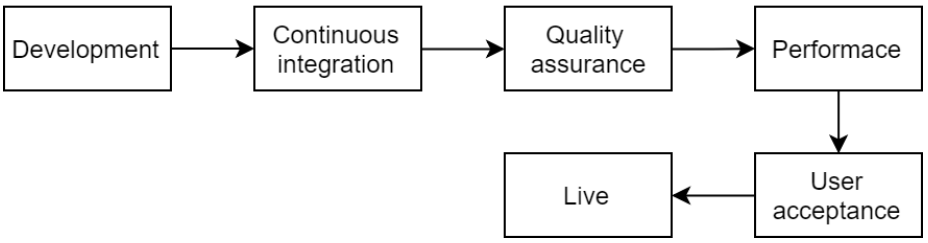
[3] https://hackernoon.com/how-to-accept-over-engineering-for-what-it-really-is-6fca9a919263

[4] Počet testov má tvar napr. *The hourglass* alebo *The snowcone*.



Obr. 3.1: Testovacia pyramída zobrazuje relatívne množstvo rôznych typov testov (prevzaté z [22, s. 182]).

Testovanie prebieha na viacerých úrovniach. Z pohľadu vývojára je dôležité najmä v prvotnej fáze overenie, že nová funkcionalita pracuje správne a že nedošlo k zhoršeniu existujúcej. Zabezpečenie dôvery v systém je však zložitejší proces. Obrázok 3.2 zobrazuje cestu od vývoja softvér až po nasadenie na produkciu. Každou fázou procesu sa zvyšuje robustnosť a dôvera v správnu funkčnosť systému [22, s. 185-186].



Obr. 3.2: Cesta softvéru od vývoja až po produkciu (prevzaté z [22, s. 186]).

V prvej fáze procesu je softvér vyvinutí. Využitím praktík *continuous integration* je otestovaný a pripravený na akceptáciu. V ďalšej časti môže dôjsť k optimalizácií a overeniu výkonnosti softvéru. Nedeliteľnou súčasťou je užívateľské testovanie, ktoré testuje dodatočné aspekty softvéru, dôležité pre užívateľa. V poslednej fázy procesu je dôvera v softvér na dostatočnej úrovni pre nasadenie do produkcie.

Výkonnosť

Dôležitá oblasť vývoja je zameraná na výkonnosť. Problémy s ňou spojené častokrát nastavajú u komplexnejších systémov s veľkým počtom užívateľov. Hoci ich dopad na funkčnosť systému nie je výrazný, majú negatívny vplyv na dostupnosť systému. Kritické parametre, ktoré treba pri vývoji sledovať sú:

- doba odozvy,
- súbežný počet užívateľov,
- vrchol zaťaženia systému,
- počet transakcií,
- počet užívateľov,
- kritické procesy [25].

Softvér, ktorý je navrhnutý s dôrazom na výkon je pre svojich užívateľov intuitívny a umožňuje im vykonávať bežnú prácu so zreteľným nárastom rýchlosti a efektivity [25]. Pri vývoji, je nevyhnutné klásť dôraz na rozšíriteľnosť a škálovateľnosť systému. Škálovaním je možné rozumieť vnútornú schopnosť systému zvládnuť rast, ktorý sa môže vyskytnúť napríklad zvýšením objemu vstupných dát alebo nárastom infraštruktúry. Aby bol systém považovaný za škálovateľný, rast by mal pokračovať neobmedzene. Veľkosť a objem spracovania sa môže rozširovať bez výrazného ovplyvnenia služby. Navyše, ak pracovné zaťaženie náhle prekročí kapacitu existujúcej kombinácie softvéru a hardvéru, systém upraví konfiguráciu, aby dosiahol zvýšené pracovné zaťaženie. Existujú dva hlavné spôsoby škálovania systémov:

- vertikálne,
- horizontálne.

Najjednoduchší spôsob ako škálovať softvér z pohľadu vývojára, je *vertikálne škálovanie*. Presúva softvér na počítač s vyššou kapacitou. Môže mať rýchlejší CPU, viac pamäte alebo rýchlejší pevný disk. Na druhú stranu *Horizontálne škálovanie* nie je tak jednoduché. Ak má softvér využívať výhody viacerých počítačov (alebo dokonca viacerých procesorov v rámci toho istého počítača), musí byť schopný paralelizovať svoje úlohy. Horizontálne škálovanie zvyšuje počet počítačov, na ktorých softvér beží. Zvyčajne sa jedná o počítače s rovnakou kapacitou ako súčasné [23]. Pri škalovaní je potrebné dávať pozor na:

- nadmerné spracovávanie,
- prezentáciu veľkého množstva údajov užívateľom,
- neefektívne UI,
- príliš veľa objektov v pamäti,
- žiaden alebo slabý caching dát [32].

Zálohovanie

Definuje požiadavky na odstránenie, zachovanie a obnovenie kľúčových údajov. Systémy občas čelia významnej výzve pri práci s veľkým množstvom existujúcich dokumentov a digitálnych údajov. Pri návrhu systému je potrebné rozhodnúť, ako dlho majú byť údaje uchovávané a aké sú požiadavky na prácu s nimi. Nesprávna konfigurácia môže mať negatívny dopad na výkonnosť výsledného systému [25].

Aktualizácia platformy

Technológia neustále napreduje a programátori musia držať krok. Frameworky, nástroje a knižnice sa pomerne rýchlo stávajú zastarané. Aktualizované verzie sú efektívnejšie, uľahčujú vývoj a umožňujú použiť najnovšiu funkcionalitu. Je, ale nutné si na nich rýchlo zvyknúť, čo môže byť pre nových vývojárov problém. Pri vývoji je preto dobré stanoviť si interval pre aktualizovanie platformy a snažiť sa pracovať s najnovšou funkcionalitou.

Bezpečnosť

Údaje uložené v systéme sú cenná komodita a užívatelia sa spoliehajú na systém, ktorý ich informácie chráni pred hrozbami. To je veľký tlak. Bohužiaľ, vývojári často prehliadajú bezpečnostné medzery vo svojom kóde a nevedia o dôsledkoch, až kým nedôjde k narušeniu bezpečnosti a strate údajov. Najmä ak sa sústredia na kód bez chýb a prehľadajú bezpečnostné medzery. Útočníci vedia túto slabosť využiť a hľadajú spôsoby, ako infiltrovať kód [25].

Vývojár nemôže zastaviť niekoho, kto by sa pokúsil kód narušiť, ale môže to pre neho urobiť ťažšie tým, že ho zabezpečí proti bežným metódam narušenia. Medzi najčastejšie bezpečnostné, aplikačné riziká patrí:

- *injection* chyby (SQL, NOSQL),
- nesprávna implementácia autentifikačných metód,
- nesprávna implementácia autorizačných metód,
- vystavenie citlivých dát,
- nesprávna bezpečnostná konfigurácia [15].

3.2 Nástroje zjednodušujúce vývoj

Vývoj kvalitného softvéru sa môže stať pre vývojára komplikovaným bez použitia nástrojov, ktoré pomáhajú vývojový proces zefektívniť a odstrániť spomenuté problémy. Je pravdou, že systémy sú často krát špecifické a určité problémy musí vývojár vyriešiť sám. V tejto časti sú popísané nástroje, ktoré zefektívňujú vývoj a pomáhajú tvoriť škálovateľné aplikácie. Uvedené informácie sú dôležité, pretože veľká časť týchto nástrojov je použitá pri návrhu aplikácie.

3.2.1 Automatizácia zostavenia aplikácie

Rozvojom vývoja moderného softvéru začali vznikať nové požiadavky na automatizáciu procesu zostavenia projektu. S nárastom agilných postupov musia zostavovacie nástroje podporovať včasnú integráciu kódu, ako aj časté a ľahké nasadenie skúšobných a produkčných prostredí.

Kompilácia väčších projektov je veľa ráz zložitá, eventuálne vyžaduje použitie viacerých nástrojov. Riešením problému je použitie jedného nástroja, ktorý zabezpečí zostavenie celej aplikácie, bez dodatočných príkazov a konfigurácií. V tejto časti sú popísané princípy dvoch významných nástrojov pre automatizáciu zostavenia aplikácie. Ďalšie možnosti ich použitia v oblasti automatizácie sú popísané v sekcií 3.3.

Maven

Maven poskytuje jednotný systém pre zostavenie aplikácie. Projekt sa vytvorí pomocou objektového modelu projektu (POM) a sade doplnkov, ktoré sú zdieľané vo všetkých projektoch pomocou systému *Maven*. Výhodou je úspora veľkého množstva času a podpora vo väčšine vývojových prostredí. Taktiež poskytuje veľa užitočných informácií o projekte, ktoré sú čiastočne prevzaté z POM a čiastočne generované zo zdrojového kódu projektu.

Okrem toho si kladie za cieľ zhromaždiť súčasné, najlepšie princípy vývoja a uľahčiť tak prácu vývojárom. Napríklad špecifikácia a vykonanie *unit* testov je súčasťou bežného procesu zostavenia projektu [17]. Medzi kľúčové vlastnosti patrí:

- jednoduché nastavenie projektu, ktoré sa riadi osvedčenými postupmi,
- možnosť konzistentne používať vo všetkých projektoch,
- vynikajúca správa závislostí vrátane automatickej aktualizácie,
- schopnosť ľahko pracovať s viacerými projektmi súčasne,
- okamžitý prístup k novým funkciám s malou alebo žiadnou ďalšou konfiguráciou,
- možnosť vytvoriť ľubovoľný počet projektov v preddefinovaných typoch výstupov, ako je JAR, WAR alebo distribúcia založená na metadátach o projekte, bez toho, aby bolo nutné vo väčšine prípadov písať skripty [17].

Gradle

Gradle je *open-source* nástroj, ktorý prichádza s podporou mnohých populárnych jazykov a technológií. Je ďalším vývojovým krokom v nástrojoch pre automatizáciu zostavenia projektu. Vychádza zo skúseností získaných zo zavedených nástrojov, ako je *Ant* a *Maven*, a posúva ich najlepšie myšlienky na ďalšiu úroveň. Vďaka prístupu založenému na klasifikácii, umožňuje *Gradle* deklaratívne modelovanie problémovej domény pomocou výkonného a expresívneho jazyka (DSL) implementovaného v *Groovy* (namiesto XML u Maven). Zameraný je prevažne na flexibilitu a výkonnosť. Vďaka jeho vysokej prispôsobiteľnosti je možná jeho rozšíriteľnosť a použitie v rôznych projektoch. Dôležitou vlastnosťou je možnosť dokončovať úlohy rýchlejšie vďaka prepoužívaniu predchádzajúcich prevedení, spracovávanie len zmenených vstupov a práca v paralelnom režime [7, 28].

Hoci je jednoduchšie pochopiť *Maven* zápis, špeciálne požiadavky na automatizáciu spôsobujú, vďaka jeho malej flexibilite, problémy. *Gradle* je vytvorený na základe dôvery v zodpovedného užívateľa, čo umožňuje použitie užívateľsky špecifických konfigurácií [8].

Zlepšenie rýchlosti zostavenia projektu, je jedna z najrýchlejších ciest, ako vyvíjať efektívnejšie. Z pohľadu výkonnosti sa *Gradle* odlišuje od ostatných nástrojov mechanizmom pre vyhýbanie sa práci. Medzi 3 kľúčové funkcionality patrí:

- Inkrementálne sprácovávanie - Sleduje vstup a výstup úloh a spracováva len potrebné.
- Cache - Prepoužíva výstupy iných *Gradle* zostavení s rovnakými vstupmi (aj medzi rôznymi strojmi).
- *Gradle Deamon* - Proces, ktorý si uchováva v pamäti informácie o zostavení [8].

Gradle s myšlienkou konvencie nad konfiguráciou podporuje tradičné riešenie správy závislostí. Medzi významné vlastnosti *Gradle*, ktoré ho robia častokrát prvou voľbou pre zaistenie automatizácie zostavenia projektov patrí:

- rozsiahle API,
- integrácia s ďalšími nástrojmi,
- rozsiahla komunitná podpora,
- jednoduchá rozšíriteľnosť,
- flexibilné konvencie,
- robustná a výkonná správa závislostí,
- škálovateľné zostavenie [28].

3.2.2 Statická analýza aplikácie

Kvalita a bezpečnosť softvéru sa zmenili na potrebu v modernom vývoji aplikácií. Kód pred nasadením do produkcie prechádza viacerými procesmi, medzi ktoré patrí aj statická analýza.

Analyzátory statického kódu hľadajú vzory, ktoré sú pre nich definované v podobe pravidiel a zapríčiňujú zraniteľnosť zabezpečenia alebo iné problémy spojené s kvalitou kódu. Analyzátory sa používajú vo vývoji už dlhšiu dobu a sú súčasťou najlepších praktík vývojového procesu. Najväčším prínosom statickej analýzy je možnosť nájsť chybu skôr, napríklad v štádiu vývoja, kedy oprava vyžaduje najmenej námahy. Avšak ich plnohodnotné využitie je najlepšie len vtedy, keď sú súčasťou procesu zostavenia projektu.

FindBugs

FindBugs je program, ktorý využíva statickú analýzu pre hľadanie chýb v Java kóde. Skenuje byte kód a hľadá vzory, ktoré naznačujú chybnú časť kódu. Hoci potrebuje skompilovaný kód, nevyžaduje jeho spustenie. Práca s týmto nástrojom pomáha predchádzať rozširovaniu problémov, ktorým sa dá vyhnúť. Je to vynikajúca motivácia pre zdokonaľovanie zručností vývojových tímov v písaní lepšieho kódu [5].

Každé zistenie nástroja sa uvádza ako upozornenie, ale nie všetky sú nevyhnutne chyby[5]. Vzory chýb sú rozdelené do niekoľkých kategórií, medzi ktoré patria napríklad:

- nesprávna prax,
- zraniteľnosť škodlivého kódu,
- výkonnosť,
- bezpečnosť,
- riskantný kód [5].

Checkstyle

Checkstyle je statický analyzátor, ktorý na rozdiel od *FindBugs* kontroluje dodržiavanie štandardov písania kódu. Sleduje mnoho aspektov zdrojového kódu, medzi ktoré patria problémy spojené s dizajnom tried alebo návrhom metód. Zameraný je taktiež na kontrolu štruktúry kódu a problémy s formátovaním. Využíva sa prevažne ako doplnok ďalších statických analyzátorov, ako je napríklad *FindBugs* [3].

[5]Napríklad varovanie týkajúce sa problémov s výkonom.

Eslint

Vývoj *single-page* aplikácií prináša nárast množstva Javascript kódu na *frontend*. Statická kontrola kódu sa preto využíva aj pri vývoji v Javascript. Jedným z najznámejších statických analyzátorov je nástroj *Eslint*. Používa sa na hľadanie problematických vzorov alebo kódu, ktorý nedodržiava stanovené pravidlá [4].

Javascript je dynamický, slabo typovaný jazyk, čím je obzvlášť náchylný na chyby vývojára. Ak neexistuje konvencia písania kódu, je to bariéra pre rozširovanie projektu a prácu v tíme. *Eslint* umožňuje vývojárom objaviť problémy v kóde bez toho, aby ho museli skompilovať a spustiť. Navyše ponúka vývojárom možnosť vytvárať vlastné pravidlá alebo použiť sadu preddefinovaných [4].

Okrem kontroly kódu, je to výborný nástroj na vyhľadávanie určitých tried chýb, ako napríklad tie, ktoré sa týkajú rozsahu platnosti premennej. Je zameraný na modulárnosť a transparentnosť pravidiel, ktoré môžu byť doplnené, zmenené, prípadne vypnuté [4].

3.2.3 Testovanie aplikácie

Testovanie kódu je súčasťou každého kvalitného softvéru. Zvyšuje efektivitu programátorov a kvalitu výsledného systému. Existuje viacero frameworkov pre testovanie rôznych aspektov systému. V tejto časti sú popísané 2 aktuálne používané platformy pre testovanie, ktoré vyžadujú minimálnu konfiguráciu a poskytujú vývojárom spätnú väzbu ku kvalite kódu. Ak sú vývojári vybavení nástrojmi pripravenými na použitie, skončia písaním väčšieho množstva testov, čo má za následok stabilnejšie a zdravšie základy projektu.

JUnit

JUnit je testovací framework, ktorý používajú vývojári pre implementáciu jednotkových testov v jazyku Java. Urýchľuje písanie testov, zvyšuje kvalitu kódu a jednoducho sa integruje do projektu. Medzi dôležité funkcionality patria:

- Fixtures,
- Test suites,
- Test runners,
- JUnit triedy.

Fixtures (inštalácie) určujú fixný stav, ktorý sa použije ako základ pre spracovanie testov. Hlavnou úlohou je zaistiť nemenné prostredie pre všetky testy. **Test suites** (testovacie sady) zlučujú testy do skupín a spúšťajú ich spolu. **Test runners** (spúšťače testov) sa používajú pre spracovanie jednotlivých testov. **JUnit triedy** sú dôležité triedy pre písanie testovacích scenárov. Patria tu napríklad triedy *Assert, TestCase* a *TestResult* [11].

