Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Doménovo-špecifický jazyk pre vývoj webových aplikácií II

Diplomová práca

2020 **Bc. Aleš Buľko**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Doménovo-špecifický jazyk pre vývoj webových aplikácií II

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	Ing. Sergej Chodarev, PhD.
Konzultant:	Ing. Sergej Chodarev, PhD.

Košice 2020 **Bc. Aleš Buľko**

Názov práce:	Doménovo-špecifický jazyk pre vývoj webových aplikácií II
Pracovisko:	Katedra počítačov a informatiky, Technická univerzita v Košiciach
Autor:	Bc. Aleš Buľko
Školiteľ:	Ing. Sergej Chodarev, PhD.
Konzultant:	Ing. Sergej Chodarev, PhD.
Dátum:	3. 5. 2020
Kľúčové slová:	metaprogramovanie, generatívne programovanie, doménovo špecifický jazyk, webová aplikácia

Abstrakt: Táto práca sa zaoberá rozšírením existujúceho softvérového rámca BFI, ktorý slúži primárne na generovanie serverovej a klientskej časti webovej aplikácie. Základom rámca je doménovo-špecifický jazyk pre definovanie doménového modelu. Jazyk sme rozšírili o nové prvky a vniesli vyššiu modularitu umožnením použitia viacerých generátorov súčasne, ktoré sú teraz chápané všeobecnejšie. Pre zvýšenie komfortu práce sme vyvinuli zásuvný modul do vývojových prostredí Jetbrains, ktorý slúži na podporu editovania zdrojových súborov a spúštania generátorov prostredníctvom akcií v novozavedenom konfiguračnom súbore.

Thesis title:	Domain-specific language for web application development II
Department:	Department of Computers and Informatics, Technical university of Košice
Author:	Bc. Aleš Buľko
Supervisor:	Ing. Sergej Chodarev, PhD.
Tutor:	Ing. Sergej Chodarev, PhD.
Date:	3. 5. 2020
Keywords:	metaprogramming, generative programming, domain specific language, web application

Abstract: This work deals with the extension of the existing BFI software framework, which is used primarily to generate the server and client part of the web application. The framework is based on a domain-specific language for defining the domain model. We have expanded the language with new elements and introduced higher modularity by allowing the use of multiple generators at the same time, which are now understood more generally. To increase work comfort, we have developed a plug-in for Jetbrains development environments, which is used to support editing source files and running generators. through actions in the newly deployed configuration file.

57747

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**
Študijný program: **Informatika**

Názov práce:

Doménovo-špecifický jazyk pre vývoj webových aplikácií II

Domain-specific language for development of web applications II

Študent: **Bc. Aleš Buľko**
Školiteľ: **Ing. Sergej Chodarev, PhD.**
Školiace pracovisko: **Katedra počítačov a informatiky**
Konzultant práce:
Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:
1. Analýzovať a zhodnotiť rámec BFI.
2. Analýzovať potreby vývojárov webových aplikácii a existujúce riešenia pre zjednotenie definícii medzi klientskou a serverovou časťou aplikácie.
3. Návrhnuť úpravy a rozšírenia rámca BFI vzhľadom na poterby vývojarov.
4. Vyhodnotiť použiteľnosti rozšírenej implementácie rámca BFI.
5. Vypracovať dokumentáciu podľa pokynov vedúceho práce.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 04.05.2020
Dátum zadania diplomovej práce: 31.10.2019

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice, 3.5.2020 .....

Vlastnoručný podpis

Poďakovanie

Rád by som poďakoval svojmu vedúcemu práce pánovi Ing. Sergejovi Chodarevovi PhD., ktorý umožnil vznik tejto práce a poskytol odborné poznatky v oblasti metaprogramovania, ktoré tvoria jej piliere. Chcem sa tiež poďakovať kolegovi Bc. Matejovi Čupkovi za spoluprácu v odhaľovaní požiadaviek webových vývojárov a dobrých zvykov v tvorení tohto typu softvéru. Rovnako si cením čas všetkých respondentov, ktorí mi pomohli s vyhodnotením mojej práce a poskytli spätnú väzbu pre jej ďalšie zlepšovanie.

Obsah

Zoznam obrázkov

Zoznam zdrojových kódov

Zoznam tabuliek

1 Úvod

1.1 Motivácia

Ako vývojári si často po určitom čase začneme všímať jednotvárnosť v určitých oblastiach našej práce. Aj napriek tomu že vývoj informačných systémov považujeme za tvorivú, nie mechanickú činnosť, často krát určité časti kódu používame opakovane. Tento jav si všímame najmä pri vývoji **webových aplikácií**, kde používame rovnaké knižnice, kostry projektov, alebo pomocné triedy, ktoré nie sú dostatočne rozsiahle, alebo dostatočne univerzálne na to, aby sme ich oddelili do knižníc. Pri vytváraní nových projektov sa teda uchyľujeme ku **kopírovaniu** zdrojových kódov z predošlých projektov, čo odporuje myšlienke znovupoužiteľnosti kódu.

Vo všeobecnosti v oblasti softvérového vývoja existuje trend zvyšovania úrovne **abstrakcie**. Od jazykov nízkej úrovne, ktoré ponúkajú vysokú flexibilitu, avšak zložitú implementáciu, sme sa dostali k jazykom vyššej úrovne, ktoré nám za cenu nižšej flexibility ponúkajú tvoriť programy s vyššou úrovňou abstrakcie, a využívať aj rôzne paradigmy ako funkcionálne, objektovo-orientované, alebo aspektovo-orientované programovanie. Na ďalšej úrovni abstrakcie leží napríklad koncept generatívneho programovania, ktorý je súčasťou **metaprogramovania**.

Cieľom metaprogramovania je vytvoriť taký program, takzvaný **generátor**, ktorý na základe metadát bude použitý pre vytváranie iného programu. Metadáta môžu byť reprezentované metamodelom, čo bude v našom prípade **doménový model** webovej aplikácie. Od tohto metamodelu sa odvíjajú dôležité komponenty serverovej časti aplikácie ako rozhranie pre vykonávanie akcií s dátami, služby ktoré umožňujú manipuláciu s dátami v dátovej vrstve, a samotná reprezentácia modelov v cieľovom jazyku. V prezentačnej vrstve sú to taktiež samotné reprezentácie modelov, služby pre volanie akcií rozhrania serverovej časti, a čiastočne aj grafické

komponenty používateľského rozhrania ako tabuľky, formuláre a prvky výberu.

Webová aplikácia však stále obsahuje aj **špecifický** kód, ktorý vyplýva z konkrétnych potrieb a charakteru projektu. Z tohto dôvodu musí existovať spôsob ako zachovať môžnosť rozšíriteľnosti takéhoto vygenerovaného programu špecifickým kódom používateľa. Pre definovanie metamodelu využijeme **doménovo-špecifický jazyk**, ktorým bude možné stručne popísať aplikáciu. Okrem dátového modelu tento jazyk bude umožňovať definovanie **bezpečnostných** pravidiel prístupu na základe rolí a atribútov.

V tejto práci sa pokúsime zistiť, či sa takýto prístup k tvorbe webových aplikácií ukáže ako efektívny a použiteľný bez obmedzení v ich funkcionalite. Budeme pri tom vychádzať z **existujúcej** práce, ktorá sa zaoberala práve vytvorením takéhoto DSL, pričom naším cieľom bude **rozšíriť** jeho existujúcu funkcionalitu a zvýšiť jeho **použiteľnosť**.

1.2 Formulácia úlohy

Pred začatím tejto práce preskúmame existujúce riešenie, na ktoré budeme nadväzovať. Budeme analyzovať **aktuálny stav** existujúceho jazyka BFI pre definíciu modelu a jeho vyjadrovaciu silu. Na základe tejto analýzy **navrhneme úpravy** a najmä rozšírime jazyk o ďalšie prvky, ktoré vývojári najčastejšie potrebujú. Ďalej zhodnotíme obmedzenia fungovania systému BFI ako celku z pohľadu používateľa rámca, ale aj vývojára, ktorý môže potencionálne rozširovať ekosystém o nové generátory. Preveríme kvalitu existujúcej implementácie a podrobíme ju v prípade potreby refaktorizácii. Preskúmame aktuálne využívané **sekundárne systémy** súvisiace s vývojom a nasadením aplikácie, a nájdeme spôsoby ako pre nich poskytnúť podporu prostredníctvom nášho softvérového rámca. Navrhneme a vyvinieme **podporné nástroje** pre vývoj a pohodlné používanie nášho rámca. To zahŕňa podporu editovania zdrojových súborov rámca, zrozumiteľné hlásenie chýb, komfortné spúšťanie generátorov, a jednoduchú distribúciu nášho systému smerom k používateľom.

2 Predošlá práca

Táto práca nadväzuje na prácu „Doménovo-špecifický jazyk pre vývoj webových aplikácií“ [1], ktorá sa zaoberala generovaním klientskej a serverovej časti webovej aplikácie na základe dátového modelu.

2.1 Doménovo-špecifický jazyk

Pre vyjadrenie tohto modelu bol vytvorený nový doménovo-špecifický jazyk s názvom BFI, čo je skratka pre „Backend Frontend Interface“. V tomto jazyku je možné definovať štruktúry a vzťahy entít, ako aj ich vlastností. Každá entita je zložená z atribútov, ktoré majú určité vlastnosti. Medzi tieto vlastnosti patrí meno, typ a popis atribútu, ako aj typ editoru ktorý je využívaný pri generovaní pohľadov. Gramatika tohto jazyka 2.1 bola vyjadrená pomocou nástroja Antlr. Tento nástroj obsahuje vlastný špeciálny jazyk používaný na špecifikovanie bezkontextovej gramatiky vo forme EBNF („Rozšírená Backus-Naurova forma“).

```
1  grammar Model;
2
3  models: model+;
4  model: MODELNAME fields+;
5
6  fields : name':'type  (fieldoptions*);
7  fieldoptions :  ('label=' label)|('editor=' editor)|virtual;
8
9  name : ID;
10 editor : 'textarea'|'string'|'select'|'multiselect'|
11 'password'|'date';
12 relation : MODELNAME;
```

```
13 label : STRING;
14 type : 'int'|'string'|'text'|'1:1' '(' relation ')'|
15 '1:N' '(' relation ')'|'N:M' '(' relation ')';
16 virtual: 'virtual';
17
18 MODELNAME  : [A-Z][a-zA-Z]+ ;
19 ID  : [a-z]+ ;
20 STRING : '"' .*? '"';
21 WS : [ \r\t\n]+ -> skip;
```

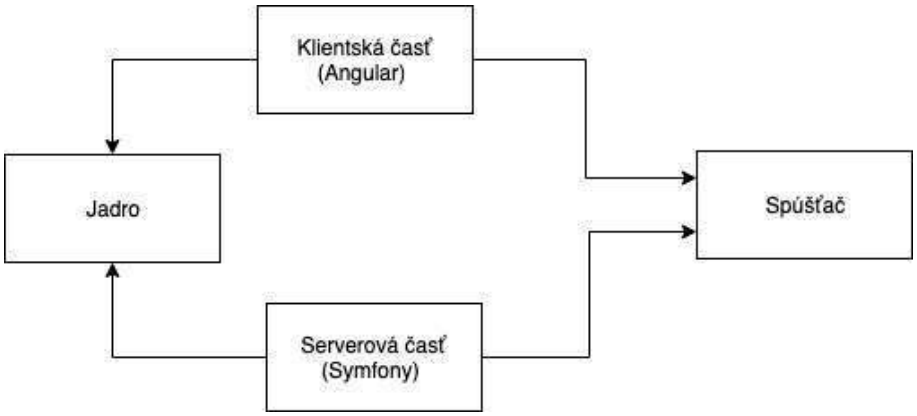
Zdrk. 2.1: BFI - definícia gramatiky pomocou Antlr

2.2 Generovanie serverovej a klientskej časti aplikácie

Účelom rámca je vyprodukovanie zdrojového kódu na základe definovaného modelu. Architektúra rámca BFI 2.1 pozostáva zo štyroch častí. Jadro je tvorené syntaktickým analyzátorom jazyka BFI vygenerovaným pomocou nástroja Antlr a nástroje pre implementáciu generátorov. Generátor zodpovedá za vytvorenie kódu v cieľovom jazyku pre ktorý je daný konkrétny generátor určený. Rámec obsahuje presne 2 druhy generátorov pre serverovú a klientskú časť webovej aplikácie. Načítanie modelu a spustenie týchto generátorov na základe argumentov zaobstaráva spúšťač.