- *Assert* obsahuje sadu *assert*[6] metód.
- *TestCase* definuje testovací prípad, pre vykonanie viacerých testov.
- *TestResult* obsahuje metódy, ktoré zbierajú výsledky po vykonávaní *TestCase*.

[6] http://tutorials.jenkov.com/java-unit-testing/asserts.html

Jest

Jest je testovací framework pre Javascript kód, ktorého zámerom je poskytnúť integrovateľné riešenie s minimálnou konfiguráciou. Podstatnou výhodou je možnosť paralelizácie testov s cieľom maximalizovať výkon. Vďaka separácií stavu jednotlivých testov, nedochádza ku konfliktom medzi testami [10].

Dôležitou súčasťou je aj jednoduché vytváranie prehľadov pokrytia kódu testami. *Jest* umožňuje zhromažďovať informácie o pokrytí kódu z celého projektu, vrátane netestovaných súborov. Silné stránky *Jest* sú rýchlosť, testovanie pomocou snímkov a voliteľnosť použitia rôznych jeho funkcionalít pri písaní testov [10].

3.2.4 Nasadenie a správa aplikácie

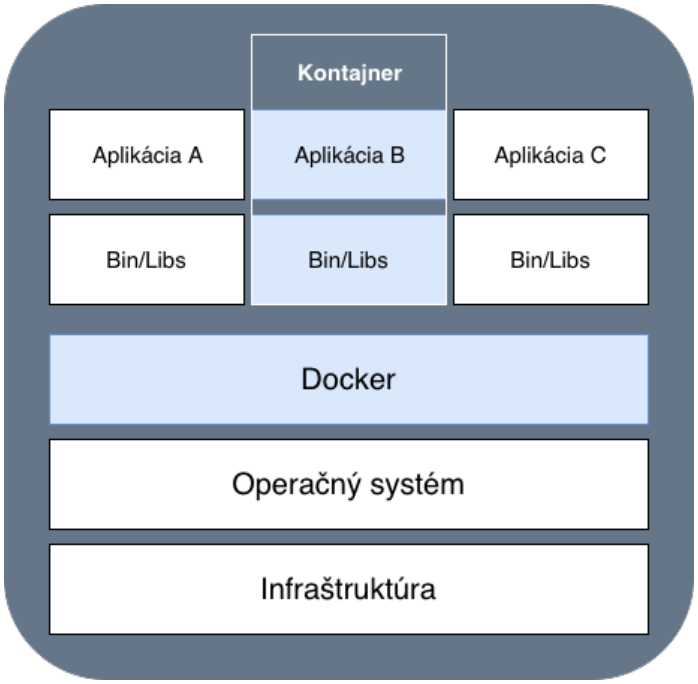
S príchodom nových vývojových nástrojov sa už vývojári nemusia zaoberať otázkou, na akom operačnom systéme bude aplikácia bežať. V tejto časti je popísaný aktuálne používaný nástroj pre nasadenie aplikácií do produkčného prostredia.

Docker

Docker je nástroj určený na zefektívnenie vytvárania, nasadzovania a spúšťania aplikácií pomocou kontajnerov, ktoré sú abstrakciou na aplikačnej vrstve. Kontajnery umožňujú vývojárom baliť aplikáciu so všetkými potrebnými časťami, ako sú napríklad knižnice a iné závislosti, a všetko to nasadiť ako jeden kontajner. Vďaka tomu sa môže vývojár uistiť, že aplikácia bude fungovať na ľubovoľnom inom počítači bez ohľadu na nastavenia stroja, ktoré by sa mohli líšiť od zariadenia používaného na písanie a testovanie kódu. Na jednom stroji môže bežať viacero kontajnerov a zdieľať tak rovnaké jadro operačného systému.

Docker využitím kontajnerov podporuje myšlienku vývoja mikro služieb a rieši tak problémy monolitickej architektúry aplikácií. Navyše kontajnery nemajú plnohodnotný operačný systém, čím zaberajú menej miesta v porovnaní s virtuálnymi strojmi. Na obrázku 3.3 je zobrazená architektúra *Docker*. Na spodných vrstvách je samotná infraštruktúra a operačný systém. *Docker* beží na hostujúcom operačnom systéme, kde využíva definované *namespace*, vďaka ktorým vie odlíšiť, ku ktorým zdrojom môže proces pristupovať. Kontajnery sú oddelené menné procesy, ktoré využívajú *Docker* k správe *namespace* a prístupu k hardvérovým zdrojom. Oproti virtuálnym strojom dochádza k izolácií procesov na vyššej úrovni, čím sa réžia zmenšuje. *Docker* využíva viacero typov *namespaces*, medzi ktoré patrí:

- *pid namespace* (izolácia na úrovni procesov),
- *net namespace* (správa sieťových rozhraní),
- *mnt namespace* (správa súborového systému).



Obr. 3.3: Architektúra *Docker*.

Docker je nástroj, ktorý je určený pre vývojárov i správcov systémov. Pre vývojárov to znamená, že sa môžu sústrediť na písanie kódu bez obáv o systém, na ktorom bude aplikácia bežať. Navyše je možné použiť hotové programy, ktoré sú navrhnuté na spustenie v kontajneri ako súčasť aplikácie. Pre prevádzkový personál prináša flexibilitu a vďaka malej réžií, znižuje počet potrebných systémov [26].

3.3 Continuous integration

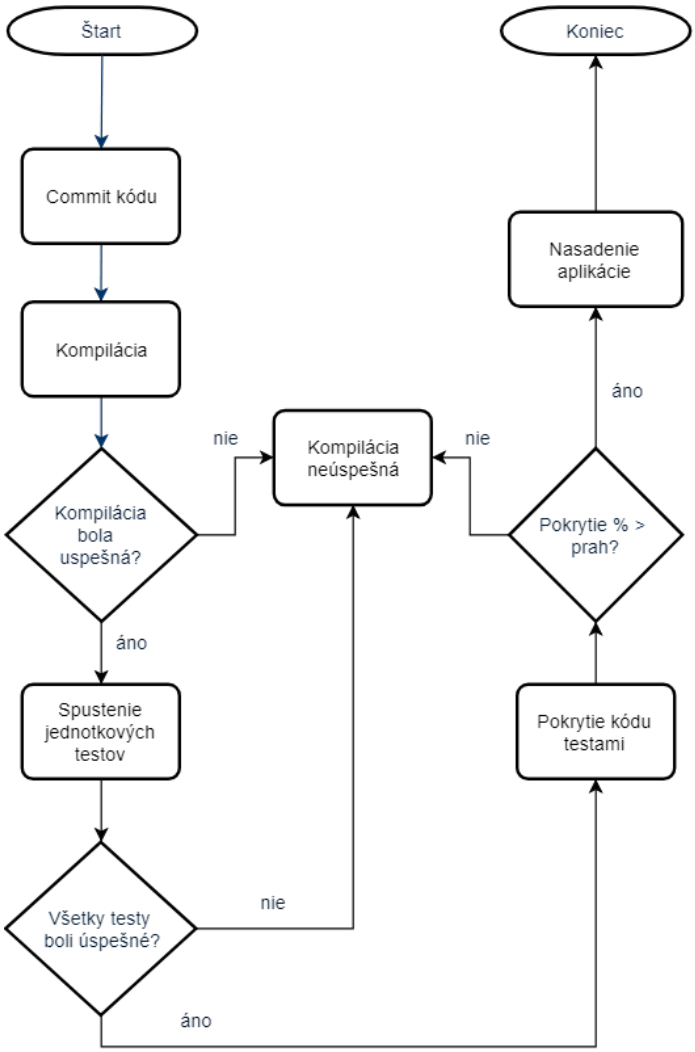
Popísané nástroje a postupy sú súčasťou vývojových štandardov, ktoré vyžadujú od vývojárov dodržiavanie stanovených pravidiel vývoja. Kontrola však nemusí prebiehať len lokálne. Kedykoľvek vývojár odovzdá kód do centrálneho úložiska, dôjde k spusteniu procesu kontroly a zostavenia aplikácie. Tento proces sa nazýva *Continuous integration* (CI). V posledných rokoch sa tento proces zaradil medzi najlepšie praktiky vývoja a je založený na princípoch, medzi ktoré patrí:

- automatické zostavenie projektu,
- automatické testovanie,
- kontrola kódu a dodržovanie pravidiel vývoja.

Každá integrácia je validovaná automatickým zostavením, ktoré zahŕňa testy a detekuje integračné chyby. Má významný vplyv na redukciu integračných problémov a pomáha vyvíjať rýchlejšie [22, s. 403-404].

Pravidelnou integráciou je možné detekovať a nájsť chyby skôr. Hoci, zostavenie aplikácie je často dostačujúci faktor kontroly, že vývojár nič nepokazil, CI server poskytuje napríklad možnosť spúšťať jednotkové testy a overiť tak, že je pokrytý dostatok kódu. Následne sa môže pokúsiť vytvoriť balík pre nasadenie. Všetky tieto kroky bežia v sérií, pričom úspech každého je podmienkou nasledujúceho kroku.

Namiesto strávenia veľkého množstva času overovaním kódu, môže vývojár len skompilovať kód, spustiť testy a zvyšok nechať na CI server. Na obrázku 3.4 je zobrazený zjednodušený *workflow* diagram CI služby.



Obr. 3.4: Zjednodušený *workflow* diagram pre CI službu (prevzaté z [22, s. 404]).

Ak sa neintegruje kód pravidelne, je exponenciálne ťažšie nájsť a odstrániť problémy. Takéto problémy s integráciou môžu ľahko narušiť projekt alebo spôsobiť jeho úplné zlyhanie. CI sa nezbavuje chýb, ale robí ich dramaticky ľahšie nájsť a odstrániť [22, s. 403-404].

Kapitola 4

Návrh aplikácie

V predchádzajúcej kapitole boli popísané problémy spojené s vývojom a správou webových aplikácií. Podstatná časť bola venovaná popisu nástrojov pre zefektívnenie a odstránenie najčastejších úskalí vývoja. Na základe týchto znalostí je v tejto kapitole popísaný návrh aplikácie, ktorá je schopná poskytnúť vývojárovi štandardizovaný základ pre vývoj *single-page* aplikácií a odstrániť tak problémy spojené so začiatkom vývoja. Navyše bude schopná automatizovane nakonfigurovať projekt pre možnosti automatického zostavenia, testovania a správy. Hlavná požiadavka, ktorá sa kladie na návrh spočíva v možnosti jednoduchej aktualizácie nástrojov a konfigurácií aplikácie. Z dôvodu, že projekty sú rôznorodé, je nevyhnutné umožniť vývojárovi rozširovať aplikáciu o vlastné funkcionality a nastavenia.

Hoci realizácia návrhu má viacero možných riešení, za najvhodnejšie bola považovaná implementácia Javascript knižnice. Jej najväčší prínos spočíva v možnosti implementácie interaktívneho a verzovateľného nástroja, ktorý je ľahko rozšíriteľný.

Samotná aplikácia (*Create Sbspa*) bude fungovať ako generátor *single-page* aplikácií, ktorého vstup bude získaný pomocou užívateľského rozhrania. Užívateľ vyberie časti aplikácie, ktoré sa majú vygenerovať. Výstupom generátora bude vygenerovaná kostra projektu, ktorá obsahuje zvolené časti *single-page* aplikácie. Jednotilivé časti bude možné modifikovať, prípadne celé pregenerovať.

Veľká časť nástrojov, konfigurácií a kódu pre vytvorenie single-page aplikácie bude v generátore reprezentovaná pomocou sémantických šablón. Táto reprezentácia umožní generovať súbory v závislosti na definovaných parametroch užívateľa. Aby bolo možné generátor voľne používať a negeneroval užívateľovi zbytočný kód, musí taktiež poskytovať možnosť výberu sémantických šablón podľa potreby.

Kapitola je rozdelená do dvoch častí. V prvej časti je uvedený návrh generátora (*Create Sbspa*), ktorý obsahuje popis navrhnutých funkcií a očakávaných výstupov po dokončení procesu generovania. Druhá časť je venovaná návrhu štruktúry *single-page* aplikácie. Zameraná je na popis nástrojov a konfigurácií, ktoré priamo zefektívňujú vývoj a patria medzi najlepšie praktiky pri tvorbe webových aplikácií.

4.1 Generátor štruktúry projektu

Hlavnou úlohou generátora je **vygenerovanie kostry projektu**. Do tejto skupiny patria všetky konfigurácie nástrojov pre zjednodušenie vývoja. Štruktúra *backend*, *frontend* a nakonfigurované automatizované zostavenie projektu. Návrh funkcionalít generátora je rozde-

lený do viacerých častí, z dôvodu lepšej separovateľnosti funkcionalít a možnosti výberu len potrebných častí kódu. Generátor bude môcť vygenerovať:

- *backend*,
- *frontend*,
- *GraphQL* API,
- *Docker* konfigurácie.

Namiesto špecifikovania všetkých častí, ktoré sa majú vygenerovať, môže vývojár vygenerovať celý projekt.

Generovanie *backend*

Pri návrhu komplexnejších aplikácií je často potrebný *backend*, ktorý sa stará o činnosti spojené s autentifikáciou alebo prístupom do databázy. Z toho dôvodu, môže vývojár vyžadovať jeho vygenerovanie. Pre vývoj *single-page* aplikácií sa za najlepší javí framework *Spring Boot*, vďaka podpore veľkého množstva funkcionalít a jednoduchému použitiu. Medzi užívateľské parametre, ktoré sú nevyhnutné pre realizáciu generovania *backend*, patrí:

- názov zložky projektu,
- *groupId,*
- *artifactId.*

Na základe vstupných parametrov bude vygenerovaná štruktúra *Spring Boot* aplikácie s konfiguráciou nástrojov *FindBugs*, *Checkstyle* a ďalších. Cesty k Java kódu sa nahradia podľa *groupId* a *artifactId* a celý projekt sa vygeneruje do zložky podľa vstupného názvu.

Generovanie *frontend*

Podstatnou časťou generátora bude možnosť vygenerovať *frontend single-page* aplikácie. Z dôvodu lepšej rozšíriteľnosti, bude použitý framework *React*. Medzi užívateľské parametre, ktoré sú nevyhnutné pre realizáciu generovania *frontend*, patrí:

- názov zložky projektu,
- názov aplikácie.

Na základe vstupných parametrov bude vygenerovaná štruktúra *React* aplikácie s konfiguráciou nástrojov *Tslint*, *Jest* a ďalších. Aplikácia bude rovnako ako *backend* vygenerovaná do zložky podľa vstupného názvu.

Generovanie *GraphQL*

V posledných rokoch sa REST stáva obvyklou súčasťou pri návrhu webových API. Systémy sú však málo pružné na rýchlo meniace sa požiadavky klientov, ktorí k nim pristupujú. *GraphQL* bol vyvinutý na to, aby zvládol potrebu väčšej flexibility a efektívnosti. Rieši mnohé z nedostatkov, s ktorými sa vývojári stretávajú pri interakcii s nástrojmi REST API [9]. V prípade ak sa vývojár rozhodne implementovať *backend* API s využitím *GraphQL*, generátor pridá závislosti do *Spring Boot* aplikácie a vytvorí vzorovú schému pre tvorbu API rozhraní. Generátor bude závislý na vygenerovaní *backend* a na vstupe bude vyžadovať:

- názov zložky projektu.

Generovanie *Docker konfigurácie*

Dôležitou súčasťou riešenia problému rozšíriteľnosti, je možnosť spustiť aplikáciu nezávisle na prostredí. Podstatnou časťou bude preto vytvorenie *Docker image*, ktorý obsahuje všetky potrebné inštrukcie pre zostavenie spustiteľnej aplikácie. Takto vytvorená aplikácia bude následne bežať v *Docker* kontajnery. Medzi nevyhnutné parametre generátora patrí:

- názov zložky projektu.

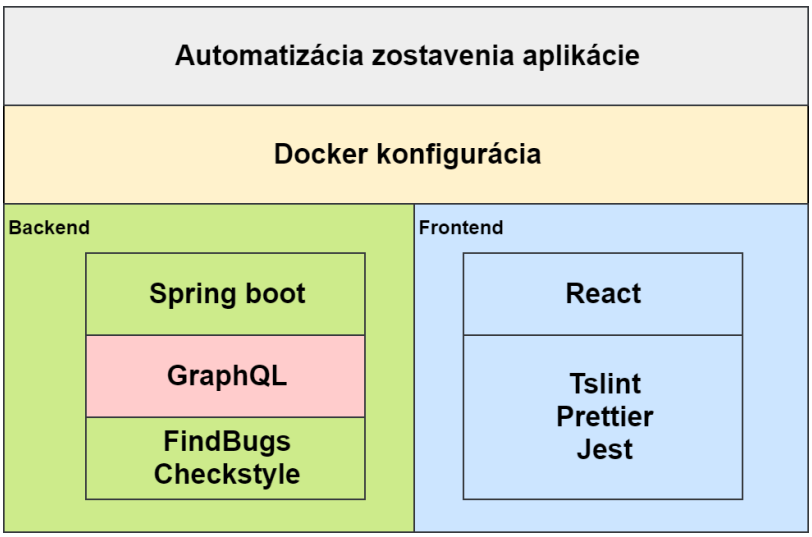
Výstupom generátoru bude konfiguračný súbor pre vytvorenie *Docker image*.