Jednou z vlastností architektúry je možnosť ľubovoľne zamieňať generátory podľa požiadavok projektu pre použitie špecifických rámcov.

Pre zachovanie možnosti kód ďalej upravovať autor vo veľkej miere využíva dedičnost. Jednotlivé komponenty, respektíve triedy, sú teda rozdelené na tie, ktoré obsahujú generovaný kód a tie, ktoré obsahujú špecifický kód používateľa. Serverová a klientská časť aplikácie komunikujú prostredníctvom architektonického štýlu REST.



Obr. 2.1: Architektúra generátora jazyka BFI

2.3 Využívanie šablón pre generovanie pohľadov

Pre klientskú časť aplikácie je možné využívať HTML šablóny, pričom autor pre tento účel zaviedol špeciálne XML značky. V procese generovania generátor vyhľadá všetky šablóny (súbory s príponou *.bfi.html*) a následne vygeneruje HTML kód s nahradenými špeciálnymi značkami.

2.3.1 Značky na úrovni atribútov modelov

Nasledujúce špeciálne XML značky sa viažu na niektorý z atribútov modelu špecifikovaného pomocou BFI. Patria sem:

- **bfi-input** element, ktorý prijíma **vstup** podľa špecifikovaného editora pre daný atribút objektu,
- **bfi-output** element pre **zobrazovanie** hodnoty danej vlastnosti,
- **bfi-label** element pre zobrazenie **popisu** uvedeného vo vlastnosti label daného objektu.

Keďže sú tieto značky sú viazané na atribúty modelov, vyžadujú tieto atribúty:

- **bfi-model** názov modelu objektu,
- **bfi-field** názov atribútu objektu.

2.3.2 Zobrazovanie tabuľky

Pre zobrazenie tabuľkovej reprezentácie dát jednotlivých objektov je možné využiť značku **bfi-tb**. Je možné ju používať analogicky so značkou **table** jazyka HTML a poskytuje teda nasledujúce alternatívne značky.

- **bfi-tb-header** zaobaľuje elementy definujúce hlavičku tabuľky,
- **bfi-tb-body** zaobaľuje elementy definujúce telo tabuľky
- **bfi-tb-th** zaobaľuje elementy hlavičky tabuľky
- **bfi-tb-tr** zaobaľuje elementy riadku tela tabuľky
- **bfi-tb-td** definuje dátovú bunku, prípadne jej názov v prípade použitia v hlavičke (využíva parametre **bfi-model** a **bfi-field**)

Použitie môžeme vidieť na príklade A.26.

```
<bfi-tb>
    <bfi-tb-header>
        <bfi-tb-th>
            <bfi-tb-td bfi-model="Task" bfi-field="name"></bfi-tb-td
            <bfi-tb-td bfi-model="Task" bfi-field="state"></bfi-tb-td>
            <bfi-tb-td>Actions</bfi-tb-td>
        </bfi-tb-th>
    </bfi-tb-header>
    <bfi-tb-body>
        <bfi-tb-tr>
            <bfi-tb-td bfi-model="Task" bfi-field="name"></bfi-tb-td>
            <bfi-tb-td>{{task.state}}</bfi-tb-td>
            <bfi-tb-td>
                <button (click)="showTaskReader(task)">Show</button>
            </bfi-tb-td>
        </bfi-tb-tr>
    </bfi-tb-body>
</bfi-tb>
```

Zdrk. 2.2: Príklad vytvorenia BFI tabuľky

2.4 Editor pre jazyk BFI

Autor vyvinul editor pre podporu vývoja v tomto jazyku v nástroji JetBrains MPS. Tento editor bolo možné použiť ako rozšírenie do prostredí Jetbrains, avšak bolo potrebné mať nainštalované rozšírenie **MPS core**. Výsledné rozšírenie je vo svojej podstate kompromis medzi textovým a formulárovým editorom.

Obr. 2.2: MPS BFI editor

Ide teda o neštandardný typ editora, čo so sebou prináša určité nevýhody. Používatelia nie sú zvyknutí na jeho používanie čo ich môže odradiť od používania nášho rámca. Aj napriek tomu, že nástroj umožňuje import a export textového formátu zdrojového súboru, tie sú pôvodne uložené vo formáte XML, ktorý je nečitateľný pre iné nástroje. Výhodou je jednoduchosť implementácie na rozdiel od iných špecializovaných nástrojov, respektíve knižníc na tvorbu editorov a rozšírení pre vývojové prostredia.

2.5 Zhodnotenie predošlej práce

Výstupom práce bol **prototyp** kde autor implementoval túto architektúru v jazyku Java, pričom vytvoril generátory pre rámec **Angular** pre klientskú časť a rámca **Symphony** pre serverovú časť.

Účelom prototypu bolo otestovanie konceptu riešenia na vzorke aktívnych webových vývojárov. Toto testovanie pozostávalo z predstavenia fungovania rámca a zadania úlohy. Cieľom teda **nebolo** vyvinúť hotový produkt, ale jeho návrh a otestovanie použiteľnosti. V tejto práci je teda priestor pre vytvorenie **univerzálneho** riešenia, ktoré bude podporovať všetky funkcionality deklarované v návrhu a bude možné ho použiť pre akýkoľvek projekt. Produkt by mal tiež sledovať najnovšie, avšak ustálené **trendy** v oblasti vývoja webových aplikácií, k čomu je potrebná ďalšia analýza. Jednou z požiadaviek BFI pre rámec obslužnej časti apli-

kácie je podpora pre automatické generovanie **databázovej** štruktúry, čo môže byť obmedzujúce. Riešenie tiež **používateľa** obmedzuje pre použitie práve dvoch úzko špecializovaných generátorov, pričom daný model ktorý je pomocou jazyka BFI definovaný môže dosahovať širšie využitie.

3 Analýza

Jedným z cieľov rámca BFI je dostatočná **univerzálnosť**, teda schopnosť poskytnúť zázemie pre implementovanie generátora pre ľubovoľný softvérový rámec, či jazyk. To bolo splnené už v predošlej práci, avšak bola stanovená požiadavka podpory **MVC architektúry** pre rámec prezentačnej vrstvy. Preto znovu zanalyzujeme nevyhnutnosť tejto požiadavky.

Aby mal rámec BFI potenciál uspieť oproti iným generátorom analyzovaným v predošlej práci, mal by vývojárom ponúknuť podporu pre automatizovanie tých činností, ktoré iné generátory neponúkajú, preto sa v prevažnej časti analýzy budeme zaoberať potrebami vývojárov a **aktuálnymi trendami** v tvorbe webových aplikácií, a ako tieto potreby odraziť v jazyku BFI. Preskúmame tiež riešenie pre odstránenie obmedzenia v podobe požiadavky na serverový rámec podporovať vytváranie databázovej štruktúry.

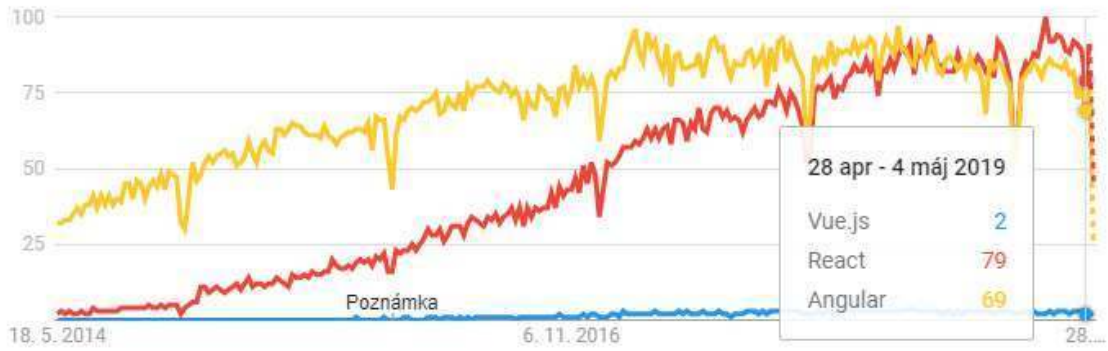
3.1 Moderné webové technológie pre vývoj používateľského rozhrania

V tejto podkapitole poskytneme prehľad v súčasnosti najpoužívanejších technológií pre vývoj prezentačnej vrstvy webových aplikácií, čo zahŕňa predstavenie ich základných konceptov. Pomôže zamyslieť sa nad možnými komplikáciami, ktoré by mohli nastať pri implementovaní generátorov BFI. Ako hlavná požiadavka pre podporu webového rámca bola stanovená podpora architektúry MVC („Model View Controller“).

Túto architektúru predstavil Trivet Reenskaug už v roku 1980 [2] a patrí teda medzi staršie, ale stále veľmi populárne architektonické vzory. Názov reprezentuje 3 súčasti:

- **Model** uchovávanie dát a stavu aplikácie, rozhranie s úložiskom
- **View** (pohľad) používateľské rozhranie zobrazujúce dáta
- **Controller** (ovládač) zapezpečenie interakcie prvkov pohľadu, komunikácia so serverom

Medzi najpopulárnejšie JS technológie v súčasnosti patria Angular, Vue.js, a React. Pre porovnanie ich popularity môžeme použiť štatistické dáta hľadaných výrazov dostupných prostredníctvom analytické platformy Google Trends [3]. Podľa grafu 3.1 sa najpopulárnejšou technológiou prednedávnom stala knižnica React.



Obr. 3.1: Porovnanie počtu hľadaní JS rámcov **Zdroj:** [3]

O niečo miernejší rozdiel zobrazuje výkaz [4]. V tabuľke 3.1 môžeme vidieť, že vysoký dopyt po znalosti Angular je takmer nasýtený, zatiaľ čo zvyšné dva rámce majú vačší potenciál rásť.

Technológia	Žiadanosť	Znalosť	Iniciatíva
Angular	38%	34%	27%
React	38%	26%	33%
Vue.js	19%	11%	25%

Tabuľka 3.1: Štatistika popularity JS rámcov **Zdroj:** [4]

Pri porovnaní aktivity v oficiálnych verejných repozitároch daných technológií v službe Github (tabuľka 3.2) v obľúbenosti vedie Vue.js.

Podobný trend si všíma aj prieskum spoločnosti Jetbrains [5], ktorý vníma každoročný nárast používateľov, ktorí používajú dané softvérové rámce na pravidelnej báze. Vue.js sa umiestnil ako druhý najpoužívanejší rámec (39%) veľmi tesne za

Technológia	Sledovaní	Hviezd	Vetvení
Angular	3300	48,704	13,066
React	6621	130,294	23,990
Vue.js	5895	140,134	20,131

Tabuľka 3.2: Porovnanie Github aktivity pre JS rámce (2019)

serverovým rámcom Express (40%), pričom rovnako ako v predošlých prípadoch je na prvom mieste React (54%).

Podarilo sa nám teda identifikovať najpopulárnejšie technológie, čo zhodnoe potvrdzujú všetky menované zdroje.

3.1.1 Angular

Angular je rámec vyvíjaný spoločnosťou Google, ktorý vznikol v roku 2010. Od verzie 2 využíva jazyk TypeScript, čo je rozšírenie jazyku JavaScript podporujúce typy. Jeden komponent ma vždy práve jednu šablónu a má na starosti aktualizovanie pohľadu, rovnako ako aj spracovanie akcie používateľa.

```
1  import { Component } from '@angular/core';
2  @Component({
3      selector: 'app-root',
4      templateUrl: './app.component.html',
5      styleUrls: ['./app.component.css']
6  })
7  export class AppComponent {
8      title: string = 'Spring Boot - Angular Application';
9      constructor() {
10     }
11 }
```

Zdrk. 3.3: Základný Angular komponent **Zdroj:** [6]

Na ukážke kódu 3.3 môžme vidieť prázdny komponent. Ten sa skladá z dekorátora *@Component*, ktorý zahŕňa cestu ku šablóne v HTML, a cestu ku štýlu v CSS. V konštruktore tohto komponentu následne nastavíme premennej *title* hodnotu. Ak sa v priradenej šablóne nachádza zástupný reťazec *{{ title }}*, jeho hodnota sa

nahradí podľa kontextu.

3.1.2 React JS

React je na rozdiel od Angular knižnica, teda zatiaľ čo Angular je nadradený aplikácii, ktorej diktuje štruktúru a následne ju používa, React je knižnica ktorá nám poskytuje stavebné prvky pre vývoj [7]. O React je v prvom rade potrebné poznamenať, že sa nedrží modelu MV*, ale celú svoju funkcionalitu spája v komponente „View“, čiže pohľade [8]. To narúša jednu z požiadaviek, ktorú zadala prvotná práca, čo však nebude musieť platiť v novom návrhu. Využíva koncept Virtual DOM, teda zobrazovaný HTML dokument je len reprezentáciou virtuálneho, s ktorým pracujeme. Hlavnou úlohou tejto koncepcie je poskytnutie vyššej úrovne abstrakcie pri vytváraní komponentov a zrýchlenie aplikovania zmien vnútorných stavov komponentov.

```
1  import React, { Component } from 'react';
2  import './App.css';
3  export default class App extends Component {
4      constructor(props) {
5          super(props);
6          this.itemService = new ItemService();
7      }
8      componentDidMount() {
9          this.itemService.retrieveItems().then(items => {
10             this.setState({items: items});
11         });
12     }
13     render() {
14         const listItems = items.map((item) =>
15             <li key={item.link} onClick={() => this.onSelect(item.link)}>
16                 <span className="item-name">{item.name}</span> |
                   ↪ {item.summary}
17             </li>
18         );
19         return ( <ul className="items"> {listItems} </ul> )
20     }
```

```
21      // ...
22  }
```

Zdrk. 3.4: Ukážkový React komponent **Zdroj:** [9]

V ukážke 3.4 vidíme základnú konštrukciu React komponentu, ktorý predstavuje pohľad na zoznam predmetov. V komponente je spojené zobrazovanie údajov vo funkcii *render()* a vkladanie údajov do tohto rozhrania v metóde *componentDidMount()*. Vo funkcii *render()* môžeme používať ďalšie komponenty, ktorým môžeme posunúť aktuálny stav a vytvoriť takto hierarchiu komponentov. Ak je to nutné, pri zmene stavu komponentu sa funkciou render aktualizuje pohľad. V tomto príklade môžeme vidieť aj triedu *ItemService*, ktorá nám môže nahradiť kontrolér v otázke generovania rozhrania s naším serverom.

```
1   export default class ItemService {
2       async retrieveItems() {
3           return fetch('http://localhost:8080/collectionItems')
4               .then(response => {
5                   if (!response.ok) {
6                       this.handleResponseError(response);
7                   }
8                   return response.json();
9           });
10      }
11      //... other methods
12  }
```

Zdrk. 3.5: Ukážkový React servis **Zdroj:** [9]

Servis je teda jednoduchá trieda bez stavu, ktorá nám poskytuje metódy reprezentujúce CRUD operácie.

3.1.3 Vue JS

Rámec Vue.js je v mnohých ohľadoch podobný s knižnicou React. Skladá sa z komponentov, ktoré kombinujú všetky tri vrstvy MVC do jednej. Z toho vyplýva že rovnako nespĺňa požiadavku pre podporu MVC architektúry, avšak potrebnú vrstvu ovládača vieme nahradiť servisom. Tak isto využíva koncept Virtual DOM,

avšak z porovnávaných technológií je najpomalší [10]. Vue je vhodný pre menej rozsiahle webové aplikácie kde budú výkonnostné nedostatky zanedbateľné. Vďaka svojej jednoduchosti umožňuje rýchle prototypovanie a vývoj.

```
<template>
    <table>
        <tbody>
            <tr v-for="record in records" v-bind:key="record.id">
                <td>{{ record.name }}</td>
                <td>{{ record.value }}</td>
            </tr>
        </tbody>
    </table>
</template>
<script>
import api from '@/FoodRecordsApiService';
export default {
    data() { return { records: []} },
    async created() { this.records = await api.getAll() }
}
</script>
```

Zdrk. 3.6: Ukážkový Vue.js komponent **Zdroj:** [11]

Na ukážke 3.6 máme jednoduchý Vue komponent. Skladá sa zo šablóny v HTML, kde využívame takzvané direktívy, čo sú špeciálne HTML atribúty. Direktívy Vue sa označujú s predponou *v-* a bližšie nám špecifikujú ako sa má náš model vložiť do šablóny. V tomto prípade využívame direktívu *v-for* pre vytvorenie elementu <*tr*> pre všetky položky zo zoznamu, pričom s *v-bind* určujeme primárny kľúč daných elementov, ktorý nám neskôr uľahčí prácu s týmito elementmi a zároveň umožní Vue prekresľovať zmenené záznamy vďaka Virtual DOM.

```
import axios from 'axios'
const client = axios.create({
    baseURL: 'http://localhost:5000/api/FoodRecords'
})
export default {
```

```
6       async execute(method, resource, data) {
7           return client({
8               method,
9               url: resource,
10              data
11          }).then( req => {return req.data} )
12      },
13      getAll() { return this.execute('get', '/') },
14      //... other methods
15  }
```

Zdrk. 3.7: Ukážkový Vue.js servis **Zdroj:** [11]

Vue v sebe nezahŕňa HTTP klienta a musíme teda použiť externú knižnicu, v tomto prípade *axios*.

3.1.4 Zhrnutie

Všetky skúmané rámce a knižnica napriek tomu, že vo svojej podstate nie sú navrhnuté podľa architektúry MVC, umožňujú jednoduché oddelenie **generovaného** a **špecifického** kódu. Sú teda vhodné pre použitie s naším rámcom BFI. V pôvodnej práci nebola spomenutá dôležitá vlastnosť analyzovaných technológií, a to možnosť **dedenia**. V prípade, že používateľ potrebuje doplniť funkcionalitu do API komponentu, vytvorí potomka vygenerovaného komponentu a zachová možnosť jednoduchého aktualizovania tohto komponentu generátorom pri zmene dátového modelu. Všetky analyzované technológie podporovali dedičnosť pomocou *extends*.

3.2 Alternatívne generovanie a evolúcia databázovej štruktúry

Náš rámec sa odvíja od definície dátového modelu a od toho je závislá najmä štruktúra **databázy**. Niektoré softvérové rámce ponúkajú možnosť automatického vytvárania a aktualizovania tejto štruktúry. V prípade, že použijeme technológiu, ktorej tejto vlastnosť chýba, databázovú štruktúru budeme musieť vytvoriť ma-

nuálne. Dátovú štruktúru sme už definovali pomocou jazyka BFI, a teda nielenže vykonávame opakovane činnosť, ktorá môže byť automatizovaná, ale vytvárame priestor pre inkonzistenciu reálnej a očakávanej databázovej štruktúry webovou aplikáciou. To sa prejaví nielen pri vytváraní, ale aj pri následnej **evolúcii** dátového modelu aplikácie.

V produkčnom prostredí sú dáta a ich štruktúra najcitlivejšou časťou biznis hodnoty. Preto je rizikové povoliť aplikácii automaticky upravovať štruktúru dát bez dodatočného overenia týchto zmien pred ich aplikovaním. Vhodnou metódou je vygenerovanie takzvaných **migračných skriptov**, teda skriptov slúžiacich na pozmenenie štruktúry databázy. Každé vykonanie takéhoto skriptu tiež zvyšuje verziu databázy, ktorá môže byť reprezentovaná napríklad časovým údajom poslednej zmeny.

Od nástroja pre migrovanie štruktúry databázy očakávame získanie aktuálnej štruktúry a **porovnanie** so želanou štruktúrou [12]. Ak je nutná zmena, nástroj vygeneruje migračný skript a prípadne ho aj sám aplikuje. V našom prípade budeme chcieť tento proces rozdeliť tak, aby sme pred jeho aplikovaním mohli zhodnotiť dopad na integritu dát a na prestoj databázy počas aplikovania zmeny.

Jednou z možností je vytvorenie vlastného takéhoto migračného nástroja. Vzhľadom na rozdielne SQL dialekty nie je táto implementácia triviálna a budeme teda hľadať možnosť prepojenia s **existujúcimi** riešeniami.

3.2.1 Flyway

Flyway je nástroj pre automatické **aplikovanie** migračných skriptov nad viacerými databázami na základe ich verzie. Rozlišuje opakovateľné (zmeny procedúr, funkcií, spúšťačov) a jednorázové migrácie (zmeny tabuliek, cudzích kľúčov), ktoré udávajú verziu databázy. Na základe tejto verzie je rozhodnuté, ktoré migračné skripty majú byť spustené pre danú databázu. Ponúka tiež možnosť definovania migrácií pomocou jazyku Java a zásuvného modulu pre Maven. Tie sú využiteľné pre zmeny, ktoré nie je možné, alebo jednoduché vyjadriť prostredníctvom SQL. Flyway navyše archivuje históriu zmien spolu s používateľom ktorý ich vykonal v dedikovanej databázovej tabuľke v cieľovej databáze.