Aktualizácie

Nástroje a závislosti projektu je potrebné udržovať aktuálne. Vďaka návrhu verzovateľnej Javascript knižnice, bude vývojár schopný aktualizovať len časti projektu, ktoré potrebuje. Navyše voľnosť výberu verzie generátora, zmenšuje riziko vzniku problémov spojených s úpravou sémantických šablón. Všetky vyššie spomenuté generátory, budú môcť okrem generovania nových častí, nahradiť aj existujúce. Využitím ľubovoľného nástroja pre správu verzií, bude možné prípadne zmeny schváliť alebo odmietnuť.

4.2 Štruktúra single-page aplikácie

V tejto časti je uvedená štruktúra projektu, ktorá bude použitá na vytvorenie sémantických šablón generátora. Zameraná je prevažne na popis použitých nástrojov a konfigurácií v rôznych častiach aplikácie. Taktiež sú v nej uvedené prínosy, ktoré zefektívňujú vývoj a pomáhajú riešiť problémy spojené s jeho začiatkom. Na obrázku 4.1 je zobrazená štruktúra *single-page* aplikácie pri použití všetkých častí generátora štruktúry projektu.



Obr. 4.1: Vygenerovaná štruktúra *single-page* aplikácie.

Vyznačené časti predstavujú jednotlivé súčasti generátora, ktoré môže vývojár vygenerovať. **Automatizácia zostavenia aplikácie** je predvolenou súčasťou generátora. Využíva nástroj *Gradle*, medzi ktorého činnosti patrí zostavenie, testovanie a vytvorenie produkčnej aplikácie. Zameraný bude prevažne na poskytnutie jednotného rozhrania pre správu

celého projektu, nezávisle na jazyku. Voliteľnou časťou bude **Docker konfigurácia**, ktorá umožňuje vytvárať verzovateľné *Docker images* zo *single-page* aplikácie. Na obrázku je zelenou farbou označený **backend** s voliteľnou možnosťou generovať podporu pre *GraphQL* a s nástrojmi pre detekciu chýb a nesprávnych praktík písania kódu. Poslednou súčasťou štruktúry je **frontend**, ktorý bude obsahovať konfiguráciu spustiteľnej *React* aplikácie a nástroje pre jej efektívnejší vývoj. V nasledujúcich častiach je uvedený podrobnejší popis *backend* a *frontend single-page* aplikácie.

4.2.1 Popis frontendu

Štruktúra *frontend* časti *single-page* aplikácie zahrňuje vytvorenie konfiguračných súborov pre zostavenie a správu aplikácie. Pre lepšiu kontrolu nad typmi bude použitý Typescript s knižnicou *React*, ktorý je zostavením kompilovaný do jednoduchého Javascript. Statická kontrola pomáha zlepšiť čitateľnosť kódu a zmenšiť riziko použitia nesprávneho typu.

Zostavenie aplikácie

Zostavenie aplikácie vyžaduje vytvorenie optimalizovaného kódu, ktorý nebude náročný na klienta a umožní vývojárovi ladenie aplikácie. Za najlepší z pohľadu možnosti konfigurácie a rozšíriteľnosti sa javí nástroj *Webpack*[1]. Nástroj poskytuje vytvorenie 2 typov zostavenia aplikácie:

- vývojový,
- produkčný.

Vývojové zostavenie projektu neobsahuje optimalizácie kódu, ktoré sú potrebné u produkčného. Namiesto toho umožňuje ladenie aplikácie v prehliadači a tzv. *Hot Module Replacement* (HMR)[2]. HMR vymieňa, pridáva alebo odstraňuje moduly počas bežiacej aplikácie bez potreby úplného znovu načítania. Zachováva stav aplikácie, ktorý by sa stratil počas úplného načítania a šetrí cenný čas vývoja, aktualizovaním len toho, čo sa zmenilo. Navyše úpravy vykonané v zdrojovom kóde majú za následok okamžitú aktualizáciu prehliadača, ktorá je porovnateľná so zmenou štýlov priamo v jeho nástrojoch (*devtools*).

Produkčné zostavenie projektu sa od vývojového líši vo viacerých aspektoch. Hoci neumožňuje vývojárovi ladiť aplikáciu a HMR, výstupom je optimalizovaný produkčný kód. Medzi podstatné časti patrí minifikácia kódu a podpora cachovania v rámci prehliadača. To umožňuje vytvorenie progresívnej webovej aplikácie s offline podporou a odolnosťou na problémy siete. Výhodou toho je možnosť používať *single-page* aplikáciu, aj po výpadku siete, v obmedzenom režime.

Navyše je výsledný kód rozdelený do tzv. *chunks*. Môžu byť cachované[3] a asynchrónne načítané. To je obzvlášť prínosné, ak je aplikácia rozsiahla. Užívateľovi je načítaný len potrebný kód, vďaka čomu sa zrýchľuje prvotné načítanie aplikácie. Optimalizované zostavenie projektu je dôležité pre zníženie počtu výkonnostných problémov a zlepšenie užívateľského zážitku.

[1] https://webpack.js.org/
[2] https://webpack.js.org/concepts/hot-module-replacement/
[3] Napríklad v prehliadači a na servery.

Testovanie aplikácie

Podstatným problém, ktorým sa návrh zaoberá je testovanie aplikácie. Existuje viacero frameworkov, ktoré zabezpečujú komplexnú platformu pre testovanie Javascript kódu. V návrhu je použitý framework *Jest*, ktorý je často používaný na testovanie *React* kódu. Konfigurácia tohto nástroja definuje podmienky pokrytia kódu testami. Špecifikuje koľko percent riadkov, funkcií a podmienok musí byť v Javascript otestovaných. Hoci sa konfigurácia dá jednoducho modifikovať, poskytuje základ pre nastavenie pravidiel testovania kódu. Navyše splnenie testovacích podmienok sa testuje, pri každom *push* do centrálneho úložiska. Keďže sú projekty rôznorodé, vývojár nie je nútený generátorom použiť vygenerované testovacie nastavenia. Môže použiť iný framework, prípadne nastavenia modifikovať.

Kontrola kódu

Vývoj komplexnejších *single-page* aplikácií vytvára v tímoch problémy spojené s písaním kódu. Ak aj sú definované pravidlá, nie vždy sú dodržiavané. Bez toho, aby si museli vývojári všetko pamätať, je výhodné použiť statický analyzátor. Pri návrhu sa za najlepšiu javí spolupráca knižnice *Tslint* a *Prettier*. Pravidlá písania kódu budú definované v konfiguračnom súbore pre nástroj *Tslint*. Ten sa stará o kontrolu zdrojových súborov a upozorňuje užívateľa na ich výskyt. Nástroj *Prettier* funguje ako voliteľný formátovací nástroj, ktorý zabezpečí rovnaké formátovanie kódu naprieč celým projektom. Oba tieto nástroje budú použité pri každom *commit* do centrálneho úložiska. Vývojár má prístup k obom konfiguračným súborom, pre jednoduchú modifikáciu pravidiel v závislosti na projekte.

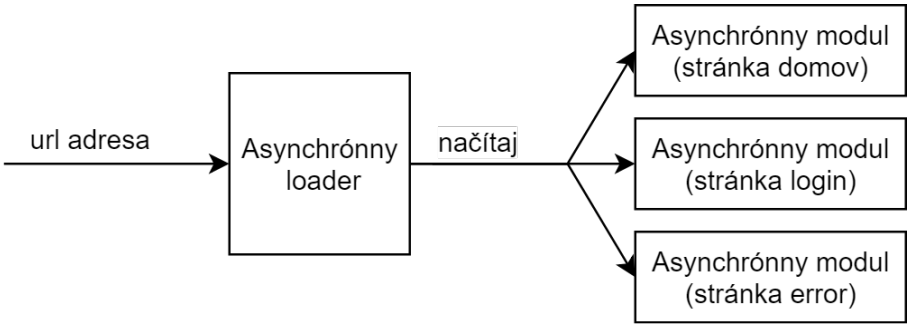
Spustenie aplikácie

Zostavením aplikácie vznikne statický obsah, ktorý je poskytovaný užívateľovi. Pre vývojový výstup je pri návrhu použitý *Webpack Dev Server*, ktorý zabezpečí popísaný HMR a umožní efektívne ladiť *single-page* aplikáciu. Produkčný výstup je poskytovaný pomocou jednoduchého servera statického obsahu, ktorý obsahuje konfiguračné súbory pre nastavenie hlavičiek[4], prípadne zachytávanie dotazov a odpovedí. Všetky konfiguračné súbory budú dostupné vývojárovi, ktorý ich môže modifikovať.

Asynchrónne načítanie modulov

Zaistenie vytvorenia asynchrónnych *chunks*, v závislosti na moduloch, vyžaduje navrhnúť spôsob, ktorým ich definícia bude prehľadná a jednoducho rozšíriteľná. Na obrázku 4.2 je zobrazený príklad asynchrónneho načítania modulov, ktorý je použitý pri návrhu aplikácie.

[4] Napríklad hlavička *Cache-Control*.



Obr. 4.2: Príklad asynchrónneho načítania modulov.

Aplikácia je rozdelená podľa stránok do skupín. Každá z nich obsahuje moduly, z ktorých je zložená. Podľa URL adresy, asynchrónny *loader* vyberie moduly, ktoré je potrebné načítať. Takto vybrané moduly sú potom asynchrónne načítané v prehliadači. Ak sa vývojár rozhodne rozdeliť aplikáciu, podľa inej logickej štruktúry, je možné prepoužiť definíciu asynchrónnych modulov a upraviť len asynchrónny *loader* bez závislostí na URL adrese.

Hlavnou výhodou je možnosť načítať komponenty až pri ich vykreslení. Hoci sa zmenšuje veľkosť zbytočného kódu, môže to mať, pri nesprávnom použití, negatívny dopad na rýchlosť užívateľského rozhrania. Asynchrónne moduly sú výhodné pre veľké celky kódu[5], ktorých načítanie pri prvotnom príchode má negatívny dopad na výkonnosť aplikácie.

4.2.2 Popis backendu

Štruktúra *backend* časti *single-page* aplikácie je vďaka presunu väčšej časti logiky na klienta výrazne jednoduchšia. Bude obsahovať konfiguračné súbory pre zostavenie a správu *Spring Boot* aplikácie. Navyše poskytne možnosť vytvoriť API pomocou *GraphQL*.

Zostavenie aplikácie

Pre zostavenie aplikácie bude možné použiť nástroj *Gradle*, ktorý je schopný spustiť a otestovať výslednú aplikáciu. Konfigurácia nástroja *Gradle* zahŕňa zoznam *backend* závislostí a definíciu úloh, ktoré sa majú vykonať. Výstup kompilácie je spustiteľný `jar` súbor.

Testovanie aplikácie

Predvolenou závislosťou **backend** aplikácie bude *jUnit* knižnica, ktorá umožňuje písanie jednotkových testov pre Java triedy. Pri návrhu bola zvolená z dôvodu jednoduchosti a možnosti použitia pre testovanie *GraphQL* API. Navyše testy budú spustené pri každom *push* do centrálneho úložiska, čo umožňuje jednoduchšiu kontrolu kódu a rýchlejšiu detekciu chýb. Konfiguráciu bude možné po vygenerovaní modifikovať v závislosti na projekte.

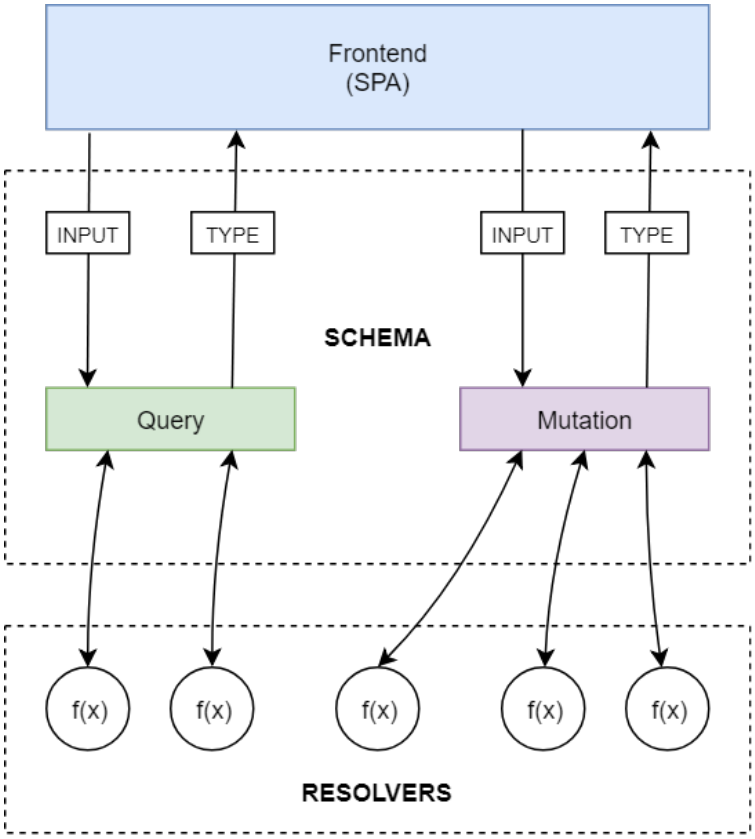
Kontrola kódu

Lepšia kontrola kódu bude zabezpečená využitím nástrojov *FindBugs* a *Checkstyle*. Konfiguračné súbory obsahujú definíciu pravidiel písania kódu v projekte. Ich dodržiavanie bude kontrolované pri každom *commit* do centrálneho úložiska. Vývojár bude môcť jednoducho doplniť ďalšie pravidlá, prípadne modifikovať predvolené.

[5] Napríklad stránky.

Konfigurácia GraphQL

Návrh použitia *GraphQL* knižnice, pomáha riešiť výkonnostné problémy aplikácie a prispieva k zlepšeniu čitateľnosti projektu. Vďaka jej efektívnemu dotazovaciemu jazyku, je možné výrazne zredukovať počet dotazov na server.



Obr. 4.3: Štruktúra *GraphQL* API (prevzaté z [24]).

Návrh štruktúry projektu zahrňuje vzorovú *GraphQl* definíciu schémy a pomocné triedy pre jej obsluhu. Na obrázku 4.3 je zobrazená štruktúra *GraphQL* API. Skladá sa z niekoľkých častí:

- *Inputs* reprezentujú dáta, ktoré na server zasielajú klienti. Patria sem napríklad dáta z *frontendu* aplikácie.
- *Types* definujú entity, ktoré na *GraphQL* servery existujú. Okrem typu objekt, to môžu byť aj základné skalárne typy ako je *Integer*, *String* alebo *Boolean*.
- *Queries* poskytujú klientovi jednotné rozhranie pre získavanie dát zo serveru. Hoci definujú štruktúru dát, ktorú je možné získať, nepopisujú spôsob, ako sa k nim samotný server dostane.
- *Mutations* umožňujú klientom upravovať dáta na serveri. Patrí tu napríklad mapovanie na DAOs (*data access objects*). Podobne ako pri *queries*, ani *mutations* nič nehovoria o tom, akým spôsobom server tieto dáta upraví.

- *Resolvers* definujú, akou metódou server získava alebo upravuje dáta. Väčšinou bývajú volané DAO funkcie pre prístup do databázy [24].
- *Schema* je spojenie *queries*, *mutations* a *resolvers*. Reprezentuje rozhranie servera, ktoré poskytuje klientovi.

Backend obsahuje konfiguráciu *GraphQL* API a vzorový API endpoint pre znázornenie závislostí jednotlivých častí a návrhu separácie API logiky.

Kapitola 5

Implementácia

Prechádzajúca kapitola sa venovala návrhu generátora, prostredníctvom ktorého je možné vygenerovať časti *single-page* aplikácie. Navyše poskytuje nástroje, ktoré majú pozitívny dopad na udržateľnosť, vývoj, testovanie, nasadzovanie a správu aplikácií.

Na základe návrhu je možné pristúpiť k samotnej implementácií nástroja, ktorý bude generovať súčasti projektu využitím sémantických šablón. Táto kapitola je rozdelená do troch častí. V prvej časti je uvedený implementačný popis generátora. Venovaná je predovšetkým popisu hlavných implementačných detailov nástroja, charakteristike jeho použitia a využitých technológií. Druhá časť je zameraná na popis vygenerovanej štruktúry aplikácie, jej súčastí a konfiguračných nastavení. V poslednej časti sú popísané ukážkové materiály, ktoré boli vytvorené za účelom ich možného použitia pri vyuke v predmete WAP.

5.1 Generátor aplikácií

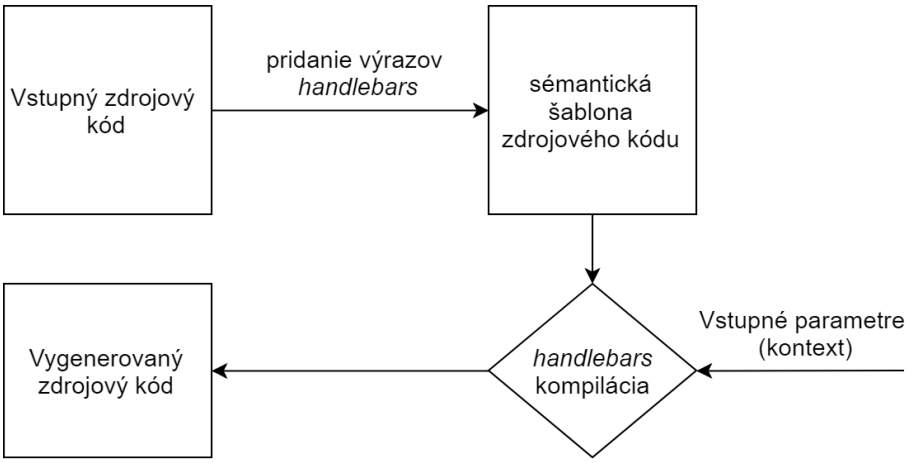
Veľká časť implementácie bola zameraná na realizáciu nástroja, ktorý zefektívni samotný vývoj *single-page* aplikácií a poskytne nástroje na ich jednoduchšiu správu. Výsledkom toho je nástroj koncipovaný v podobe konzolovej aplikácie, ktorá je implementovaná v jazyku Javascript. Medzi externé knižnice a nástroje, ktoré sú nevyhnuté na implementáciu a beh aplikácie, patrí knižnica *plop*[1]. Je použitá na realizáciu interaktívneho užívateľského rozhrania. Navyše poskytuje nástroje pre interpretáciu a spracovanie sémantických šablón využitím knižnice *Handlebars*[2].

5.1.1 Generovanie kódu pomocou sémantických šablón

Pred samotným popisom toho, čo sa bude generovať, je dôležité zmieniť elementárne časti generátora, ktoré sú na sebe závislé. Pre spracovávanie a tvorbu sémantických šablón je použitý nástroj *Handlebars*. Hoci veľká časť kódu a nastavení je generovaná rovnako, u niektorých konfigurácií a súborov je dôležité rozlíšiť špecifické nastavenia projektu. Medzi ne môže patriť názov projektu, triedy, verzia alebo iné. Pretože kód a konfigurácia je vložená do šablón v nezmenenom formáte, sú výrazy *Handlebars* izolované od zbytku kódu pomocou dvojitých zátvoriek **{{**. Výraz je potom vložený medzi zátvorky a vyzerá takto - **{{výraz}}**. Podobne ako v iných jazykoch je možné využívať **for** cykly alebo podmienky **if**. Navyše je vývojár schopný definovať vlastné funkcie, ktoré sú dostupné v rámci akéhokoľvek miesta v šablóne. Vytvorené šablóny je možné skompilovať pomocou kompilátora

[1] https://www.npmjs.com/package/plop
[2] https://handlebarsjs.com/

Handlebars. Na obrázku 5.1 je zobrazený proces generovania kódu pomocou sémantických šablón.

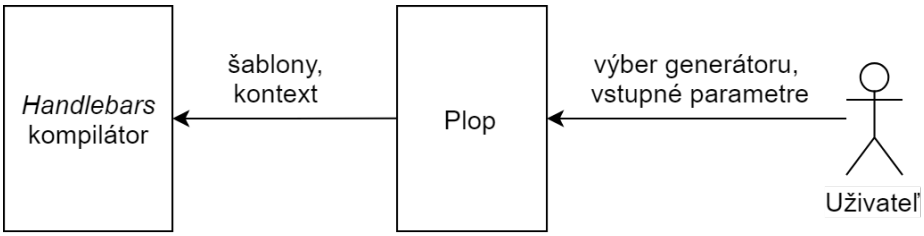


Obr. 5.1: Proces generovania kódu pomocou sémantických šablón a knižnice *Handlebars*.

Kód alebo konfigurácia, ktorá sa bude generovať, sa upraví do formátu šablóny. Pridajú sa výrazy a funkcie, ktoré generujú výstup v závislosti na vstupných parametroch. Kompilátor vyhodnotí výrazy a vygeneruje výsledný kód. Tento proces umožňuje tvorbu variabilných šablón, nezávisle na zdrojovom kóde a závisle na vstupných parametroch.

5.1.2 Tvora užívateľského rozhrania a správa šablón

Aby bol generátor efektívny a vývojárovi negeneroval zbytočný kód, je potrebné implemetovať rozhranie, ktoré zabezpečí správu a poskytne vstupné parametre kompilátoru sémantických šablón. V tejto časti je uvedený popis tvorby kontextu pomocou knižnice *plop*, ktorá je použitá z dôvodu jednoduchej možnosti prepojenia užívateľského vstupu a knižnice *Handlebars*. Na obrázku 5.2 je názorne uvedené napojenie *plop* na kompilátor sémantických šablón.



Obr. 5.2: Napojenie *plop* na kompilátor sémantických šablón.

Plop funguje ako mikro generátorový framework, ktorého výsledkom je konzolová aplikácia s prehľadným menu. Všetky sémantické šablóny sa rozdelia do skupín podľa významu. Každá skupina môže obsahovať ďalšie podskupiny alebo zoznam sémantických šablón, ktoré má vygenerovať. Generátor získava parametre kontextu pomocou doplňujúcich dotazov a na základe nich vyberá šablóny. Navyše vývojárovi umožňuje prepísať vygenerovaný kód. To je obzvlášť výhodné pri aktualizácií knižníc alebo konfigurácií generátora. Aby sa zamedzilo prepísaniu kódu bez vedomia vývojára, generátor požaduje pri každej takejto zmene súhlas.

Práca s knižnicou prebieha pomocou objektového API, v ktorom sú nadefinované rôzne typy generátorov. Každý z nich obsahuje *prompts* (užívateľské dotazy) a *actions* (akcie). Užívateľské dotazy slúžia pre získanie vstupných parametrov alebo špecifických vlastností, ktoré sa použijú pri generovaní šablón. Akcie definujú výstupný súbor a šablóny, ktoré sa majú použiť pre generovanie, prípadne prepísanie kódu.

5.1.3 Popis generovania

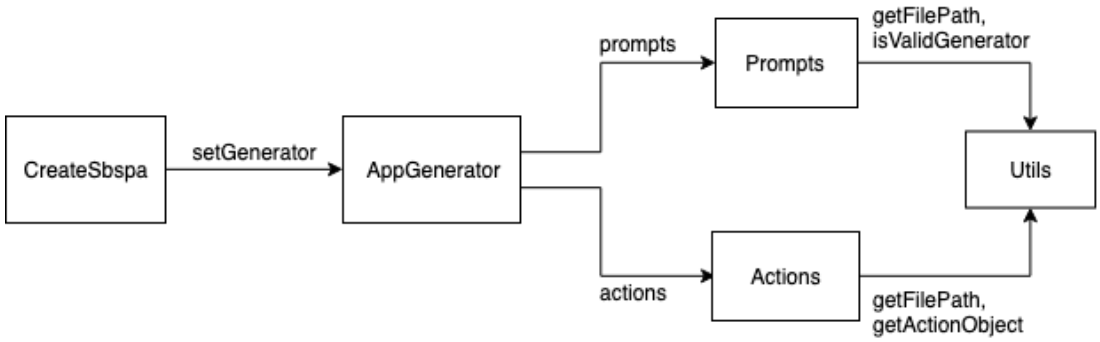
Aplikácia s užívateľom komunikuje pomocou interaktívneho menu, prostredníctvom ktorého je možné vybrať časti projektu, ktoré sa majú vygenerovať. Tieto časti je navyše generátor schopný modifikovať, pripadne celé pregenerovať. Nástroj poskytuje niekoľko možností generovania:

- generovať všetko,
- *backend*,
- *frontend*,
- *GraphQL* API,
- *Docker* konfigurácie.

Na základe týchto možností je stanovená dekompozícia aplikácie do funkčne špecifických modulov, ktorá umožní efektívnejšiu správu aplikácie s nízkym počtom závislostí. Implementovaná je podľa definovaných pravidiel knižnice *plop* a dekomponovaná do 5 modulov:

- *CreateSbspa,*
- *AppGenerator,*
- *Prompts,*
- *Actions,*
- *Utils.*

CreateSbspa je modul, ktorý sa stará o inicializáciu a spustenie knižnice *plop*. *AppGenerator* je hlavný modul generátora, ktorý riadi spracovávanie užívateľského vstupu a výber sémantických šablón. K spracovaniu využíva modul *Prompts*, ktorý definuje parametre a užívateľské výzvy. Navyše vykonáva kontrolu správnosti vstupu a výber nadväzujúcich výziev, ktoré sú závislé na aktuálne získaných parametroch. Po dokončení všetkých výziev a získaní parametrov od užívateľa, dochádza k samotnej voľbe sémantických šablón. *AppGenerator* k tomu využíva modul *Actions*, ktorý riadi výber akcií, ktoré v závislosti na vstupných parametroch vyberú sémantické šablóny a vygenerujú zdrojový kód. Modul *Utils* obsahuje pomocné funkcie pre spracovávanie a úpravu vstupných parametrov. Vzťahy medzi jednotlivými modulmi sú zobrazené na obrázku 5.3. V nasledujúcich častiach sú detailnejšie rozobrané princípy jednotlivých modulov generátora a ich kľúčové funkcie.



Obr. 5.3: Vzťahy medzi modulmi generátora.

Modul CreateSbspa

Implementáciu modulu *CreateSbspa* je možné nájsť v súbore *create-sbspa.js*. Zodpovedná je za inicializáciu knižnice *plop* a jej spustenie, ktoré zabezpečí vytvorenie generátora. Tento modul slúži ako inicializačný konzolový skript, ktorý je volaný po inštalácií a spustení aplikácie. K behu využíva prostredie *Node*, v ktorom zavolá príkaz `plop`. Argumentom tohto príkazu je koreňový skript generátora.

Modul AppGenerator

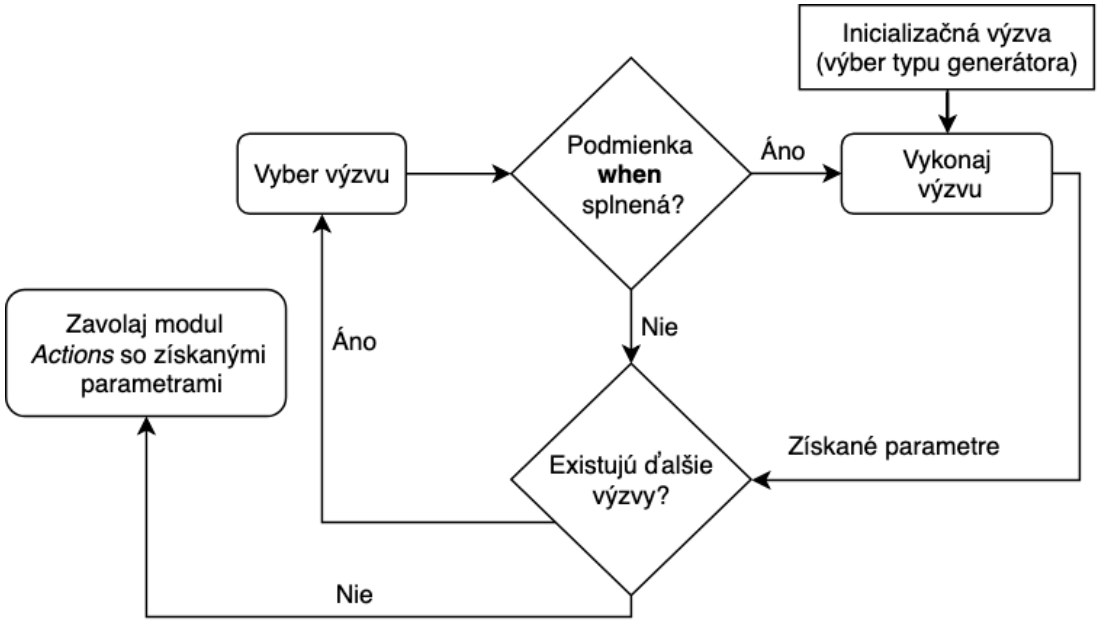
Implementáciu modulu *AppGenerator* je možné nájsť v súbore *AppGenerator.js*. Tento modul je zodpovedný za vytvorenie inštancie generátora, ktorý obsahuje definíciu užívateľských výziev a sémantických šablón. Registrácia generátora je zabezpečená modulom *CreateSbspa* a koreňovým skriptom, ktorý sa nachádza v súbore *index.js*. Pomocou *plop* funkcie `setGenerator` je generátor sprístupnený užívateľovi.

Medzi dôležité vlastnosti tohto modulu patrí definícia objektu *prompts* a *actions*. *Prompts* definuje užívateľské výzvy, ktoré slúžia na získavanie potrebných parametrov. Okrem základnej výzvy, ktorá určuje, ktorá časť aplikácie sa vygeneruje, obsahuje aj výzvy špecifické pre konkrétnu súčasť aplikácie. Tieto výzvy sú definované v module *Prompts* a načítané do modulu *AppGenerator* pomocou funkcie `prompts`. Objekt *actions* definuje akcie, ktoré sa majú zavolať po získaní užívateľského vstupu. K tomu využíva modul *Actions*, ktorý definuje sémantické šablóny (akcie) pre jednotlivé súčasti aplikácie. Akcie sú načítané do modulu využitím funkcie `actions`. Medzi ďalšie nastavenia *AppGenerator* patrí možnosť definície pomocných funkcií a parametrov, ktoré budú predané závislým modulom.

Modul Prompts

Implementáciu modulu *Prompts* je možné nájsť v súbore *prompts/index.js*. Inštancia tohto modulu je zodpovedná za výber užívateľských výzviev. K tomu využíva *plop* parameter `when`, v ktorom z doposiaľ získaných užívateľských vstupov vyhodnotí, či má danú výzvu zobraziť alebo nie.

Navyše zabezpečuje validáciu vstupných parametrov a nastavenie predvolených hodnôt. K tomu využíva metódy modulu *Utils*. Funkcia `getFilePath` upraví vstupný parameter užívateľa do správneho tvaru pre pomenovanie súboru a funkcia `isValidGenerator` overí, či je daná výzva povolená pre danú možnosť generátora. Na obrázku 5.4 je vizualizovaný proces výberu užívateľských výziev.



Obr. 5.4: Proces výberu užívateľských výziev.

Po zvolení typu generátora, je výber ďalších výziev riadený v závislosti na získaných parametroch z predchádzajúcich výziev. To umožňuje vyberať len nevyhnutné výzvy na vygenerovanie zvolenej časti projektu.

Modul Actions

Implementácia tohto modulu je umiestnená v súboroch *actions.js*. Je rozdelená do viacerých súborov v závislosti na časti aplikácie, ktorú generuje. Nachádzajú sa tu akcie pre *frontend*, *backend* a celý projekt. Modul na vstupe prijíma získané vstupné parametre užívateľa, na základe ktorých vygeneruje kód. Každá akcia definuje šablónu, výstupný súbor a časti aplikácie, pre ktoré je platná. Vďaka filtrovacej funkcií sú vybrané len tie akcie, ktoré patria pod danú časť aplikácie.

Dôležitou súčasťou je aj *plop* parameter *type*, v ktorom je možné povoliť prípadne zakázať modifikáciu súboru. To je prínosné hlavne z dôvodu, ak užívateľ nechce zámerne prepísať svoj kód. Modul využíva dve funkcie modulu *Utils*. Rovnako ako modul *Prompts* využíva pomocnú funkciu `getFilePath` a navyše funkciu `getActionObject`, ktorá rozhodne, či vytvorí alebo modifikuje existujúci súbor pri generovaní zdrojového kódu.

Modul Utils

Tento modul implementuje spomenuté pomocné funkcie pre ostatné moduly a nachádza sa v súbore *helpers.js*. Medzi kľúčové funkcie patrí vyhodnocovanie *type* parametru *Actions* modulu (funkcia `getActionObject`) alebo overovanie validity akcie v závislosti na zadaných vstupných parametroch (funkcia `isValidGenerator`). Okrem toho tu patrí funkcia `getFilePath`, ktorá upraví názov súboru do správneho formátu.

5.2 Štruktúra single-page aplikácie

Neodeliteľnou súčasťou generátora sú sémantické šablóny, ktoré boli vytvorené zo zdrojového kódu, podľa návrhu aplikácie. Výstupom generátora po získaní užívateľských parametrov a zavolaní vybraných akcií je prevod sémantických šablón do zdrojového kódu.