3.2.2 LiquiBase + Hibernate

LiquidBase je podobne ako Flyway nástroj pre úpravu databázovej štruktúry. Migračné skripty je možné vyjadrovať aj pomocou takzvaných *ChangeSet* vo formáte XML. Na rozdiel od Flyway ale dokáže **generovať** rozdielové SQL skripty, ktoré vyplynú z **porovnania** dvoch databáz. V kombinácii s Hibernate však dokáže ako jednu z týchto štruktúr pre porovnanie použiť Java **triedy** oanotované ako *@Entity*. Náš BFI rámec by teda musel na základe BFI modelu vygenerovať Hibernate entity a následne použiť LiquidBase. Pre vygenerovanie tohto rozdielového migračného skriptu je možné použiť balíček pre Maven a príkaz *mvn liquibase:diff*.

3.3 Riadenie prístupu k zdrojom

Jednou z významných súčastí webových aplikácií je implementácia kontroly poverení prístupu k rôznym zdrojom a akciám. Potrebujeme nájsť univerzálny spôsob ako čo najjednoduchšie priamo v modeli BFI vyjadriť **privilégiá** používateľov. Medzi najpoužívanejšie modely riadenia prístupu patrí IBAC (na základe identity), RBAC (na základe rolí) a ABAC (na základe atribútov).

IBAC V tomto modeli sú oprávnenia asociované s **identifikátormi** subjektu, napríklad používateľským menom. Prístup je udelený len ak takáto asociácia so zdrojom existuje [13]. ACL je najbežnejšou a najjednoduchšou implementáciou IBAC, ktorý je využitý aj v operačných systémoch MS Windows 2000, 2003, XP, Linux a FreeBSD [14]. Ku každému objektu definuje zoznam autorizovaných entít a ich povolených akcii pomocou matice prístupov. Tento prístup označujeme ako zdrojovo-orientovaný.

Object	
File_a	.*: r; Group.group1: rx
File_b	User. Kathy: rwx
File_c	Group.group1: rx; User. Janet: - - -; *.*: r

Tabuľka 3.3: Zjednodušený príklad ACL **Zdroj:** [14]

RBAC Riadenie prístupu na základe **rolí** obmedzuje prístup k zdrojom na základe biznis funkcie alebo roly. Zavádza úroveň abstrakcie, ktorá šetrí administratívu vďaka tomu, že privilégia už nemusia byť opakovane menované [15].

ABAC Riadenie prístupu na základe **atribútov** definuje povolenia na základe relevantných bezpečnostných charakteristík. Existujú atribúty:

- subjektu - atribúty entity, ktorá vykonáva akciu na zdrojoch, napríklad identifikátor, organizácia, pracovná pozícia a podobne,
- zdroja - atribúty zabezpečenej entity na ktorej chceme vykonať akciu, napríklad nadpis, predmet, dátum, autor, vlastník, taxonómia a podobne,
- prostredia - atribúty kontextu v akom sa akcia vykonáva, napríklad sieť.

Pretože rolu môžeme považovať za atribút, ABAC teda kombinuje modely IBAC (na základe identity) a RBAC. Naše politiky teda dokážeme špecifikovať detailnejšie ako v prípade RBAC. Politika je definovaná pravdivostnou funkciou, ktorej argumenty sú množiny troch uvedených typov atribútov [13].

Informačné systémy sú dnes väčšinou používané viacerými používateľmi, ktorí majú obmedzený prístup len k **vlastným** zdrojom. Rovnako existujú administratívne **úrovne** zodpovedností, ktoré uľahčujú administratívu poverení pre rôzne časti systému. V našom prípade teda bude vhodná podpora kontroly prístupu na základe identity, aj na základe rolí, ktorá je kombinovaná v modeli ABAC. Prihlasovanie predstavuje potenciálnu zraniteľnosť a preto si vyžaduje dôkladnú implementáciu. Existuje však viacero externých poskytovateľov autentifikácie a autorizácie, ktoré sú široko využívané a dobre otestované.

Tvorcovia webových aplikácií sa nezriedka pokúšajú tvoriť vlastné správy prístupu, ktoré majú vo výsledku ťažkosti s riešením týchto výziev. Medzi tieto ťažkosti patrí obmedzená funkčnosť, slabá integrácia s aplikáciami tretích strán, zlá používateľská skúsenosť a najmä slabá bezpečnosť [16]. Budeme sa teda spoliehať predovšetkým na **hotové** riešenia, takzvané PaaS („platforma ako služba“). Tieto systémy často podporujú aj autentifikáciu prostredníctvom Google, alebo Facebook účtov, čo prináša používateľovi istú mieru komfortu.

Cognito je služba zo skupiny *Amazon Web services*, ktorá ponúka systém registrácie aj prihlasovania pomocou týchto platform, ale aj pomocou emailovej adresy.

Ponúka knižnice pre rôzne jazyky a platformy a možnosť prihlásenia napríklad pomocou *Facebook Connect*, či *Google Sign-in*. Táto služba je **zdarma** do 50,000 aktívnych používateľov mesačne a tým šetrí náklady na vývoj pre malé webové aplikácie. Cognito nám umožňuje používateľom pridávať **skupiny**, ktoré môžeme chápať ako roly, a neobmedzuje nás v pridávaní vlastných **atribútov** pre jednotlivých používateľov. Pre detailnejšiu správu poverení jednotlivých rôl a prístup k iným službám v AWS je možné prepojiť tieto roly so službou IAM, ktorá slúži na spravovanie identity a prístupu k službám a zdrojom platformy AWS.

Ďalšou službou poskytujúcou správu poverení je **Okta**. Oproti spomenutému produktu AWS jej **bezplatná** varianta je dostupná len pre 7000 aktívnych používateľov mesačne. Oproti AWS registrácia nie je viazaná na číslo platobnej karty. Rozhranie je komplexnejšie a ponúka detailnejšie nastavenie obsahu vracaných autentifikačných reťazcov. Z pohľadu funkcionality je Okta bohatšia, avšak pre jednoduchú webovú aplikáciu sú obe služby postačujúce.

Toto sú dve príklady služieb štandardne používaných pre implementáciu autentifikácie, pričom alternatív je mnoho. Ich **integrácia** je obvykle jednoduchá, pretože máme k dispozícii vývojové balíčky (SDK), ktoré vyžadujú len minimálnu konfiguráciu pre ich správne fungovanie. Pre zachovanie univerzálnosti výber služby nechávame na používateľovi a budeme sa zaoberať **kontrolou** získaných privilégií pri prístupe k zdrojom v serverovej časti.

3.4 Pravidlá návrhu komunikačného rozhrania medzi klientom a serverom

Tento systém sa drží konceptu modulárnej architektúry, podľa ktorého by mali byť jednotlivé vygenerované komponenty vzájomne zameniteľné. Ak spolu komponenty potrebujú komunikovať, musíme zapezpečiť, aby si vzájomne rozumeli. To sa týka najmä rozhrania serverovej a klientskej časti webovej aplikácie. Prvou časťou je určenie presných pravidiel a vytvorenie **dohody**, ktorou sa budú vývojári riadiť. Aby sme zabezpečili vzájomnú kompatibilitu komponentov, je ďalej nutné čo najspoľahlivejšie zabezpečiť **dodržiavanie** tejto dohody.

K tomu je možné pristupovať rôzne.

- **Dokumentácia**

Požiadavky v textovej alebo grafickej forme sú prvou alternatívou, ktorá sa ale spolieha na zodpovednosť programátora v ich dodržiavaní.

- **Použitie dizajnovacích jazykov**

  Existujú jazyky pre návrh rozhraní, ktoré sú zamerané na znovu použiteľnosť tohto návrhu pre generovanie dokumentácie, testov, ale aj kódu v rôznych programovacích jazykoch a softvérových rámcoch.

- **Existujúce štandardy**

  Pre zaužívané štandardy existuje integrovaná podpora priamo v jazykoch, alebo formou knižníc, čo znižuje riziko inkonzistencií.

3.4.1 Technológie pre návrh rozhrania

V súčasnosti sú najpoužívanejšími technológiami pre výmenu správ medzi aplikáciami SOAP a REST.

SOAP bol dlho najpoužívanejší protokol v serverovej komunikácii v podnikovej sfére, a existuje teda mnoho nástrojov pre jeho podporu, ako napríklad WSDL, DTD, alebo SoapUI. Pre komunikáciu využíva jazyk XML.

REST je charakteristický menšou krivkou učenia a menším dátovým tokom [17]. Pre komunikáciu využíva jazyk JSON. Nevýhodou je nejasnosť konvencií pre jeho tvorbu, pretože je to architektonický štýl, nie protokol. Medzi jazyky pre modelovanie REST rozhrania patrí napríklad Swagger, alebo RAML.

Okrem toho existuje relatívne mladá technológia **GraphQL**, ktorá pre definovanie rozhrania a následné dopytovanie využíva vlastný doménovo-špecifický jazyk.

OpenAPI (Swagger)

OpenAPI je nástroj pre definovanie štruktúry REST rozhrania. Je ekvivalentom dokumentu WSDL, ktorý je používaný pre SOAP rozhrania. Pomocou dokumentu napísaného v YAML, alebo JSON vieme navrhnúť aké prístupové body budú v rozhraní existovať, informácie o ich parametroch, tele požiadaviek, hlavičiek, kódov odpovedí, potrebných prístupových rôl, štruktúru a kódy odpovedí a podobne. Na základe tejto definície je možné následne generovať kontroléry na strane servera a klientský kód pre komunikáciu s týmto serverom. Pozostáva z troch nástrojov.

- **Swagger editor** - slúži na vytvorenie Swagger dokumentu s definíciami štruktúry požiadaviek a odpovedí
- **Swagger UI** - vizualizuje a testuje REST API priamo vo webovom prehliadači
- **Swagger Codegen** - generuje SDK v rôznych programovacích jazykoch vrátane Java, Objective-C, PHP, alebo Python

RAML

RAML („Jazyk pre modelovanie REST API“) je rovnako ako OpenAPI nástrojom pre opis API, ktorý ponúka takmer totožné vlastnosti. Oproti OpenAPI umožňuje vkladanie obsahu z externých súborov a je vo všeobecnosti lepšie čitateľný, avšak ponúka menej dostupných generátorov a menej rozsiahlu dokumentáciu [18]. Ponúka vývojové prostredie API workbench.