Výsledkom toho je spustiteľná aplikácia, ktorá obsahuje vybrané časti *single-page* aplikácie a poskytuje nástroje pre jej efektívnejší vývoj. V tejto sekcií sú popísané implementačné detaily nástrojov a modulov, ktoré boli uvedené v návrhu štruktúry *single-page* aplikácie v sekcií 4.2 a vygenerované pomocou sémantických šablón.

5.2.1 Popis backendu

Backend časť kódu je spustiteľná aplikácia, ktorá využíva framework *Spring-boot* a voliteľnú implementáciu API pomocou *GraphQL*.

Zostavenie a spustenie aplikácie

Konfigurácia závislostí, nástrojov a zostavenia aplikácie sa nachádza v súbore *build.gradle*. Okrem spustenia aplikácie pomocou nástroja *Gradle*, je vývojárovi dostupná aj konfiguráciu *Docker* kontajneru. Umožňuje jednoduché nasadenie a škalovanie aplikácie. Navyše obsahuje nástroje na tvorbu testov pre *GraphQL* a Java triedy.

Kľúčovou triedou *Spring Boot* aplikácie je trieda *Application.java*, ktorá obsahuje metódu `main`. Zabezpečuje vytvorenie a spustenie inštancie aplikácie. Spustenie aplikácie je možné nástrojom *Gradle* a príkazom `runApi`.

Kontrola kódu

Statická kontrola kódu je zabezpečená vďaka konfiguráciám nástrojov *Findbugs* a *Checkstyle*, ktoré sa nachádzajú v adresári *config*. Kontrolu je možné spustiť pomocou nástroja *Gradle* príkazom `checkstyleMain` a `findbugsMain`. Nástroje obsahujú predkonfigurované pravidlá statickej kontroly používané pri tvorbe produkčných projektov. Výstup statickej kontroly je dostupný v adresári */build/reports* v HTML dokumente, ktorý popisuje jednotlivé chyby a miesta ich výskytu. Proces kontroly je spustený počas zostavenia aplikácie, kde vývojára o výskytoch chýb informuje.

Konfigurácia GraphQL

Vygenerovaná časť aplikácie *GraphQL*, okrem samotnej konfigurácie poskytuje aj jednoduchý príklad použitia tejto knižnice. Vývojár je schopný jednoduchšie pochopiť použitie *GraphQL* a vyskúšať tak funkčnosť na vzorovom príklade. Navyše sú všetky konfigurácie dostupné, vďaka čomu je ich možné modifikovať prípadne odstrániť. Vygenerovaný príklad obsahuje definíciu schémy (*blog.graphqls*) a triedu pre *Query* (*Query.java*), Resolvers (*PostResolver.java*) a *Mutation* (*Mutation.java*). Po spustení je vývojárovi sprístupnený nástroj s užívateľským rozhraním pre testovanie *GraphQL* dotazov.

5.2.2 Popis frontendu

Frontend časť kódu je spustiteľná aplikácia, ktorá využíva knižnicu *React*. Statické typovanie je zabezpečené prostredníctvom jazyka Typescript, ktorý je zostavením zkompilovaný do jednoduchého Javascript.

Zostavenie aplikácie

Hoci je *frontend* aplikácia rovnako ako *backend* spustiteľná pomocou nástrojov *Gradle* a *Docker*, na zostavenie využíva nástroj *Webpack*[3]. Konfigurácia zostavenia aplikácie sa nachádza v súbore *webpack.config.js*. Umožňuje vytvoriť produkčné a vyvojové zostavenie aplikácie, ktoré zabezpečí transpiláciu zdrojového kódu.

Dôležitou časťou sú definície a nastavenia knižníc, ktoré zabezpečujú spracovanie *scss* súborov, minifikáciu a optimalizácie kódu, generovanie stránok a dodatočné transpilácie. Jednou z nich je aj knižnica *Babel*[4]. Používa sa prevažne na prevod kódu do spätne kompatibilnej verzie Javascriptu v súčasných a starších prehliadačoch. Konfiguračné nastavenia sa náchádzajú v súbore *.babelrc.js*. Obsahujú sadu transpilačných pravidiel a množinu podporovaných prehliadačov. Vďaka tomu je možné zabezpečiť kompatibilitu skrz väčšinu moderných prehliadačov a prostredí.

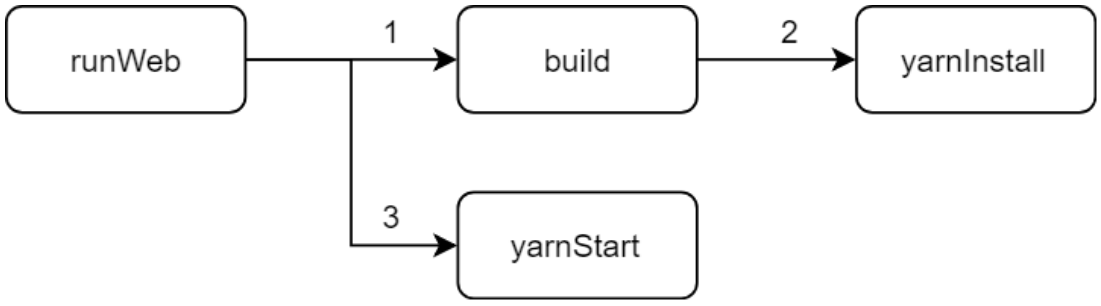
Použitie dvoch typov zostavení, umožňuje vývojárovi definovať odlišné knižnice a nastavenia v závislosti na type. Vývojové zostavenie aplikácie je zamerané na zjednodušenie testovania a vývoja webovej aplikácie. Podporuje HMR a zachováva stav aplikácie, ktorý by sa stratil počas úplného načítania. Na druhú stranu produkčné zostavenie aplikácie obsahuje konfigurácie, ktoré zabezpečia na výstupe optimalizovaný a minifikovaný produkčný kód. Vďaka tomu je možné vytvoriť progresívnu webovú aplikáciu s *offline* podporou a odolnosťou na problémy siete.

Aplikačné skripty a závislosti sa nachádzajú v súbore *package.json*. Skripty je možné volať pomocou nástroja *yarn* alebo *npm*. Okrem toho je vývojár schopný nastaviť aj ďalšie konfigurácie, ako sú názov projektu a verzia Node.

Spustenie aplikácie

Po zostavení je možné aplikáciu spustiť využitím aplikačného skriptu. Produkčné zostavenie aplikácie využíva na spustenie lokálneho servera knižnicu *serve*[5] a vývojové knižnicu *Webpack* s nástrojom *Webpack Dev Server*.

Okrem aplikačných skriptov je možné aplikáciu spustiť aj využitím príkazu `gradle` a metódy `runWeb`. Na obrázku 5.5 je vizualizovaný proces spustenia aplikácie s jednotlivými krokmi.



Obr. 5.5: Proces spustenia aplikácie príkazom `runWeb`.

[3] https://webpack.js.org/
[4] https://babeljs.io/
[5] https://www.npmjs.com/package/serve

Po vyvolaní metódy `runWeb` je v prvom kroku zavolaná metóda `build`, ktorá pomocou metódy `yarnInstall` nainštaluje aplikačné závislosti. V ďalšom kroku je aplikácia zostavená a následne metódou `yarnStart` spustená.

Testovanie aplikácie

Testovanie je dôležitou súčasťou každej produkčnej aplikácie. Okrem overenia funkcionality má aj dokumentačný charakter. Vygenerovaná aplikácia obsahuje konfigurácie testovacieho frameworku *Jest* v súbore *jest.config.js*. Dôležitou súčasťou konfigurácie je parameter `collectCoverageFrom`, ktorý vymedzuje koleciu tried, pre ktoré si uchováva informácie o pokrytí kódu testami. Testy je možné spustiť pre všetky definované moduly pomocou aplikačného skriptu `test:coverage`. Skript na výstup vypíše výsledok kontroly pokrytia kódu.

Kontrola kódu

Podľa návrhu štruktúry aplikácie, boli implementované konfigurácie nástrojov *Tslint* (*tslint.json*) a *Prettier* (*.prettierrc*). Použitím týchto nástrojov je možné zabezpečiť efektívnejšiu kontrolu kódu a vyhnúť sa tak častým vývojárskym chybám. Konfigurácia nástrojov obsahuje sadu doporučených pravidiel, ktoré sa používajú pri implementácií produkčných aplikácií. Tieto pravidlá je vývojár schopný upraviť prípadne zakázať.

Aplikačné moduly

Okrem implementovaných konfigurácií, generátor poskytuje moduly, ktoré pomáhajú riešiť časté implementačné výzvy, s ktorými sa vývojári stretávajú. Jednou z nich je aj asynchrónne načítanie modulov, ktoré bolo popísané v kapitole č. 4.2. Vďaka použitiu *Webpack* nástroja pre zostavenie aplikácie, je možné dekomponovať *single-page* aplikáciu na menšie časti, ktoré sa načítajú asynchrónne. Implementácia sa nachádza v súbore *App.tsx*.

Využitím knižnice *react-router*[6] je aplikácia rozdelená do samostatných stránok, ktoré sa pomocou *React* funkcie `lazy` a komponenty `Suspense` načítajú až pred vykreslením. Samotná komponenta, ktorá zabezpečuje asynchronné načítanie stránok je implementovaná v súbore *Router.tsx*. Na obrázku 5.6 je zobrazená spolupráca medzi týmito komponentami.

Obr. 5.6: Spolupráca tried pri asynchrónnom načítaní stránok.

Každá url adresa stránky má určenú cestu ku koreňovej komponente, ktorá sa má asynchrónne načítať. Trieda *App* pomocou komponenty *DynamicRoute* vytvorí mapovanie, ktoré pre danú URL vykreslí definovanú komponentu. *Router* komponenta zabezpečí prostredníctvom *React* funkcií a funkcie `getAsyncComponent` ich načítanie.

[6] https://reacttraining.com/react-router/web/guides/quick-start

Medzi ďalšie časti, ktoré sú poskytnuté vývojárovi patrí implementácia založená na architektúre *Flux*, ktorá je uvedená v kapitole č. 2.2. Využíva knižnicu *redux*[7] a *redux-saga*[8]. Implementačné súbory sa nachádzajú v adresári *common/store/* a sú rozdelené do niekoľkých častí:

- *Store*,
- *ReducerRegistry*,
- *Saga*.

Store je zodpovedný za správu dát a ich aktualizáciu. Implementácia tohto modulu sa nachádza v súbore *store.tsx*. Zameraná je prevažne na registráciu *reducers* a *sagas*. Veľký rozdiel od *Flux* architektúry je spôsobený apsenciou komponenty *Dispatcher* a možnosti použitia viacerých *store*. Knižnica *redux* využíva len jeden *store* a *reducer*. *Reducer* je schopný pre definované akcie aktualizovať aplikačný stav v *store*. *ReducerRegistry* je implementovaný v súbore *reducerRegistry.tsx* a vývojárovi poskytuje možnosť dynamicky vytvárať viaceré nezávislé *reducers* a spájať ich do jedného. *Sagas* umožňujú reagovať na definované akcie v UI vyvolaním asynchrónnych funkcií (tzv. ság). To je prínosné predovšetkým pri volaní asynchrónnych funkcií, medzi ktoré patrí napríklad doťahovanie dát zo servera. Implemetácia sa nachádza v súbore *sagas.tsx*. Umožňuje dynamicky pridávať a modifikovať ságy.

Dynamické pridávanie *reducers* a *sagas* je obzvlášť prínosné pri asynchrónnom načítaní komponent a separácií aplikácie do logických častí. Pomocou implementovanej funkcie `withInjectedReducersAndSagas`, ktorá sa nachádza v súbore *helpers.tsx*, je možné dynamicky pridávať *reducers* a *sagas*. Na vstupe je definovaný zoznam *reducers* a *sagas*, ktoré sú následne nato registrované do *store*. Pri viacnásobnom načítaní komponenty, je zabranené viacnásobnej registrácií a strate aplikačného stavu. Príklad komponenty s dynamickým pridanými *reducers* a *sagas* sa nachádza v prílohe C.

5.3 Vzorové materiály pre WAP

Okrem užívateľskej dokumentácie, bolo vytvorených aj niekoľko videí, ktoré je možné použiť pri vyuke v predmete WAP. Použitie implementovaného generátora (*Create Sbspa*) je názorne ukázané v 3 videách, ktoré sa nachádzajú na priloženom DVD. Video *Generator_setup.mp4* je zamerané na ukážku inštalácie knižnice *Create Sbspa*. Navyše demonštruje generovanie projektu, spustenie testov a aplikácie. Ďalšie vytvorené video *Generator_options.mp4* názorne predvádza použitie dostupných funkcií generátora, generovanie a modifikáciu častí projektu. Posledné video *Dynamic_reducer_and_saga_injection.mp4* sa venuje dynamickému vytvoreniu *reducers* a *sagas*, ktoré bolo popísané v predchádzajúcej časti. Názorne ukazuje vytvorenie akcií a ich použitie v komponentách.

[7] https://redux.js.org/
[8] https://redux-saga.js.org/

Kapitola 6

Vzorová aplikácia

Ďalšou časťou diplomovej práce bolo vytvorenie užívateľského rozhrania k nástroju vzikajúcemu v rámci projektu *Integrated platform for analysis of digital data from security incidents* (TARZAN) riešenému na VUT FIT. Kapitola je rozdelená do dvoch častí. Prvá časť je venovaná návrhu vzorovej aplikácie využitím implementovaného generátora *Create Sbspa*. Druhá časť je zameraná na popis implementácie jednotlivých častí a funkcionalít aplikácie.

6.1 Návrh aplikácie

V tejto sekcií je uvedený návrh vzorovej *single-page* aplikácie, ktorá slúži na ukážku výhod použitia generátora. Podstatná časť je venovaná vysvetleniu štruktúry aplikácie a nástrojov, ktoré budú pri návrhu použité. Aplikácia slúži ako webové rozhranie pre službu *Toreator*[1], ktorá umožňuje získavať informácie o IP adresách v Tor sieti.

Pomocou generátora je vygenerovaný *backend* a *frontend* aplikácie. Keďže samotný *Toreator* pracuje ako *backend* služba, vygenerovaná *Spring Boot* aplikácia bude fungovať ako cachovateľné proxy s podporou *GraphQL*. Takéto zapojenie umožní lepšiu škálovateľnosť aplikácie a možnosť presunu služby *Toreator* do privátnej siete. Vygenerovaná *Docker* konfigurácia umožňuje vytvorenie viacerých nezávislých kontajnerov, ktorým bude rozdeľovaná záťaž. V nasledujúcich častiach je uvedený podrobnejší popis *frontend*, *backend* a škálovania aplikácie využitím navrhnutého generátora. Služba *Toreator* pri návrhu vzorovej aplikácie poskytuje sadu endpointov pre vyhľadávanie informácií o IP adresách v Tor sieti. Navyše poskytuje možnosť získané dáta filtrovať v závislosti na dátume a čase aktivity.

Aplikácia používa vygenerované automatizované zostavenie aplikácie využitím nástroja *Gradle* a postupov *continuous integration*.

6.1.1 Popis frontendu

Pomocou generátora je vygenerovaný *frontend* s konfiguráciou optimalizovanej kompilácie a automatizovaného zostavenia aplikácie. Vďaka nakonfigurovaným nástrojom na písanie testov a kontrolu kódu je jednoduché, v pomerne krátkom čase, navrhnúť a implementovať webové rozhranie. Aplikácia umožňuje užívateľovi vyhľadávať a zobrazovať dáta získané z endpointov služby *Toreator*. Všetky dotazy klienta sú posielané na vygenerovaný *Spring Boot backend*, kde sú pomocou *GraphQL* mapované na odpovedajúce endpointy.

[1] http://toreator.fit.vutbr.cz/

Návrh užívateľského rozhrania by mal byť pre užívateľa prehľadný a intuitívny. Z toho dôvodu je použitá knižnica *styled-components*[2], ktorá ponúka nástroje pre jednoduchú tvorbu komponent interaktívneho UI a bude použitá pri implementácií aplikácie. Pre správu stavu aplikácie je v návrhu použitá knižnica *Redux*[3], ktorá využíva návrhový vzor *Flux* spomenutý v sekcii 2.2. Vďaka tomu umožňuje lepšie zdieľať a kontrolovať stav aplikácie. Navyše poskytuje možnosť cachovať dotazy, čím zlepšuje užívateľský zážitok a zvyšuje výkonnosť aplikácie.

6.1.2 Popis backendu

Backend časť, ktorá je vygenerovaná pomocou generátora, využíva *GraphQL* API pre komunikáciu s webovým rozhraním aplikácie. Všetky prichádzajúce dotazy sú mapované na endpointy služby *Toreator*, čo pomáha zredukovať počet potrebných dotazov klienta a znížiť tak celkovú réžiu.

Pre potreby cachovania sú použité dostupné *Spring* anotácie, ktoré umožňujú cachovať dotazy a tým znížiť zaťaženie na službu *Toreator*.