HATEOAS

Medzi technológie postavené na princípoch REST patrí HATEOAS („Hypermédium ako základ stavu aplikácie“). Aj keď HATEOAS nie je štandardom, ale podobne ako REST len architektonickým štýlom, môže ísť o zaujímavú technológiu na zváženie. Podľa HATEOAS klient interaguje s aplikáciou dynamicky pomocou odkazov. Klient pristúpi k serverovej časti prostredníctvom fixnej URL adresy a ďalej je **navigovaný** k ostatným zdrojom a stránkam prostredníctvom **odkazov**. Klient vidí okrem dát aj povolené akcie nad danými dátami v aktuálnom stave. Každý vzťah agregácie a kompozície je vyjadrený odkazom a táto architektúra má veľký potenciál v oblasti škálovateľnosti, kde napriek tomu že vykonávame viac požiadaviek, tieto požiadavky môžu byť nezávisle spracované rôznymi inštanciami v potenciálne kratšom čase. To je výhodné najmä v cloudových prostrediach kde vieme škálovať rýchlo a lacno [19].

GraphQL

Ďalšou zaujímavou technológiou pre komunikáciu klient-server je GraphQL. GraphQL je jazyk dopytov, ktorý slúži ako alternatíva oproti prístupu k zdrojom pomocou ciest a základných akcií CRUD v prípade REST. Server obsahuje GraphQL **schému**, v ktorej sú definované typy, teda objekty a ich štruktúry, a operácie pre čítanie („query“) a zápis („mutation“). Klient teda dokáže prehľadávať túto schému

na čo mu môže slúžiť aj webové rozhranie GraphiQL. Výhodou tohto prístupu sú preddefinované pravidlá pre podobu rozhrania.

```
...
/user/{id}:
  operationId: getUserById
  get:
    responses:
      ...
      links:
        EmployerCompany:
          operationId: getCompanyById
          parameters:
            companyName:
              $request.body#/employerId
```

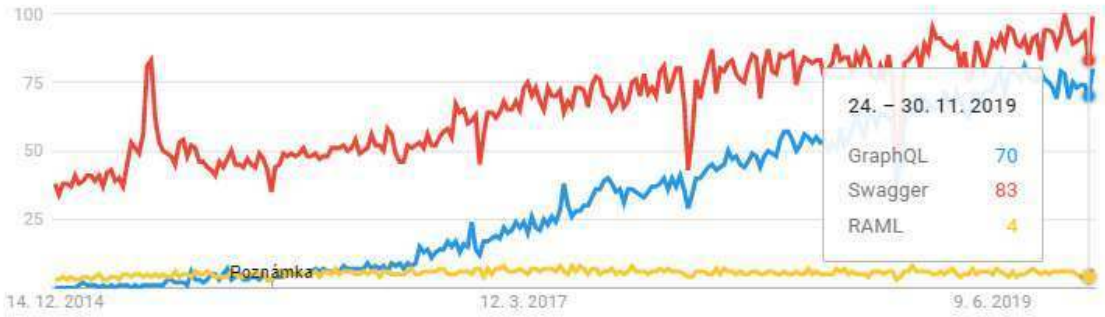
(a) OAS link definition in YAML.

```
query {
  user (id: "erik") {
    employerId
    employerCompany {
      companyName
    }
  }
}
```

(b) GraphQL query using link.

Obr. 3.2: Porovnanie OpenAPI špecifikácie a GraphQL schémy **Zdroj:** [20]

Na porovnaní 3.2 vidíme, že GraphQL je o niečo **stručnejší** v porovnaní s OpenAPI špecifikáciou (Swagger). Rovnako ako OAS ponúka podporu **generovania** kódu v rôznych technológiách a jazykoch ako Angular, React, Node.js, Java, alebo JSON schema.



Obr. 3.3: Porovnanie popularity rozhraní **Zdroj:** [21]

V grafe 3.3 je vidieť zjavne rastúcu tendenciu jeho popularity, ktorá čoskoro môže prerásť ustálený Swagger. S podporou jazykov vo forme 35 knižníc v čase

písania práce, vývojových prostredí, a služieb AWS AppSync a Prisma sa môže čoskoro stať silným štandardom pre návrh aplikačných rozhraní.

3.4.2 Zhrnutie a ustanovenie pravidiel

V tejto kapitole sme sa pokúšali nájsť spôsob, akým vytvoriť a vynútiť dohodu, podľa ktorej budú vygenerované aplikácie komunikovať. Zhrnuli sme si niekoľko zaužívaných technológií, ktoré sme stručne popísali a porovnali ich popularitu. Všetky analyzované nástroje pre opis rozhraní OpenAPI, RAML a GraphQL sú postačujúce pre naše potreby. Rozhodli sme sa použiť technológiu **GraphQL**, pri ktorej zavážil najmä stručný, dobre čitateľný a **zrozumiteľný** opis, ktorý je čitateľný nielen pre aplikáciu ale aj pre človeka, ktorý nemá s danou technológiou skúsenosti. Oproti REST sa neobmedzuje na CRUD operácie, čo nám dáva väčšiu voľnosť v tvorení metód pre manipuláciu s dátami. Ako jediná technológia tiež umožňuje **dynamickú** odpoveď, čo môže výrazne šetriť množstvo prenášaných dát a tým aj dôležitý čas odozvy webového rozhrania.

3.5 Rozšírené funkcie

Implementácia funkčnej aplikácie je len jedna z častí vývojového cyklu. Dôležitou časťou sú **DevOps** („vývoj a operácie“) nástroje, ktoré uľahčujú spravovanie požiadaviek, testovanie, udržiavanie kvality produktu a jeho rýchlejšie dodanie. Ich využitie môže znížiť čas vývojového cyklu o 10 až 30% a znížiť náklady na prevádzku o 20% [22]. Medzi hlavné domény patrí automatizované nasadzovanie softvéru do prostredia, monitorovanie infraštruktúry, testovanie, škálovanie a znižovanie chybovosti prevádzky. Tieto nástroje si vyžadujú určitú **konfiguráciu**, ktorá je v závislosti od charakteru aplikácie často veľmi podobná, čo vytvára priestor pre jej generovanie.

3.5.1 Virtualizácia pomocou Docker

Virtualizačné technológie historicky vyplynuli z potreby prevádzkovania procesov ako spravovateľných **kontajnerov**. Ide o spúšťanie aplikácií v oddelených virtuálnych prostrediach v podobe samostatných operačných systémov. Tieto balíky operačného systému s aplikáciou sa nazývajú kontajnery a predstavujú časti sys-

tému pripravené na nasadenie. Najrozšírenejšou technológiou prístupu ku kontajnerizácii je Docker. Ten umožňuje [23]:

- izolovanie procesu - proces v jednom kontajneri neovplyvňuje proces v inom kontajneri,
- izolovanie zdrojov - používa *cgroups* a *namespace* („menný priestor“) koncept z Linux kontajnerov,
- izolovanie siete - využíva Linux kontajnery,
- izolovanie súborového systému - využíva Linux kontajnery,
- spravovanie životného cyklus - spravuje sa pomocou démona a príkazového riadku,
- spravovanie stavu kontajnera - ukladanie a načítanie stavu kontajnerov.

Docker kontajner je inštancia vytvorená z obrazu. Obraz je súbor zložený z viacerých **vrstiev**. Každá vrstva reprezentuje inštrukciu zapísanú v Dockerfile a je určená len na čítanie, s výnimkou poslednej. Tieto obrazy sú uložené v súkromných, alebo verejných **repozitároch**. Jedným z takýchto repozitárov je *Docker Hub*, kde sú dostupné aj oficiálne obrazy vytvorené samotnými vývojármi Docker [24], alebo overenými spoločnosťami. Každý používateľ má možnosť nahrávať vlastné obrazy do tohto repozitára ktoré môžu byť verejne dostupné, alebo za poplatok privátne. **Dockerfile** je súbor inštrukcií potrebný k vytvoreniu Docker obrazu zapísaný v doménovo-špecifickom jazyku. Pre vytvorenie obrazu pre Springboot aplikáciu môže vyzerať napríklad nasledovne 3.8.

```
1 FROM openjdk:13-alpine
2 RUN apk update && apk add bash
3 WORKDIR /app
4 COPY /target/bfi-demo.jar /app
5 EXPOSE 8080
6 CMD ["java", "-jar", "bfi-demo.jar"]
```

Zdrk. 3.8: Príklad Dockerfile pre SpringBoot aplikáciu

Prvá inštrukcia spravidla špecifikuje obraz z ktorého budeme vychádzať a ktorý môžeme nájsť napríklad v *Docker Hub* repozitári. Pre zjednodušenie práce s nasadeným kontajnerom je vhodné nainštalovať pohodlný interpreter príkazového riadku, v našom prípade *bash*. Vykonanie príkazu počas budovania obrazu je možné pomocou slova **RUN**. Inštrukcia **WORKDIR** nám nastaví pracovný adresár, v ktorom budú vykonávané nasledujúce inštrukcie, podobne ako príkaz *cd* v Unixe. Podobne ako pri manuálnom nasadzovaní presunieme našu skompilovanú aplikáciu do cieľového prostredia príkazom **COPY**. Slovo **EXPOSE** pridáva nápovedu pre osobu, ktorá bude nasadzovať našu aplikáciu o tom, na akom porte naša aplikácia prijíma komunikáciu. Poslednou inštrukciou je spravidla **CMD** (alebo **ENTRYPOINT**), ktorá uvádza príkaz, ktorý má byť vykonaný pri spustení kontajnera, a ktorá ako jediná môže byť prepísaná pri spustení kontajnera.

3.5.2 Orchestrácia klastra pomocou Kubernetes

Kubernetes je open-source systém pre automatizovanie nasadzovanie, škálovanie, a správu kontajnerizovaných aplikácií. Na základe konfigurácií v doménovošpecifickom jazyku nám umožňuje nastaviť jednotlivé zložky produkčného prostredia a nasadiť komponenty našej aplikácie. Menovito poskytuje:

- horizontálne škálovanie replikovaním kontainerov podľa definovaných pravidiel,
- umiestňovanie kontainerov na dostatočne veľké výpočtové jednotky klastra podľa požadovaných zdrojov aplikácie,
- samo-uzdravovanie v prípade zlyhaní vďaka používateľom definovaných monitorovacích pravidiel,
- správu tajných objektov ako heslá, certifikáty, kľúče, prihlasovacie údaje a podobne,
- pripájanie úložiska s integráciou internetových úložných systémov,
- automatický návrat zmien v prípade zistenia chyby,
- presmerovanie komunikácie a rozkladanie záťaže v prípade viacerých inštancií aplikácie.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bfi-demo-deployment
spec:
  selector:
    matchLabels:
      app: bfi-demo
  replicas: 2
  template:
    metadata:
      labels:
        app: bfi-demo
    spec:
      imagePullSecrets:
        - name: regcred
      containers:
        - name: bfi-demo
          image: git.kpi.fei.tuke.sk:4567/ab123xy/bfi-demo:1.0.0
          ports:
            - containerPort: 8080
```

Zdrk. 3.9: Príklad Kubernetes Deployment pre Web servis

Na príklade 3.9 vidíme objekt typu **Deployment**. Tento objekt slúži ako konfigurácia pre nasadenie aplikácie s názvom **bfi-demo**, ktorá sa vytvorí z obrazu umiestnenom v školskom repozitári s názvom **bfi-demo** a tagom **1.0.0**. Školský repozitár je privátny, a preto táto konfigurácia obsahuje referenciu na prístupové údaje s názvom **regcred**, čo je objekt typu *secrets*. Znovu musíme tiež špecifikovať port, na akom bude aplikácia počúvať príchodziu komunikáciu.

Kubernetes prostredie pre našu aplikáciu je zložené z rôznych takýchto objektov. Aby bola naša aplikácia viditeľná mimo nášho klastra, musíme definovať objekt typu **Service**, z ktorého obdržíme IP adresu, a ktorý vytvorí mapovanie verejného portu na interný port inštancie/inštancií našej aplikácie. Ak chceme presmerovať komunikácie na základe URL adresy, musíme definovať objekt typu *Ingress*. Kubernetes ponúka množstvo nástrojov, ktoré bežne potrebujeme na prevádzku

informačných systémov, databáz, webov, ETL nástrojov, aplikácií, proxy serverov, IOT infraštruktúr, FTP serverov a podobne. V našom prípade budeme generovať konfigurácie pre vyššie menované objekty, ktoré sú nevyhnutné pre prevádzku webovej aplikácie.

3.5.3 Podpora nepretržitého vývoja a integrácie

Nepretržitý vývoj a integrácia softvéru je praktika, počas ktorej sa práca vývojárov spája a nasadzuje v rýchlom tempe. Účelom tejto stratégie je **zrýchliť** vydávanie zmien, **skvalitniť** softvér, a zvýšiť produktivitu. Tento postup končí automatizovaným nasadením každej zmeny do produkčného prostredia [25]. Túto techniku obvykle podporujú verzionovacie systémy ako *Github*, *Gitlab*, alebo *Jenkins*, *Travis CI* a podobne. Implementácia prebieha obvykle pomocou konfiguračného súboru, respektíve **skriptu**. Jeho forma nie je štandardizovaná a každý systém teda používa vlastný formát. Popisuje kroky aké je nutné spraviť pre kompiláciu, zostavenie a nasadenie aplikácie, prípadne aj jej otestovanie a statickú analýzu. V našom prípade sa budeme zameriavať na architektúru kontainerov Docker a ich spravovania vnútri klastra Kubernetes. Tomuto zámeru budú prispôsobované aj jednotlivé kroky v konfiguračnom súbore.

3.6 Zhrnutie návrhu budúcej podoby rámca BFI

V prvej časti analýzy sme sa zamerali na obmedzenia, ktoré platili pre predošlú verziu práce, a či tieto obmedzenia stále platia alebo môžu byť odstránené. Prvou vytýčenou podmienku bola podmienka pre podporu webového rámca, a to aby podporoval **MVC** architektúru. Zhodnotili sme, že pre všetky tri najpopulárnejšie rámce platí, že kód je možné generovať a čo je najdôležitejšie, pohodlne oddeliť generovaný a špecifický kód pomocou injekcie závislostí (z angl. „dependency injection“) a/alebo dedičnosti.

Druhé obmedzenie sa týkalo serverového rámca, a to podpory generovania **databázovej** štruktúry. Pre odstránenie tohto obmedzenia sme predstavili možnosť využitia externej aplikácie, ktorá by slúžila na vytváranie migračných skriptov, ktoré sú zároveň bezpečnejším riešením pre použitie v produkčnom prostredí oproti automatickému aktualizovaniu databázy nad ktorým nemáme kontrolu. Toto riešenie však prináša podmienku na zvýšenie modularity rámca, pretože bu-

deme potrebovať použitie nového generátora, ktorý nemá charakter generátora webovej ani serverovej aplikácie. Bude teda potrebné upraviť samotné jadro, respektíve spúšťač generátorov pre podporu viacerých generátorov.

Ďalej sme si položili otázku, ako pri ďalšom vývoji rámca zabezpečiť kompatibilitu z pohľadu vzájomnej **komunikácie** medzi aplikáciami serverovej a klientskej časti, vygenerovaných rôznymi generátormi. Zvolili sme použitie technológie GraphQL, ktorá sa ukázala ako najuniverzálnejšia a jednoduchá pre pochopenie. Do jadra softvérového rámca budeme potrebovať implementovať generovanie definičných súborov rozhrania s použitím tejto technológie, pretože tento dokument bude využívaný rôznymi generátormi.

V poslednej časti sme sa venovali podpore **moderných** technológií, ktoré sa spájajú s vývojom softvérových riešení. Nejde o komplexnú podporu DevOps nástrojov, ale skôr o úzku podporu vybraných najpoužívanejších nástrojov, ktorá vo viacerých prípadoch nemusí byť dostatočná a mohla by byť v budúcnosti rozšírená. Cieľom je skôr otvoriť otázku do akej miery je možné generatívnym programovaním uľahčiť prácu vývojárom.

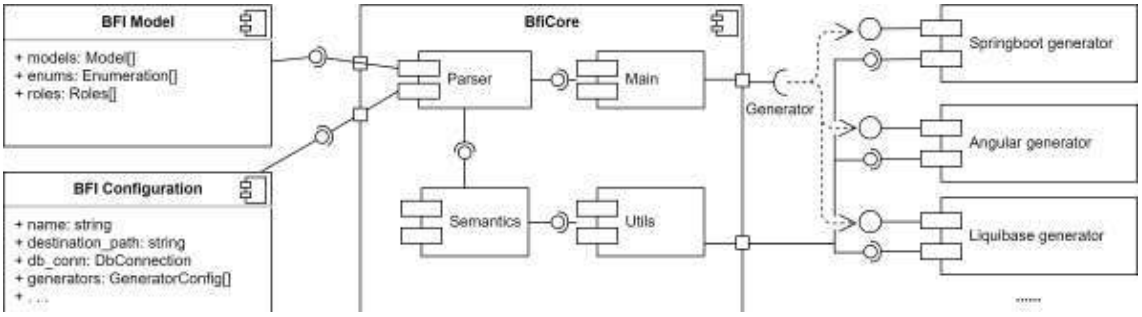
Tieto nové funkcionality budú vyžadovať rozšírenú konfiguráciu, ktorá v predošlej práci nebola potrebná. Bude teda potrebné implementovať podporu **konfiguračného** súboru, ktorý bude slúžiť pre zadanie atribútov projektu, daných funkcionalít, ale aj generátorov.

4 Návrh rozšírenia funkcionality BFI

Počas analýzy sme odhalili jednotlivé funkcionality, ktoré by mala moderná webová aplikácia poskytovať. Tie sa týkali nielen jej fungovania, ale aj celého jej životného cyklu ako nasadenie, prevádzka, dokumentácia a evolúcia. V tejto časti budeme postupne navrhovať novú verziu jazyka, fungovanie generátora a výsledného systému.

4.1 Úprava procesu generovania

Predošlé jadro umožňovalo použitie vždy dvoch generátorov pre klientskú a serverovú časť aplikácie. V našom prípade chceme používať ľubovoľný počet a druhy **generátorov**. Ich zadávanie pomocou argumentov spúšťania sa stáva nepraktické, a preto sme okrem vstupného modelu zaviedli aj **konfiguračný** súbor, ktorého štruktúra bude uvedená v kapitole.



Obr. 4.1: Komponentový UML diagram generátora

Jadro BFI je zložené z štyroch komponentov. Komponent **Parser** slúži na spracovanie vstupných súborov a definuje tiež množinu podporovaných formátov zdrojových súborov. Obsahuje syntaktický analyzátor súboru definujúceho modely a konfiguračného súboru. Pre BFI model je podporovaný len jeden formát kon-

krétnej syntaxe nášho jazyka definovanej pomocou Antlr. Pre konfiguračný súbor existujú dva podporované formáty YAML a JSON.

Modul **Semantics** obsahuje sémantickú reprezentáciu definovaného modelu a konfigurácie, ktorá je neskôr dostupná jednotlivým generátorom.

Modul **Main** je hlavným modulom, ktorý je zároveň rozhraním funkčnej časti rámca. Pomocou neho je možné spúšťanie jednotlivých generátorov, ktorým zabezpečí potrebný kontext pre ich beh. Main načíta cesty ku vstupným súborom a pomocou modulu Parser vytvorí ich sémantickú reprezentáciu. Následne nájde potrebné generátory, ktoré zadal používateľ, a spustí ich s danou konfiguráciou a modelom.

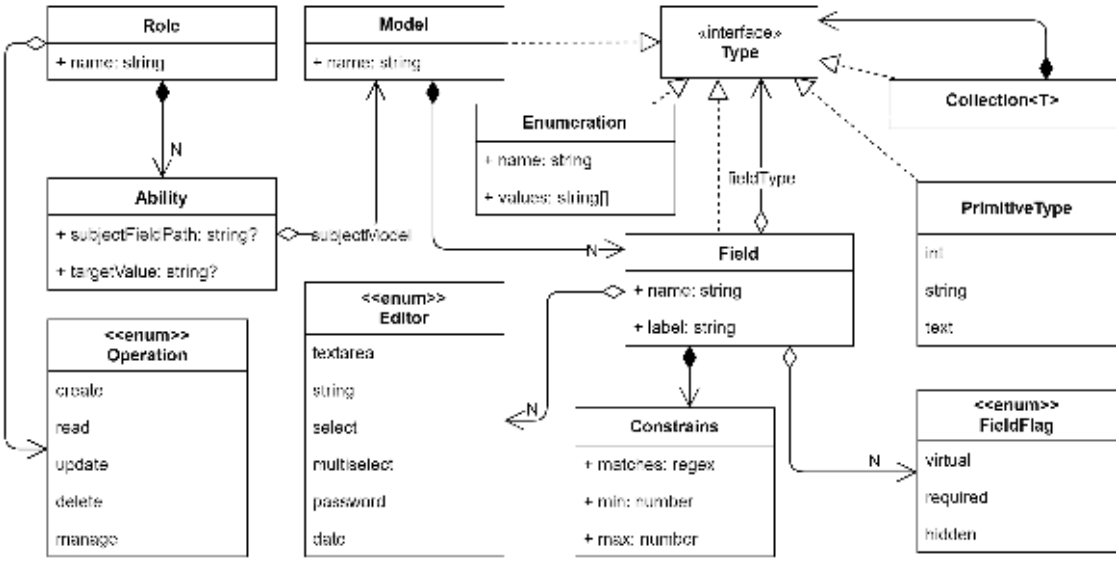
Generátory sú implementované s použitím modulu **Utils**, ktorý už zahŕňa modul Semantics a obsahuje dodatočný pomocný kód pre generátory, napríklad pre generovanie súborov definujúcich rozhranie GraphQL. V prípade, že generátor nepotrebuje pridanú funkcionalitu z modulu Utils, je možné použiť modul Semantics.

4.2 Rozšírenie jazyka BFI

V tejto časti doplníme jazyk o nové **elementy** a znázorníme ho pomocou UML diagramu tried (obr. 4.2). Keďže sa autor tejto práce ako vývojár softvéru orientuje v doméne, návrhy na rozšírenie jazyka nie sú výsledkom konzultácii s cieľovými používateľmi, ale vlastných poznatkov a skúseností.

Pôvodne jazyk podporoval množinu primitívnych typov a referenciu na iné modely. V rámci referencie bolo možné špecifikovať kardinalitu, čo predstavovalo riziko vzniku inkonzistencií v prípade obojsmerných vzťahov, respektíve potreby implementácie editora s ohľadom na kontext. Bezpečnejším riešením je nahradenie kardinalít zavedením nového podporovaného typu, kolekcie (**Collection**). Jazyk sme tiež rozšírili o podporu enumeračného typu (**Enumeration**).

Pre triedu **Field** máme k dispozícii nové nástroje (**Constrains**) ako zabezpečiť správnosť jeho obsahu. Týmito nástrojmi sú napríklad špecifikovanie **minima** a **maxima** v prípade číselného typu, respektíve dĺžky reťazca. Túto funkcionalitu môžeme využiť napríklad pre vek, výšku, rok, dátum, bodové hodnotenie, percentuálne vyjadrenie a podobne. Pre textové typy máme možnosť zabezpečiť správnosť dát aj pomocou regulárneho výrazu v atribúte **matches**. Keďže náš



Obr. 4.2: UML Diagram typov

rámec nepodporuje časové typy ako `date`, `time`, `datetime`, z dôvodu obmedzení štandardnej definície GraphQL bez dodatočných rozšírení, je možné ich čiastočne nahradiť týmto spôsobom. Tento atribút je vhodné tiež použiť napríklad na zabezpečenie veľkého začiatočného písmena pri vlastných podstatných menách, zabezpečenie formátu poštového smerovacieho čísla, telefónneho čísla, čísla kreditnej karty, emailovej adresy a mnoho ďalšieho. Táto funkcionalita je realizovateľná v serverovej časti, prostredníctvom validácie pred vkladaním, alebo upravovaním hodnôt v dátovom úložisku a taktiež vo webovej časti aplikácie vo formulároch. V druhom prípade je toto generovanie obtiažnejšie, pretože generátor musí identifikovať správne HTML značky kde má tieto atribúty zohľadniť. V prípade Angular generátora boli v predošlej práci zavedené špeciálne XML značky, ktoré sa v procese generovania preložia na HTML elementy. Jedna z týchto špeciálnych značiek je **bfi-input**, ktorá je využiteľná práve pre zabezpečenie validity dát pomocou spomínaných nástrojov `min`, `max` a `matches`.

Bola tiež pridaná možnosť definovať roly (**Role**), ktoré môžu byť následne v systéme pre správu prístupu priradené používateľom. Tieto roly sú zoznamy schopností (**Ability**), ktoré používateľ môže vykonávať. Schopnosť je definovaná operáciou (**Operation**) a modelom, na ktorej je možné danú operáciu vykonať. Upresniť schopnosť je možné pomocou podmienky, ktorá musí byť splnená aby bolo možné operáciu vykonať. Podmienka je definovaná rovnosťou hodnoty atribútu daného

modelu, špecifikovaného cestou (**subjectFieldPath**), a konštanty (**targetValue**), prípadne hodnoty mena prihláseného používateľa (špeciálna konštanta `#username`). Príkladom takejto rovnosti pre model `Patient` môže byť `.doctor.name = #username`. Takúto schopnosť môže obsahovať napríklad rola `Doktor`, a hovorí, že operácia sa viaže len ak je daný používateľ doktorom daného pacienta. Iným príkladom môže byť podmienka `.active = 1`, ktorá obmedzí platnosť schopnosti len pre aktívnych pacientov.

```
8  model Category
9    name:string label="Category name" editor=string required
     ↪  matches="[A-Za-z ]{1,20}" min=1 max=20
10   tasks:Task[] label="Tasks"
11 model User
12   name:string
13 enum State [Todo,InProgress,Done]
14 role User
15   can manage Task where .user.name: #username
16   can read Category
```

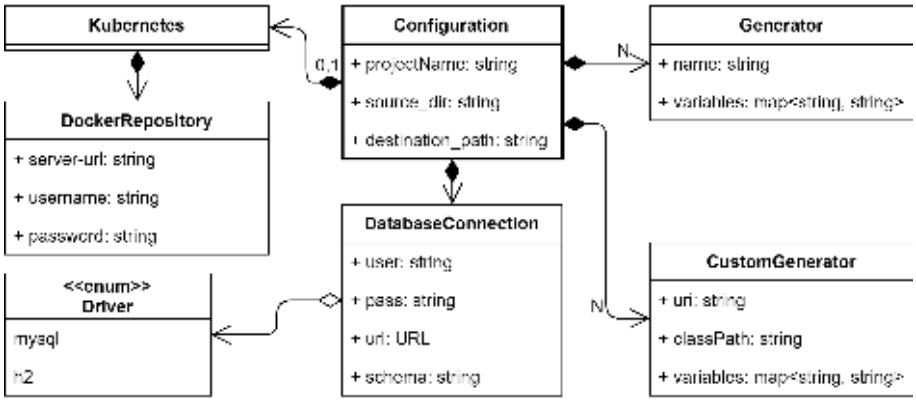
Zdrk. 4.10: BFI - príklad modelu (nekompletný)

Na ukážke 4.10 vidíme časť vzorového modelu s využitím zavedených prvkov. Pribudlo tiež nové označenie **required**, ktoré označuje atribút, ktorý nesmie byť prázdny.

4.3 Konfiguračný súbor

Bola pridaná podpora konfiguračného súboru, ktorý nahrádza doteraz používané spúšťacie **argumenty**. Súčasťou tejto konfigurácie je nastavenie pripojenia na **databázu**, a najmä konfigurácia požadovaných **generátorov**. Každý generátor má unikátne meno, a vlastné podporované parametre.

Podľa predvolených nastavení je vyhľadávanie BFI modelových súborov vykonávané v zložke s týmto konfiguračným súborom. Ak používateľ preferuje mať tieto súbory oddelené, je možné upresniť cestu k nim pomocou atribútu **source dir**. Atribút **destination path** upresňuje cestu, kde budú umiestnené generované súbory projektov. Pomocou objektu **Kubernetes** zadávame atribúty, potrebné pre



Obr. 4.3: Diagram tried konfiguračného súboru

generovanie súborov používaných pre nasadenie aplikácie do tohto systému. V tomto momente sú jediným atribútom prihlasovacie údaje do Docker repozitára, kde bude dostupný obraz aplikácií. Na základe tohto objektu sa vygeneruje v projektoch súbor `Secret.yaml`, ktorý je nutné nasadiť, aby bol Kubernetes schopný stiahnuť potrebný obraz.

Podobne ako syntaktická analýza súboru BFI je aj analýza konfiguračného súboru zahrnutá v module **core-parser**. Výsledkom tejto fázy je sémantický model reprezentovaný Java objektami. Táto konfigurácia spolu s BFI modelom tvorí kontext pre generátory.

4.4 Manažment generátorov a procesu generovania

V predošlej práci bolo spúšťanie generovania nutné vykonať pomocou príkazu v rozhraní príkazového riadku. Navyše tieto generátory museli byť umiestnené lokálne na disku. Išlo o nepraktické riešenie z pohľadu pohodlnosti, ale aj aktualizácií generátorov. Preto sa chceme zbaviť nutnosti ich manuálneho sťahovania a aktualizovania, a vytvoriť **spúšťač**, ktorý bude schopný tieto úlohy vykonať na pozadí.

Generátory umiestníme na internetové úložisko v podobe balíčkov JAR. Tieto balíčky nebudú samostatne spustiteľné a všetky budú musieť obsahovať triedu, ktorá implementuje rozhranie `Generator` z jadra BFI. Spúšťač teda musí mať informáciu o presnej lokácii generátora a tiež cestu k hlavnej triede generátora. Následne bude pri prvom použití daného generátora **stiahnutý** potrebný balíček, ktorý je identifikonvaý pomocou **mena** generátora v konfiguračnom súbore. Keďže

chceme mať verziu spúšťača a tým aj zásuvného modulu nezávislú od vývoja generátorov, tieto informácie sú uložené v **definičnom súbore** 4.11.