6.1.3 Škálovanie aplikácie

V tejto časti je popísaný návrh škalovania ukážkovej aplikácie. Samotná implementácia nie je súčasťou práce, pretože závisí na prostredí, na ktorom bude aplikácia prevádzkovaná.

Vďaka vygenerovanej konfigurácií *Docker images*, je aplikácia rozdelená do logických blokov, ktoré je vývojár schopný jednoducho škálovať. Z dôvodu, že inštancií *Docker* kontajnerov môže byť vo všeobecnosti ľubovoľné množstvo, je potrebné nad nimi vykonávať vyvažovanie záťaže (tzv. *load balancing*). V návrhu je použitý nástroj *Nginx*, ktorý bude vyvažovať záťaž medzi *Docker* kontajnery a poskytovať cachovanie pre dotazy. Navyše umožňuje upravovať hlavičky dotazov, odpovedí a odtieniť tak priamu komunikáciu klienta so službou *Toreator*.

Návrh je zameraný na horizontálne škálovanie, ktoré popisuje proces duplikovania inštancií aplikácie pre obsluhu väčšieho množstva prichádzajúcich pripojení. Jedným z častých spôsobov, ako zvýšiť výkonnosť aplikácie, je vytvoriť jeden proces pre každé jadro stroja. Týmto spôsobom sa dá násobiť a paralelizovať už efektívne riadenie dotazov v systéme *Node.js* [6].

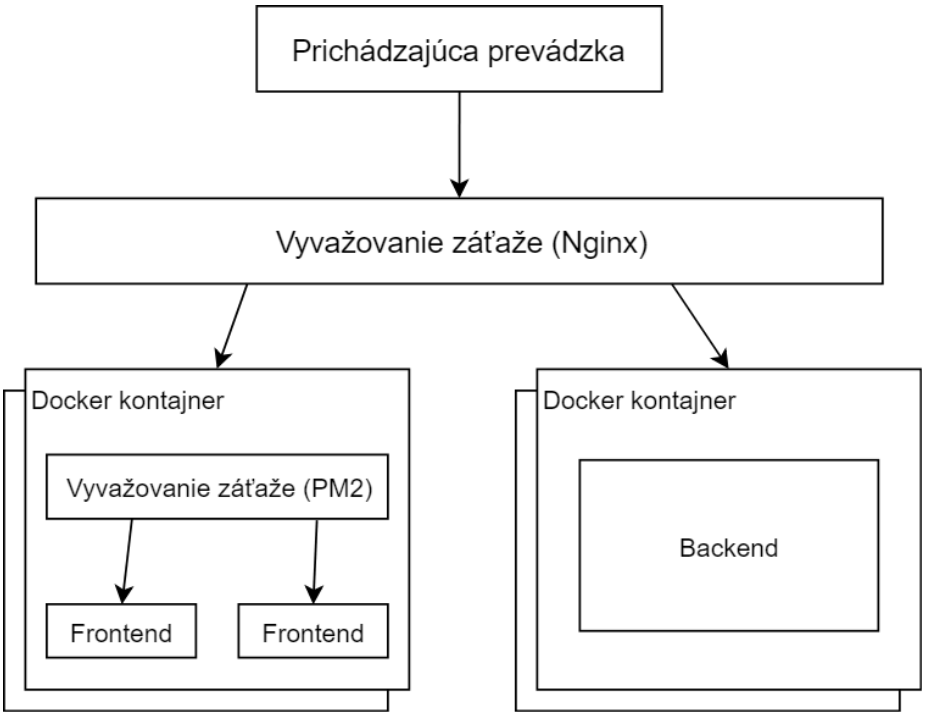
Vzorová aplikácia môže byť taktiež škálovaná v rámci jadier stroja (v kontajnery). Existujú rôzne stratégie pre škálovanie na jednom stroji alebo v rámci kontajneru. Bežnou koncepciou je, aby sa na tom istom porte spustili viaceré procesy a použije sa nástroj, ktorý medzi nich rozdelí pripojenia. V návrhu je použitá funkcionalita nástroja *PM2*, ktorý pracuje ako správca procesov. Pomáha škálovať aplikácie bez obavy o implementáciu klastru. Na obrázku 6.1 je detailnejšie zobrazený spôsob škálovanie vzorovej aplikácie.

Prichádzajúca prevádzka od klientov je rozdeľovaná medzi *Docker* kontajnery, ktoré obsahujú *backend* alebo *frontend* časť aplikácie. V rámci *frontend* kontajnerov beží správca procesov, nástroj *PM2*. Rozdeľuje pripojenia a pomáha vyvažovať záťaž medzi všetky jadrá procesoru.

Po spustení aplikácie v režime klastra, je možné nastaviť počet inštancií a vykonať opätovné spustenie. Procesy sa reštartujú sériovo, čím v aplikácií ostáva stále aspoň jeden

[2] https://www.styled-components.com/
[3] https://redux.js.org/

proces bežať. Architektúra aplikácie navyše pomáha vytvoriť základy pre použitie praktík *Continuous Deployment*[4].



Obr. 6.1: Škálovanie aplikácie a vyvažovanie záťaže (prevzaté z [6]).

6.2 Implementácia aplikácie

V tejto časti sú uvedené implementačné detaily aplikácie. Počas implementácie boli použité dostupné nástroje a moduly získané pomocou generátora *single-page* aplikácií. Detailná dokumentácia je dostupná na externom odkaze uvedenom v prílohe B.

6.2.1 Popis backendu

Použitím generátora bola vygenerovaná štruktúra *backend* časti projektu. Boli vytvorené konfigurácie *GraphQL* a *Docker*. Implementácia je dostupná v priečinku *toreator-proxy*. Jej hlavnou úlohou je realizácia nástroja, ktorý je schopný cachovať a preposielať dotazy na službu *Toreator*. Aplikácia má rovnaké možnosti zostavenia, spustenia, testovania a kontroly kódu, ako je uvedené v sekcií 5.2. Implementácia *backend* funkcionality je dekomponovaná do 5 modulov:

- *GraphQL* schéma,
- *Query*,
- *DAOs*,
- *Services*,
- *Resolvers*.

[4] https://www.scaledagileframework.com/continuous-deployment/

GraphQL schéma

GraphQL schéma je implementovaná v súbore *address.graphqls* a zodpovedá za definíciu objektov a dotazov, ktoré je možné použiť. Patrí sem definícia objektu *Address*, *Info*, *Response* a *Query*. Objekt *Address* reprezentuje IP adresu v *Tor* sieti. Okrem základných informácií[5] obsahuje aj zoznam objektov *Info*, ktoré reprezentujú jednotlivé záznamy o danej IP adrese. Objekt *Response* reprezentuje API odpoveď. Tento objekt je použitý pri implementácií možnosti zasielania ľubovoľných dotazov na službu *Toreator*. Definované objekty sú následne nato použité v objekte *Query*, ktorý definuje zoznam funkcií, kde každá funkcia reprezentuje jeden *GraphQL* endpoint.

Query

Trieda *Query* rozširuje triedu *GraphQLQueryResolver* a implementuje metódy, ktoré sú definované v *GraphQL* schéme v objekte *Query*. Implementácia sa nachádza v súbore *Query.java*. Metódy využívajú *DAO* triedy na spracovávanie a vyhodnocovanie dotazov. Odpoveď je po vyhodnotení odoslaná vo formáte, podľa schémy.

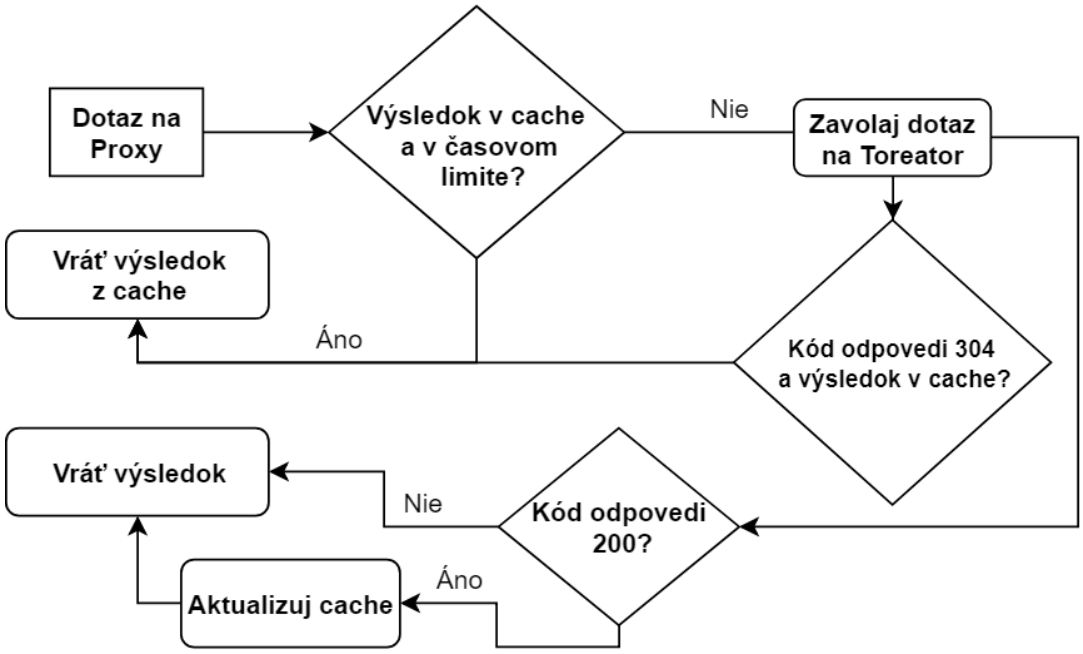
DAOs

Modul *DAOs* reprezentuje triedy, ktoré sú zodpovedné za sprácovávanie dotazov. Pre každý typ dotazu je vytvorená funkcia, ktorá zabezpečí jeho vyhodnotenie. Podľa objektov sú implementované triedy: *AddressDao*, *InfoDao* a *RequestDao*. Implementácia triedy *AddressDao* sa nachádza v súbore *AddressDao.java*. Zodpovedná je za spracovávanie dotazov na objekt *Address*. Trieda *InfoDao* je implementovaná v súbore *InfoDao.java* a spracováva dotazy na objekty typu *Info*. Poslednou triedou je trieda *RequestDao* implementovaná v súbore *RequestDao.java*, ktorá vyhodnocuje dotazy na objekt typu *Response*.

Funkcionalita týchto metód je z hľadiska kľúčových krokov rovnaká. Po prijatí dotazu v triede *DAO*, je zo vstupných parametrov vytvorený kľúč, ktorý slúži na získanie a uloženie výsledku do cache. Ak sa výsledok nachádza v cache a čas doby v cache je menší ako hraničný, použije sa tento výsledok. Minimálna doba v cache je definovaná užívateľskou premennou alebo hlavičkou `Cache-Control` a parametrom `max-age`.

V opačnom prípade je využitím definovanej *service* zavolaná služba *Toreator*. Spolu s dotazom je odoslaná aj HTTP hlavička `If-Modified-Since`, ktorá informuje službu *Toreator* o časovom razítku posledného dotazu. Po prijatí odpovedi s kódom 200 (*OK*) je výsledok uložený do cache s novým časovým razítkom a vrátený užívateľovi. Po prijatí odpovedi s kódom 304 (*Not Modified*) je použitý výsledok z cache. V inom prípade je vrátený výsledok bez uloženia do cache. Vizualizácia tohto procesu je zobrazená na obrázku 6.2.

[5] Napríklad IP adresa, adresa siete, maska.



Obr. 6.2: Vizualizácia procesu spracovania dotazu triedou *DAO*.

Services

Implementácia tohto modulu sa nachádza v priečinku *service*. *DAO* objekty pre vytváranie *Toreator* dotazov využívajú oddelené triedy. Pre každú *DAO* triedu existuje samostatná služba (*service*). Okrem toho sa tu nachádzajú aj 2 špeciálne triedy: *CacheServiceImpl* a *ApiServiceImpl*. Trieda *CacheServiceImpl* je implementovaná v súbore *CacheServiceImpl.java* a je zodpovedná za správu cache. Obsahuje metódy na pridávanie, mazanie a vyhľadávanie dát. Implementácia triedy *ApiServiceImpl* sa nachádza v súbore *ApiServiceImpl.java*. Obsahuje volanie služby *Toreator* a spracovávanie odpovedí. K tomu využíva triedu *RestTemplate*[6], ktorá implementuje klienta pre zasielanie dotazov na server. Konfigurácia tejto triedy sa nachádza v súbore *ApiConfiguration.java*. Okrem nastavenia URL adresy služby *Toreator*, obsahuje aj metódu `getRestHeaders`, v ktorej pridáva dotazom spomenutú hlavičku `If-Modified-Since`.

Medzi ďalšie triedy patrí *AddressServiceImpl*, *InfoServiceImpl* a *RequestServiceImpl*. Obsahujú prevažne funkcie na definíciu parametrov a URL adresy *Toreator* dotazov. Zložený dotaz následne pomocou služby *ApiServiceImpl* odošlú.

Resolvers

Tento modul obsahuje triedu *AddressResolver*, ktorá rozširuje triedu *GraphQLResolver*. Nachádza sa v súbore *AddressResolver.java* a je zodpovedná za vyhodnocovanie niektorých parametrov objektu *Address*. Zahrňuje hlavne tie, ktoré si vyžadujú dodatočné spracovávanie. Jedných z nich je aj parameter *info*, objektu *Address*. Zodpovedná je za ňu funkcia `getInfo`, ktorá volaním *DAO* metódy triedy *InfoDao*, vyhodnotí tento parameter. *GraphQL* vyhodnocuje objekty na základe dotazu. Vyhodnotené sú len tie parametre objektu,

[6] https://www.baeldung.com/rest-template

ktoré užívateľ vyžaduje. Vďaka tomu užívateľ prijme dátovú štruktúru, ktorú definoval, bez zbytočných parametrov.

6.2.2 Popis frontendu

Použitím generátora bola vygenerovaná štruktúra a *Docker* konfigurácia *frontend* časti projektu. Implementácia sa nachádza v adresári *toreator-web*. Zodpovedná je za realizáciu užívateľského rozhrania služby *Toreator*. Aplikácia ma rovnaké možnosti zostavenia, spustenia, testovania a kontroly kódu, ako je uvedené v sekcií 5.2.2. Navyše využíva aplikačné moduly generátora. Implementáciu je možné rozdeliť do 3 hlavných modulov, popísaných nižšie:

- *containers*,
- *ducks*,
- *components.*

Containers

Tento modul obsahuje implementáciu kľúčových komponent užívateľského rozhrania. Najdôležitejšou z nich je komponenta *HomePage*, ktorá sa nachádza v súbore *HomePage.tsx*. Zaobaľuje domovskú stránku, ktorá je jedinou stránkou aplikácie. Zodpovedná je za inicializáciu *reducers* a *sagas* a rozdeľuje UI do 2 časti: vyhľadávací formulár a zoznam výsledkov. Každá z týchto častí je implementovaná ako samostatná komponenta v tomto module.

Vyhľadávací formulár je implemementovaný v súbore *SearchForm.tsx*. Zameraný je na implementáciu formulárových prvkov, ich validáciu a vyhodnocovanie. Patrí sem výber filtrov implementovaný v module *Filters* a formulárové polia pracujúce s užívateľským vstupom (modul *CustomFilterInput* a *StyledInput*).

Získané výsledky sú spracované a zobrazené pomocou modulu *Results*, ktorý je implementovaný v súbore *Results.tsx*. Okrem zobrazenia výsledkov, obsahuje aj implementáciu dodatočných filtrov a histórie vyhľadávaní. Filtrovanie zoznamu výsledkov sa nachádza v module *ResultFilters*, ktorý je implementovaný v súbore *ResultFilters.tsx*. Umožňuje filtrovať zoznam výsledkov podľa definovaných filtrov[7]. História vyhľadávaní je implementovaná v module *History*, v súbore *History.tsx*. Zodpovedná je za vykreslenie zoznamu posledných užívateľských vyhľadávaní, čo umožňuje užívateľovi jednoduchší návrat k predchádzajúcim výsledkom.

Ducks

Modul *Ducks* je zameraný na implementáciu užívateľských akcií a funkcií, ktoré sú zodpovedné za ukladanie a aktualizáciu stavu aplikácie. Z hľadiska funkcionality je ho možné rozdeliť do 4 špecifických častí: *actions*, *reducers*, *sagas* a *selectors*.

Actions obsahujú implementáciu užívateľských akcií, ktoré aktualizujú stav vyhľadavacieho formuláru a histórie vyhľadávaní. Navyše obsahujú akcie, ktoré vytvoria a odošlú API dotaz na *backend*. Implementácia týchto akcií sa nachádza v súbore *actions/api.tsx*. Dotazy sú špecifikované v tele akcie spolu s parametrami, ktoré sa majú odoslať. Medzi hlavné parametre patrí parameter *query*, *cache* a *requestKey*. V parametri *query* je definovaný *GraphQL* dotaz. Cachovanie dotazov v prehliadači je možné vďaka parametri *cache*, ktorý využíva ako kľúč do cache, parameter *requestKey*.

[7] Napríklad filter IPv4 a IPv6.

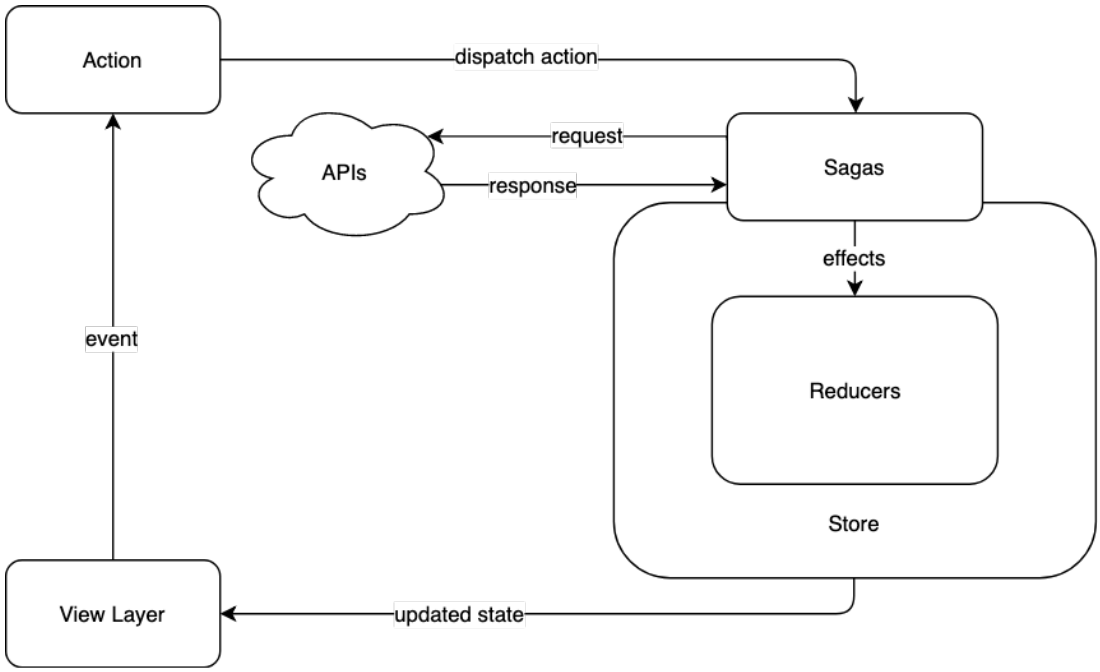
Reducers sú zodpovedné za implementáciu funkcií, ktoré na vstupe prijímajú akcie a ich výstupom je nový stav aplikácie. Tento stav môže byť aktualizovaný, prípadne totožný so stavom pred prevedením akcie. Každý *reducer* je schopný definovať, ako sa má zachovať pri určitej akcii. V tejto časti sa nachádzajú implementácie *reducers* pre vyhľadávací formulár, históriu vyhľadávaní a API volania.

Jedným z najdôležitejších je *reducer* pre API volania, ktorý je implementovaný v súbore *reducers/api.tsx*. Zodpovedný je za aktualizáciu stavu API volania, ktorá sa nachádza vo funkcii `reducer`. Umožňuje meniť stav dotazu a zaznamenávať si informácie o tom, či je dotaz spracovávaný, dokončený alebo skončil chybou. Okrem toho si ukladá informácie o poslednom dotaze a poslednej úspešnej odpovedi.

Vyhľadávací formulár využíva *reducer* implementovaný v súbore *reducers/searchForm.tsx*. Zodpovedný je za uloženie a aktualizáciu stavu formulára. Uchováva nastavenia vyhľadávacích, výsledkových filtrov a užívateľské vstupy. *Reducer* pre uchovávanie histórie vyhľadávaní je implementovaný v súbore *reducer/searchPath.tsx*. Zameraný je na aktualizáciu histórie vyhľadávaní a obsahuje odkazy na posledné vyhľadané dotazy užívateľa.

Sagas umožňujú reagovať na akcie UI volaním asynchrónnych funkcií, ktoré je možné jednoducho testovať. Ich použitie je obzvlášť prínosné pri volaní API dotazov. Implementácia tohto modulu sa nachádza v súbore *sagas/api.tsx*. Obsahuje metódu `rootSaga`, ktorá definuje dvojice akcia a funkcia. Pri výskyte akcie sa zavolá funkcia, ktorá ako parameter dostane akciu, ktorá ju vyvolala.

Medzi najdôležitejšie funkcie patrí funkcia `watchApiRequest`. Táto asynchrónna funkcia je zodpovedná za volania API. Pri každom volaní sa kontroluje, či je parameter akcie *cache* povolený a odpoveď sa nachádza v *store*. V kladnom prípade sa dotaz okamžite vyhodnotí bez potreby dotazu na *backend*.



Obr. 6.3: Vizualizácia závislosti medzi časťami *Ducks*.

Ducks poskytujú vývojárovi možnosť efektívnejšej kontroly UI a zdielania dát skrz celú aplikáciu. Na obrázku 6.3 sú vizualizované závislosti medzi popísanými časťami *Ducks*.

Stav aplikácie je uložený v *Store*, o ktorého obsluhu sa starajú *reducers*. Po aktualizovaní aplikačného stavu je aktualizovaná prezentačná vrstva (*View layer*). Po vyvolaní akcie *Action* v komponente, je akcia odoslaná k spracovávaniu do *Store*. Počas toho môže byť zachytená v *Sagas*, kde sú implementované asynchrónne funkcie (napríklad volanie API). Počas vykonávania asynchrónnych funkcií môžu ságy vyvolať nové akcie a efekty, ktoré aktualizujú *Store*.

Components

Components sa od *containers* odlišujú tým, že neobsahujú žiadnu logiku, ktorá upravuje dáta alebo mení stav aplikácie. Sú to jednoduché funkcie, ktoré slúžia na vykreslovanie časti UI. Patria tú dva kľúčové komponenty: *Info* a *AddressList*. Oba sú pre lepšiu prehľadnosť rozdelené do viacerých malých komponent.

Komponenta *Info* je implementovaná v súbore *Info.tsx*. Obsahuje funkciu `Info`, ktorá na vstupe prijíma dáta získané z API a na výstupe vracia HTML štruktúru, ktorá sa má vykresliť. Pre väčšiu prehľadnosť je komponenta dekomponovaná do viacerých častí, medzi ktoré patrí komponenta *Consensuses* a *MaxMindGeolocation*. Oba komponenty sa venujú spracovávaniu jednej časti API odpovede so špecifickou dátovou štruktúrou.

Komponenta *AddressList* je implementovaná v súbore *AddressList.tsx*. Narozdiel od predchádzajúcej komponenty, ktorá je zodpovedná za vykreslenie informácií o danej adrese, je funkcia `AddressList` zodpovedná za vykreslenie zoznamu adries.

Kapitola 7

Testovanie

Po dokončení implementácie nástroja *Create Sbspa*, ktorý umožňuje vygenerovať *single-page* aplikácie, bolo možné pristúpiť k samotnému testovaniu. Jeho priebeh je možné rozdeliť do dvoch fáz, pričom v každej z nich prebehlo porovnanie s nástrojmi s podobnou funkcionalitou. V prvej fázy testovania bolo nevyhnutné overiť výkonnosť jednoduchej *single-page* aplikácie a overiť dodržiavanie doporučených pravidiel z hľadiska tvorby progresívnych webových aplikácií a dostupnosti. Druhá fáza testovania je zameraná prevažne na porovnanie funkcionalít generátora s inými nástrojmi, pričom obsahuje zoznam významných funkcionalít a ich podporu skrz vybrané generátory.

Z dôvodu testovania kvalitatívnych parametrov vygenerovanej *single-page* aplikácie, bol zvolený nástroj *Lighthouse*[1], ktorý je dostupný prostredníctvom vývojárskych nástrojov prehľadača Chrome. Umožňuje automatizovane otestovať výbrané parametre stránky a poskytnúť tak detailný audit, ktorý okrem zoznamu chýb, obsahuje aj odkazy na doporučené riešenia.

Porovnávanie implementovaného generátora (*Create Sbspa*) s inými nástrojmi je dôležitou súčasťou testovacích scenárov, z dôvodu lepšieho overania prínosov a návrhu budúcich optimalizácií. Z dostupných nástrojov boli vybrané 2 najznámejšie:

- *jHipster,*
- *Create React app.*

Nástroj *jHipster* je vývojárska platforma, ktorá umožňuje generovanie, vývoj a nasadenie *single-page* aplikácií s podporou *Spring Boot*. Jej cieľom je vygenerovať modernú webovú aplikáciu, ktorá je koncipovaná z výkonného *backend* a *mobile-first frontend*. *Create React app* je nástroj implementovaný firmou Facebook, ktorý umožňuje vygenerovať *single-page* aplikáciu bez dodatočnej konfigurácie nástrojov, ako sú *Webpack* a *Babel*. Taktiež poskytuje sadu ďalších nástrojov, ktoré zefektívňujú samotný vývoj a testovanie.

7.1 Testovanie funkčnosti

Táto časť sa venuje prvej fázy testovania implementovaného nástroja, ktorá spočíva v testovaní funkčnosti *single-page* aplikácie. Konkrétny príklad je demonštrovaný na jednoduchej webovej aplikácií so stránkou s obsahom o 2000 znakoch. K získaniu výsledkov bol použitý nástroj *Lighthouse*. Výsledky testovania sú rozdelené 5 kategórií:

[1] https://developers.google.com/web/tools/lighthouse/

- *Performance*,
- *Accessibility*,
- *Best Practices*,
- *Progressive Web App*.

Performance výsledky hodnotia metriky, medzi ktoré patrí: *First Contentful Paint*, *Speed Index*, *First Meaningful Paint*, *Time to Interactive*, *First CPU Idle* a *Estimated Input Latency*. Metrika *First Contentful Paint* značí čas, kedy je vykreslený prvý text alebo obrázok na stránke. *Speed Index* označuje, ako rýchlo je načítaný obsah stránky. Metrika *First Meaningful Paint* značí dobu, kedy je vykreslený hlavný obsah stránky. *Time to Interactive* označuje čas, po ktorom je stránka interaktívna a užívateľ s ňou vie interagovať. *First CPU Idle* metrika označuje čas, kedy je hlavné vlákno stránky dostupné a vie reagovať na užívateľské akcie. Metrika *Estimated Input Latency* určuje dobu odozvy aplikácie na užívateľské vstupy.

Accessibility výsledky sú zamerané na hodnotenie samotnej dostupnosti stránky. Dostupnosť v tomto význame znamená, že obsah stránky je dostupný užívateľovi a vie s nim interagovať pomocou nástrojov, ako sú klávestnica, myš, dotykový displej a pod. Definuje metriky a pravidlá, ktoré zlepšujú dostupnosť pre užívateľa, skrz rôzne zariadenia a prehliadače.

Best Practices výsledky hodnotia dodržiavanie doporučených praktík vývoja, ktoré majú vplyv na bezpečnosť alebo na dostupnosť aplikácie na rôznych zariadeniach. Patrí tu použitie protokolu HTTPS, odstránenie použitia aplikačnej cache a zastaraných knižníc s bezpečnostnými rizikami.

Progressive Web App výsledky sú zodpovedné za hodnotenie *single-page* aplikácie z pohľadu dodržania aspektov progresívnej webovej aplikácie. Výsledky sú rozdelené do 3 časti:

- rýchlosť a spoľahlivosť (anglicky *Fast and reliable*),
- inštalovateľnosť (anglicky *Installable*),
- PWA optimalizácie (anglicky *PWA Optimized*).

Rýchlosť a spoľahlivosť aplikácie je meraná pomocou metrík, ktoré sledujú dostupnosť a dobu načítania stránky v *offline* režime. Inštalovateľnosť aplikácie je overená dostupnosťou a správnym konfiguračným nastavením *manifest*[2] súboru. PWA optimalizácie súvisia so správnym nastavením parametrov stránky, medzi ktoré patrí napríklad správna veľkosť obsahu voči stránke alebo automatické presmerovanie HTTP na HTTPS.

Prvým z testovaných nástrojov bol implementovaný generátor. Výsledky sú vizualizované pomocou bodového ohodnotenia v jednotlivých kategóriách. V tabuľke 7.1 sú uvedené výsledky pri použití simulovaného 3G pripojenia so 4 násobným spomalením CPU.

Performance	**Accessibility**	**Best Practices**	**Progressive Web App**
98/100	100/100	93/100	7/12

Tabuľka 7.1: Výsledky testovania *single-page* aplikácie so simulovaným 3G pripojením.

[2] https://developers.google.com/web/fundamentals/web-app-manifest/

Aplikácia dosiahla z hľadiska výkonnosti postačujúce výsledky. Podľa nástroja *Lighthouse* stránky s výsledkami nad 90 bodov patria do 5 percent najvýkonnejších stránok. Body potrebné pre získanie plného počtu bodov často krát vyžadujú aj nastavenia cache, komprimácie a ďalších parametrov na strane serveru, ktorý je závislý na výbere produkčného prostredia. Dostupnosť aplikácie získala 100 bodov, vďaka odstráneniu nedostatkov, ktoré boli v čase vývoja identifikované. Dodržiavanie doporučených pravidiel rovnako ako výkonnosť vyžaduje dodatočné optimalizácie na strane serveru pre získanie plného počtu bodov. Pravidlá vývoja progresívnej webovej aplikácie majú menšie nedostatky v nastavení *manifest* súboru, ktorý obsahuje informácie o aplikácií. Keďže vygenerovaná aplikácia slúži ako základ pri tvorbe *single-page* aplikácie a parametre sú špecifické pre každú aplikáciu, je toto nastavenie prenechané vývojárovi.

Ďalšie testovanie prebehlo na reálnom 3G pripojení so 4 násobným spomalením CPU. V tabuľke 7.2 sú uvedené výsledky meraní.

Performance	**Accessibility**	**Best Practices**	**Progressive Web App**
95/100	100/100	93/100	7/12

Tabuľka 7.2: Výsledky testovania *single-page* aplikácie s reálnym 3G pripojením.

Vo výsledkoch je možné vidieť pokles výkonnosti pri použití reálneho 3G pripojenia. Zníženie prenosovej rýchlosti má za následok zvýšenie doby potrebnej na načítanie stránky. Napríklad v metrike *Time to Interactive* došlo k nárastu z 2,2 sekundy na 2,9 sekundy. Výsledok toho je zníženie počtu bodov a zhoršenie celkového hodnotenia.

Porovnanie s vybranými generátormi

Rovnako boli testované aj *single-page* aplikácie vygenerované generátormi *Create React App* a *jHipster*. Použitý bol rovnaký príklad jednej webovej stránky s obsahom o 2000 znakoch. Výsledky *Create React App* sú zobrazené v tabuľke 7.3.

Connection	**Performance**	**Accessibility**	**Best Practices**	**Progressive Web App**
Simulated 3G, 4x CPU Slowdown	99/100	100/100	93/100	6/12
Applied 3G, 4x CPU Slowdown	98/100	100/100	93/100	6/12

Tabuľka 7.3: Výsledky testovania *single-page* aplikácie vygenerovanej nástrojom *Create React App*.

Výsledky sú vo všetkých kategóriách porovnateľné s implementovaným generátorom. Menšie zlepšenie vo výkonnosti je možné vidieť pri pomalšom pripojení (*Applied 3G, 4x CPU Slowdown*), vďaka použitiu väčšieho množstva optimalizácií. Na druhej strane je z hľadiska metrík dodržaných menej pravidiel progresívnych aplikácií ako u implementovaného generátora.

Posledným testovaným generátorom bol nástroj *jHipster*. Výsledky testovania *single-page* aplikácie sú zobrazené v tabuľke 7.4.

Connection	Performance	Accessibility	Best Practices	Progressive Web App
Simulated 3G, 4x CPU Slowdown	39/100	100/100	86/100	8/12
Applied 3G, 4x CPU Slowdown	30/100	100/100	86/100	8/12

Tabuľka 7.4: Výsledky testovania *single-page* aplikácie vygenerovanej nástrojom *jHipster*.

Výkonosť aplikácie sa výrazne zhoršila v porovnaní s predchádzajúcimi generátormi. Pri investigácií problému bolo identifikovaných niekoľko blokujúcich procesov, ktoré spôsobili spomalenie celej aplikácie. Medzi najpočetnejšie patria blokujúce dotazy na *css* a *js* súbory, ktoré spôsobujú blokovanie hlavného vlákna stránky, čo má za následok pomalšie vykreslenie.

7.2 Porovnanie funkcionalít

Okrem testovania funkčnosti, je dôležitým ukazateľom prínosu implementovaného riešenia aj porovnanie funkcionalít s vybranými nástrojmi. Táto časť je venovaná druhej fázy testovania, kde sú popísané významné funkcionality jednotlivých nástrojov. Spoločnou vlastnosťou všetkých generátorov je možnosť vygenerovať *single-page* aplikáciu, ktorá je bez dodatočnej konfigurácie spustiteľná.

Implementovaný generátor (*Create Sbspa*) podporuje základné konfigurácie pre vytvorenie spustiteľného *backend*. Na druhú stranu generátor *jHipster* ponúka veľké množstvo špecifických funkcionalít, ako sú autentifikácia, napojenie na databázu alebo konfigurácia cache. Tieto funkcionality nie sú nevyhnutné pri tvorbe *single-page* aplikácií, z toho dôvodu ich *Create Sbspa* neobsahuje. Zameraný je na jednoduchosť a prehľadnosť vygenerovaného projektu, využitím nástrojov, ktoré zefektívňujú vývoj a správu. Generátor *Create React App* umožňuje generovať len *frontend*, generovanie *backend* konfigurácie nepodporuje. V tabuľke 7.5 je uvedený zoznam *backend* funkcionalít s informáciou o ich platnosti v jednotlivých generátoroch.

Funkcionalita	Create Sbspa	Create React App	jHipster
Pregenerovanie súborov	Áno	Nie	Nie
Gradle konfigurácia	Áno	Nie	Áno
GraphQL konfigurácia	Áno	Nie	Nie
Docker konfigurácia	Áno	Nie	Nie
Konfigurácia testovacieho frameworku	Áno	Nie	Áno
Konfigurácia statickej kontroly	Áno	Nie	Nie

Tabuľka 7.5: Zoznam *backend* funkcionalít a ich platnosť vo vybraných generátoroch.

Medzi špecifické funkcionality *Create Sbspa* generátora, ktorými sa odlišuje od ostatných generátorov, patrí možnosť pregenerovať súbory, *GraphQL* konfigurácia, *Docker* konfigurácia a konfigurácia statickej kontroly. Pregenerovanie súborov umožňuje aktualizovať

konfiguračné súbory počas vývoja. *GraphQL* konfigurácia poskytuje základné nastavenia závislostí a použitia tejto knižnice. *Docker* konfigurácia umožňuje *backend* aplikáciu spustiť v *Docker* kontajnery a statická kontrola vytvára podrobné správy o chybách a bezpečnostných rizikách kódu.

Dôležitou časťou *single-page* aplikácií je *frontend*, na ktorý je kladený veľký dôraz z hľadiska poskytnutia maximálnej výkonnosti a dostupnosti. V tabuľke 7.6 je uvedený zoznam dôležitých funkcionalít s informáciou o ich platnosti v jednotlivých generátoroch.

Funkcionalita	**Create Sbspa**	**Create React App**	**jHipster**
Pregenerovanie súborov	Áno	Nie	Nie
Gradle konfigurácia	Áno	Nie	Nie
Docker konfigurácia	Áno	Nie	Nie
Tslint konfigurácia	Áno	Nie	Áno
Podpora typescript	Áno	Áno	Áno
Konfigurácia testovacieho frameworku	Áno	Áno	Áno
Konfigurácia statickej kontroly	Áno	Nie	Áno
Dostupné konfiguračné súbory zostavenia	Áno	Áno	Áno
Code splitting	Áno	Áno	Áno
Dynamické pridávanie *reducers* a *sagas*	Áno	Nie	Nie
Git hooks	Áno	Nie	Nie
Vytvorenie produkčného servera	Áno	Nie	Nie

Tabuľka 7.6: Zoznam *frontend* funkcionalít a ich platnosť vo vybraných generátoroch.

Medzi významné funkcionality *Create Sbspa* generátora, ktorými sa odlišuje od ostatných generátorov, patrí pregenerovnie súborov, *Gradle* konfigurácia, *Docker* konfigurácia, Dynamické pridávanie *reducers* a *sagas*, *Git hooks* a vytvorenie produkčného servera. Pregenerovanie súborov rovnako ako u *backend* umožňuje aktualizáciu vygenerovaného kódu. *Gradle* konfigurácia umožňuje automatizovane zostavovať *backend* a *frontend* spoločným nástrojom. *Docker* konfigurácia poskytuje možnosť vytvoriť *Docker* kontajner. Dôležitou súčasťou aplikácie je dynamické pridávanie *reducers* a *sagas*, ktoré umožňuje efektívnejšiu dekompozíciu *single-page* aplikácie. *Git hooks* pridávajú kontrolu kódu pri *commit* a *push* do centrálneho repozitára. Vytvorené produkčné zostavenie je bez potreby dodatočných nástrojov spustiteľné.

7.3 Zhodnotenie

Implementovaný generátor, z hľadiska funkčnosti a poskytnutých funkcionalít vytvára kompromis medzi vybranými nástrojmi. Spája jednoduchosť a optimalizačné nastavenia generátora *Create React App* a väčšie množstvo možností generovania, ktoré je dostupné v nástroji *jHipster*.

Narozdiel od generátora *Create React App* umožňuje vygenerovať aj *backend* časť aplikácie. Zároveň však zachováva voľnosť výberu jednotlivých častí. Generátor *jHipster* vytvára komplexné aplikácie, so špecifickými funkcionalitami, bez voľnosti výberu. Tým generuje veľké množstvo kódu, ktoré nie je nevyhnutné pre vývoj *single-page* aplikácií.

Create Sbspa je zameraný na nástroje, ktoré zefektívňujú vývoj a sú nevyhnutnou súčasťou pri tvorbe *single-page* aplikácií. Poskytuje možnosť výberu častí aplikácie pred a počas vývoja, pričom zachováva jednoduchosť vygenerovaného kódu a implementáciu špecifických funkcionalít ponecháva na vývojára. Nástroj je dostupný *online* v podobe *npm* knižnice[3] s aktuálnym počtom viac ako 100 stiahnutí.

[3] https://www.npmjs.com/package/create-sbspa

Kapitola 8

Záver

Cieľom tejto diplomovej práce bolo navrhnúť a implementovať nástroj, ktorý povedie k zefektívneniu tvorby *single-page* aplikácií. Pri návrhu bol kladený dôraz na prehľadnosť a jednoduchú rozšíriteľnosť o nové funkcionality. Z toho dôvodu, navrhnutý spôsob umožňuje vývojárovi pridávať a aktualizovať časti projektu.

Pred samotným návrhom nástroja, bolo nevyhnutné identifikovať problémové miesta vývoja, ktoré môžu spomaliť jeho začiatok. Taktiež bolo potrebné sa zoznámiť s aktuálne používanými frameworkami na tvorbu *single-page* aplikácií, naštudovať ich princípy a prínosy do moderného vývoja webových aplikácií a v neposlednom rade zistiť ich možnosti spolupráce s frameworkom *Spring Boot*. V ďalšej časti boli identifikované najčastejšie úskalia vývoja, ktoré majú negatívny dopad na kvalitu softvérového produktu a ich riešenie, je častokrát náročné a výrazne spomaľuje začiatok vývoja. Na základe uvedených problémov boli popísané vybrané nástroje, ktoré uľahčujú tvorbu, testovanie, udržovanie a nasadzovanie *single-page* aplikácií.

Získané informácie slúžili ako základ pri tvorbe návrhu, ktorého hlavnou úlohou bolo zefektívnenie tvorby *single-page* aplikácií a odstránenie problémových miest vývoja. Návrh obsahuje definované časti projektu a vybrané nástroje uľahčujúce vývojový proces. Navyše popisuje spôsob generovania sémantickými šablónami využitím vybraných nástrojov a užívateľské rozhranie, ktoré vývojárovi poskytne interaktívny výber a správu šablón.

Výsledkom tohto kroku je návrh v podobe generátora, ktorý umožňuje vygenerovať časti projektu. Tieto časti obsahujú štruktúru projektu s konfiguráciami pre vybrané nástroje, ktoré pomáhajú vytvárať rozšíriteľné a škálovateľné aplikácie. Vývojár môže pomocou rozhrania špecifikovať časti, ktoré sa majú pridať, prípadne aktualizovať.

Následne nato bolo možné pristúpiť k implementácií samotného generátora. Okrem navrhnutých konfiguračných nastavení nástrojov, boli implementované aj aplikačné moduly, ktoré umožňujú efektívnejšie dekomponovať *single-page* aplikácie. Týmito modulmi bola optimalizovaná výkonnosť a rýchlosť načítania aplikácie, vďaka čomu je vytvorená *single-page* aplikácia schopná zredukovať množstvo závislostí medzi jej nezávislými časťami.

Implementovaný generátor bol otestovaný na vzorovej aplikácií, ktorá bude použitá pre účely projektu TARZAN riešenému na VUT FIT. Táto aplikácia realizuje webové rozhranie pre službu *Toreator*. Pri jeho tvorbe použila nástroje generátora a doporučené praktiky pre vytvorenie škálovateľnej a rozšíriteľnej aplikácie. Hlavnou implementačnou výhodou použitia generátora bol rýchly štart vývoja, ktorý nevyžadoval dodatočné nastavovanie knižníc a nástrojov.

Okrem toho bol generátor *Create Sbspa* testovaný z hľadiska funkčnosti a podporovaných funkcionalít. Získané výsledky boli porovnané s 2 vybranými generátormi, *Create React App*

a *jHipster*. *Create Sbspa* splnil stanovené ciele, vďaka ktorým vytvoril kompromis medzi vybranými nástrojmi. Spája jednoduchosť a optimalizačné nastavenia generátora *Create React App* a väčšie množstvo možností generovania, ktoré je dostupné v nástroji *jHipster*.

Ďalší vývoj tejto práce je možné venovať implementácií nových možností generovania a optimalizácií aktuálnych konfiguračných nastavení. Rozširovanie možnosti generátora poskytne nástroje pre väčšie množstvo projektov, veľkého, či malého charakteru. Nevyhnutné je zachovať jednoduchosť a voľnosť výberu častí aplikácie, ktorá je dôležitá pre jej ďalšie použitie. Okrem toho je potrebné udržovať nástroj aktuálny a reagovať na najnovšie trendy vývoja webových aplikácií.

Literatúra

[1] *Angular For Beginners Guide: Why Angular? Understanding The Top Benefits.* [Online; navštívené 15.10.2018].
URL https://blog.angular-university.io/why-angular-angular-vs-jquery-a-beginner-friendly-explanation-on-the-advantages-of-angular-and-mvc/

[2] *Attribute Directives.* [Online; navštívené 15.10.2018].
URL https://angular.io/guide/attribute-directives

[3] *Checkstyle.* [Online; navštívené 04.11.2018].
URL http://checkstyle.sourceforge.net/

[4] *Eslint.* [Online; navštívené 04.11.2018].
URL https://eslint.org/docs/about/

[5] *Find Bugs in Java Programs.* [Online; navštívené 04.11.2018].
URL http://findbugs.sourceforge.net/

[6] *Good practices for high-performance and scalable Node.js applications [Part 1/3].* [Online; navštívené 08.12.2018].
URL https://medium.com/iquii/good-practices-for-high-performance-and-scalable-node-js-applications-part-1-3-bb06b6204197

[7] *Gradle User Manual.* [Online; navštívené 03.11.2018].
URL https://docs.gradle.org/current/userguide/userguide.html

[8] *Gradle vs Maven Comparison.* [Online; navštívené 03.11.2018].
URL https://gradle.org/maven-vs-gradle/

[9] *GraphQL is the better REST.* [Online; navštívené 08.12.2018].
URL https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

[10] *Jest: Delightful JavaScript Testing.* [Online; navštívené 05.11.2018].
URL https://jestjs.io/

[11] *JUnit - Test Framework.* [Online; navštívené 04.11.2018].
URL https://www.tutorialspoint.com/junit/junit_test_framework.htm

[12] *React A JavaScript library for building user interfaces.* [Online; navštívené 20.10.2018].
URL https://reactjs.org/

[13] *Spring Boot.* [Online; navštívené 20.10.2018].
URL https://spring.io/projects/spring-boot

[14] *Structural Directives*. [Online; navštívené 16.10.2018].
URL https://angular.io/guide/structural-directives

[15] *Top 10-2017 Top 10*. [Online; navštívené 08.12.2018].
URL https://www.owasp.org/index.php/Top_10-2017_Top_10

[16] *Understanding Components*. [Online; navštívené 15.10.2018].
URL https://docs.angularjs.org/guide/component

[17] *What is Maven?* [Online; navštívené 03.11.2018].
URL https://maven.apache.org/what-is-maven.html

[18] Abdur Rahman, C. D.: *A framework for ultra-responsive light weight web application using Angularjs*. 4 2015, s. 1-2.
URL https://ieeexplore.ieee.org/document/7453857

[19] Ambler, T.; Cloud, N.: *JavaScript Frameworks for Modern Web Dev*. Apress, 2015, s. 155-189, ISBN 978-1-4842-0662-8.

[20] Ezell, W.: *Everything You Need to Know About Single Page Applications*. [Online; navštívené 14.10.2018].
URL https://dotcms.com/blog/post/everything-you-need-to-know-about-single-page-applications

[21] Gackenheimer, C.: *Introduction to React*. Apress, 2015, s. 1-20, ISBN 978-1-4842-1246-2.

[22] Hall, G.: *Adaptive Code: Agile coding with design patterns and SOLID priciples*. Microsoft Press, 2017, ISBN 978-1-5093-0258-1.

[23] Jenkov, J.: *Scalable Architectures*. [Online; navštívené 05.12.2018].
URL http://tutorials.jenkov.com/software-architecture/scalable-architectures.html

[24] Ján Bušfy, M. M.: *Úvod do GraphQL - workshop*. [Online; navštívené 08.12.2018].
URL https://onas.azet.sk/blog/30-nodejs-server-graphql-workshop/

[25] Kakkar, S.: *Implementation Aspects of Software Development Projects*. 2 2006, s. 1-2.
URL https://ieeexplore.ieee.org/document/4086221

[26] Koukia, A.: *Why Docker? Pros and Cons*. [Online; navštívené 07.11.2018].
URL https://koukia.ca/why-docker-pros-and-cons-949d104478c5

[27] Lawson, K.: *What is a Single Page Application and why do people like them so much*. [Online; navštívené 14.10.2018].
URL https://www.bloomreach.com/en/blog/2018/07/what-is-a-single-page-application.html

[28] Muschko, B.: *Why Build Your Java Projects with Gradle Rather than Ant or Maven?* [Online; navštívené 03.11.2018].
URL http://www.drdobbs.com/jvm/why-build-your-java-projects-with-gradle/240168608

[29] Prasad, P. R.: *A Detailed Study of Flux: the React.js Application Architecture*. [Online; navštívené 20.10.2018].
URL https://www.cabotsolutions.com/2017/01/detailed-study-flux-react-js-application-architecture

[30] Sanja Delcev, D. D.: *Modern JavaScript frameworks: A Survey Study*. 8 2018, s. 106.
URL https://ieeexplore.ieee.org/document/8448444

[31] Syer, D.: *Spring and Angular JS: A Secure Single Page Application*. [Online; navštívené 20.10.2018].
URL https://spring.io/blog/2015/01/12/spring-and-angular-js-a-secure-single-page-application

[32] V, D.: *Building Blocks of a Scalable Architecture*. [Online; navštívené 08.12.2018].
URL https://dzone.com/articles/component-load-testing

[33] Wheeler, K.: *Getting To Know Flux, the React.js Architecture*. [Online; navštívené 20.10.2018].
URL https://scotch.io/tutorials/getting-to-know-flux-the-react-js-architecture

Príloha A

Obsah DVD

DVD priložené k tejto práci obsahuje:

- Zdrojové kódy textu diplomovej práce v LaTeX(adresár *src/latex*)
- Text diplomovej práce vo formáte PDF (súbor *dip.pdf*)
- Zdrojové kódy implementovaného generátora (adresár *src/create-sbspa*)
- Zdrojové kódy ukážkovej aplikácie (adresár *src/toreator-ui*)
- Videá demonštrujúce použitie generátora (adresár *tutorials*)

Príloha B

Externé odkazy

Externé odkazy diplomovej práce sú:

- Zdrojové kódy generátora *single-page* aplikácií
  https://bitbucket.org/norbertdurcansk/sb-spa
- Zdrojové kódy vzorovej aplikácie
  https://bitbucket.org/norbertdurcansk/toreator-ui
- Knižnica *Create Sbspa* pre *Node.js*
  https://www.npmjs.com/package/create-sbspa

Príloha C

Príklad komponenty s dynamicky pridanými reducers a sagas

Komponenta

```
const ExampleComponent = ({value, myReducerAction}) =>
    <div onClick={()=>myReducerAction("click")}>
        My component with value: {value}!
    </div>
```

Dynamicky pridaný *reducer* a *saga*

```
export default compose(
    withInjectedReducersAndSagas({
        reducers:[
            {
                key: "MY_REDUCER",
                value: reducer
            }
        ],
        sagas:[
            {
                key: "MY_SAGA",
                value: saga
            }
        ]
    }),
    connect(
        (state)=>({
                value: state["MY_REDUCER"].value
            }),
        (dispatch)=>({
                myReducerAction: value => dispatch(myReducerAction(value))
            })
        )
    )(ExampleComponent);
```