```
1 - name: springboot
2   generator:
3     classPath: sk.tuke.ales.bulko.bfi.generator.spring.boot.generator.Spr⌋
      ↪ ingBootGenerator
4     uri: https://s3.eu-west-3.amazonaws.com/bfi.framework/generator/sprin⌋
      ↪ g-boot/spring-boot-0.0.1-SNAPSHOT.jar
```

Zdrk. 4.11: BFI zoznam generátorov

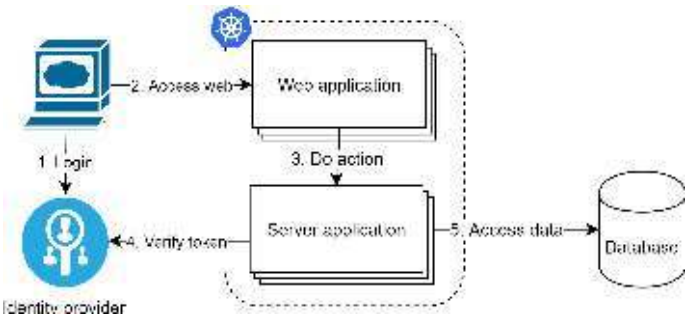
Tento súbor bude rovnako uložený v internetovom úložisku. V prípade vydania nového generátora je teda postačujúce upovedomiť správcu softvérového rámca a požiadať ho o pridanie nového záznamu bez nutnosti vydania novej verzie spúšťača. Pre účely použitia neregistrovaných generátorov je možné využiť atribút `customGenerator` v konfiguračnom súbore, pomocou ktorého uvedieme cestu ku generátoru pomocou URI a cestu k hlavnej triede generátora.

4.5 Vzniknutý systém

Cieľovým systémom je jednoduchá webová aplikácia pozostávajúca zo **serveru** a klientskej **webovej** aplikácie. Používateľ je pri prístupe na stránku vyzvaný na prihlásenie sa, alebo registráciu. Tú zaobstaráva zvolený systém pre autorizáciu, ktorý je obvykle prepojený s rôznymi službami poskytovateľov emailových služieb, sociálnymi sieťami a podobne. Po prihlásení používateľov prehliadač obdrží bezpečnostný prístupový token, ktorým sú následne autentifikované všetky nasledujúce požiadavky.

Autorizačné služby a databáza sú poskytované **externe** a závisia od konfigurácie používateľa. Pripojenie na databázu je možné nakonfigurovať v konfiguračnom súbore BFI. Popísanú architektúru môžeme vidieť na diagrame 4.4.

Samotnú serverovú a klientskú časť aplikácie je možné prevádzkovať v **Kubernetese**, pričom všetky potrebné konfiguračné súbory sú vygenerované v adresároch jednotlivých projektov. Aplikácie sú navrhnuté so zameraním na bezstavovú (z angl. „stateless“) architektúru a v prípade použitia v Kubernetes je možné vytvárať ľubovoľný počet ich replík. Výhody tejto architektúry boli popí-



Obr. 4.4: Znázornenie architektúry webovej aplikácie

sané v časti 3.5.2.

Táto architektúra pozostáva z niekoľkých častí, respektíve Kubernetes objektov. Pomocou objektu **ConfigMap** definujeme premenné prostredia potrebné pre beh našej aplikácie. Hlavným objektom je objekt **Deployment**, ktorý obsahuje deklaráciu želaného stavu, v našom prípade nasadenej serverovej, respektíve klientskej aplikácie. Objekt Deployment špecifikuje:

- referenciu na obraz aplikácie,
- referenciu na objekt ConfigMap,
- počet želaných replík,
- metadáta (názov objektu, selektor, značky, využívané porty).

Tretí objekt **Service** slúži na zviditeľnenie aplikácie v sieti. Každému objektu Deployment priradí DNS meno a zároveň automaticky rozloží záťaž, teda vhodne presmeruje požiadavky medzi jeho repliky. Konfigurácia týchto objektov bude uložená v zložke kubernetes v jednotlivých projektoch.

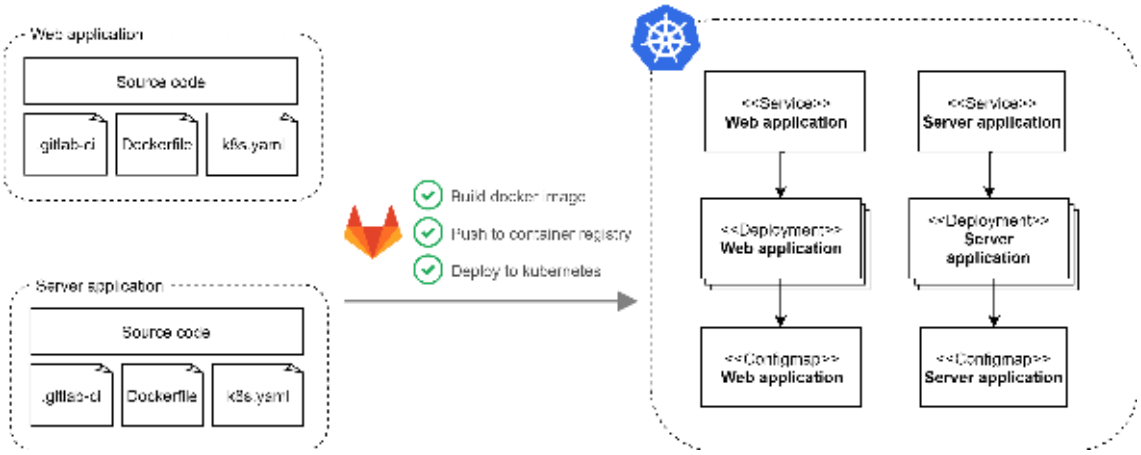
4.6 Podpora procesu nasadenia aplikácie

Pre nasadenie aplikácia do systému Kubernetes je potrebné vytvoriť obraz aplikácie pomocou Docker. Vytvorenie tohto obrazu prebieha vrstvením, a tento postup je definovaný v súbore **Dockerfile**. Bez ohľadu na použitý generátor budú tieto vrstvy pozostávať z:

- existujúceho obrazu dostupného vo verejnom repozitári Docker Hub,

- pridania zdrojového kódu, prípadne jeho skompilovanej podoby,
- príkazu na spustenie aplikácie.

V závislosti od typu aplikácie a generátora môže obsahovať napríklad informáciu o odhalenom porte, príkazu na stiahnutie knižníc, príkazov na spustenie interných generátorov a podobne.



Obr. 4.5: Znázornenie nasadenia aplikácie

Spomínaná podpora nepretržitej integrácie bude realizovaná prostredníctvom služby Gitlab CI integrovanej do verzionovacieho systému Gitlab. Konfigurácia tohto procesu je popísaná pomocou súboru **.gitlab-ci**. Ten bude obsahovať:

- zostavenie aplikácie (skompilovanie, zabalenie knižníc, kompresia zdrojového kódu a podobne),
- vytvorenie obrazu aplikácie pomocou Dockerfile,
- propagácia obrazu verzie do repozitára,
- nasadenie aplikácie pomocou súboru k8s.yaml

Tento súbor bude následne **automaticky** spustený pri každej aktualizácii vzdialenej vetvy v systéme Gitlab. Pred tým je však nutné aby používateľ v systéme Gitlab nastavil potrebné **premenné** prostredia, ako je to uvedené v súboroch `Readme.md` v daných projektoch.

5 Implementácia rámca BFI a podporných systémov

V tejto kapitole si popíšeme implementáciu návrhov, ktoré boli spomenuté v predošlej kapitole 4. Bude pozostávať z implementácie procesu samotného generovania na základe BFI modelu, a z implementácie systémov pre podporu vývoja v našom softvérovom rámci.

5.1 Generovanie komponentov cieľového systému

Budeme sa zaoberať generovaním kódu jednotlivých aplikácií, ako aj zabezpečenie ich vzájomnej kompatibility. Jednou z priorít tiež bude zabezpečenie dostatočnej schopnosti modifikovať vygenerovanú aplikáciu formou oddelenia generovaného a špecifického kódu dotvoreného programátorom. Všetky generátory budú implementované v jazyku Java.

5.1.1 Definícia GraphQL rozhrania

Pre zjednotenie komunikačného rozhrania bola zavedená technológia GraphQL. Umožňuje presne definovať objekty a metódy pre prácu s dátami pomocou doménovo-špecifického jazyka SDL („Jazyk pre definíciu schémy“). Táto definícia sa skladá zo **schémy**, ktorá definuje štruktúru rozhrania, a **dokumentov**, ktoré slúžia ako šablóny pre odosielanie požiadaviek. Generovanie týchto súborov je implementované v jadre BFI, pretože sú **spoločné** pre všetky aplikácie typu klient a server.

Schéma

Je uložená v súbore, obvykle s názvom `schema.graphql` (zdrojový kód 5.12).

```
type Task {
  id: ID!
  name: String,
  state: State,
  categories: [Category]
}
enum State {
  TODO, IN_PROGRESS, DONE
}
type Query {
  tasks(page: Int, size: Int, name: String, state: State, categories:
  ↪  Int):[Task]
  task(id: ID!):Task
}
type Mutation {
  createTask(name: String!, state: State, categories: [Int]):Task
  updateTask(id: ID!, name: String!, state: State, categories: [Int]):Task
  deleteTask(id: ID!):Boolean
}
```

Zdrk. 5.12: GraphQL schéma

V prvej časti schémy vidíme definíciu **typov**, ktorá pozostáva z jeho mena a atribútov. Ďalej môžeme vidieť definíciu **enumeračné** typu a jeho hodnôt. Tieto objekty presne odrážajú modely definované pomocou BFI a vo veľkej miere medzi sebou zdieľajú obojstranné referencie. V každom objekte je pridaný **identifikátor**, ktorý má špeciálny typ ID. S týmto typom je zaobchádzané rovnako ako s reťazcom znakov, avšak jeho myšlienkou je navádzať používateľa, že tento atribút nie je možné meniť. Výhodou jeho použitia oproti číselnému identifikátoru, ktorý je často prvou voľbou v prípade SQL databáz, je jednoduchý prechod na UUID reťazce, používané v no-SQL databázach.

Druhou časťou schémy je definovanie vykonateľných metód pre získavanie (**Query**) a úpravu dát (**Mutation**). Pre každý objekt sme zvolili definovanie dvoch základných metód pre čítanie. Prvá slúži na vyhľadanie všetkých objektov podľa všetkých jeho atribútov s podporou stránkovania. Druhou metódou je možné získať objekt podľa jeho identifikátora. Pre úpravu objektov definujeme 3 metódy, a

to pre **vytvorenie** nového záznamu, **úpravu** existujúceho záznamu a jeho **zmazanie**. Návratovou hodnotou operácií pre vytvorenie a úpravu je výsledný objekt.

Daná schéma definuje čo rozhranie ponúka. Na základe nej môžeme prostredníctvom GraphQL knižníc pre cieľové jazyky/rámce **serverovej** aplikácie využiť pomocné nástroje pre manažovanie požiadaviek API. Okrem toho plne opisuje rozhranie a je ju teda možné využiť napríklad pre vygenerovanie dokumentácie, alebo spustenie interaktívnych prehliadačova API ako napríklad *GraphiQL*.

Dokument

Druhým generovaným typom súboru je GraphQL dokument. Ako bolo spomenuté, dokument je akási **šablóna** pre vykonávanie operácií rozhrania, z ktorej doplnením kontextu vzniká požiadavka. Deklaruje volanie, ktoré odkazuje na existujúcu metódu rozhrania zo schémy, avšak už s konkrétnou odpoveďou akú z tejto metódy očakávame.

```
query GetAllTasks($name: String, name: String, $state: State, $categories:
↪ Int) {
    tasks(name: $name, state: $state, categories: $categories) {
        id
        name
        state
        categories {
            ... CategoryInfo
        }
    }
}

fragment CategoryInfo on Category {
    id
    name
}
```

Zdrk. 5.13: GraphQL dokument modelu Task

Toto volanie deklaruje **parametre**, ktoré je treba následne poskytnúť v požiadavke spolu s touto šablónou. Hlavnou výhodou GraphQL je však variabilita **od-**

povede, ktorej požadovanú štruktúru definujeme práve na tomto mieste. Nevieme však aké dáta bude používateľ skutočne potrebovať, a preto si bude musieť podľa potreby takéto dokumentu vytvoriť manuálne. Keďže používame vzájomné referencie medzi objektami, je potrebné zabezpečiť, aby v odpovedi nevznikla nekonečná rekurzia. Ak teda návratový typ metódy rozhrania obsahuje referenciu na iný objekt, nesmieme požadovať znovu spätnú referenciu na pôvodný objekt. Pri komplexných databázových štruktúrach by mohla vzniknúť nadmerne veľká odpoveď a tiež požiadavka na databázové úložisko. Preto sme v rámci generovaných dokumentov definovali návratové typy obsahujúce všetky atribúty prvej úrovne požadovaného objektu a primitívne atribúty druhej úrovne, teda referovaných objektov. Pre udržanie prehľadnosti a obmedzenie duplicity kódu je v tomto dokumente použitý element **fragment**, ktorý predstavuje určitú časť komplexnejšieho objektu, v našom prípade jeho primitívnych atribútov.

Generovanie tohto dokumentu je rovnako zahrnuté v jadre BFI, avšak jeho použitie pri tvorbe generátora nie je potrebné. Je však odporúčané z dôvodu možnosti využitia GraphQL generátorov klientských častí rozhrania. Pomocou nich je možné **generovať** klientské metódy v cieľovom jazyku. Bez týchto dokumentov totiž GraphQL generátoru chýba informácia o tom, aké atribúty objektu (fragment) by mala potenciálne vygenerovaná metóda obsahovať.

5.1.2 Springboot generátor

V rámci práce sme implementovali nový generátor **serverovej** aplikácie v jazyku Java s softvérovým rámcom Spring, presnejšie jeho nadstavbou Springboot. Tento rámec vo veľkej miere využíva techniky meta-programovania prostredníctvom anotácií. To nám uľahčuje implementáciu generátora.

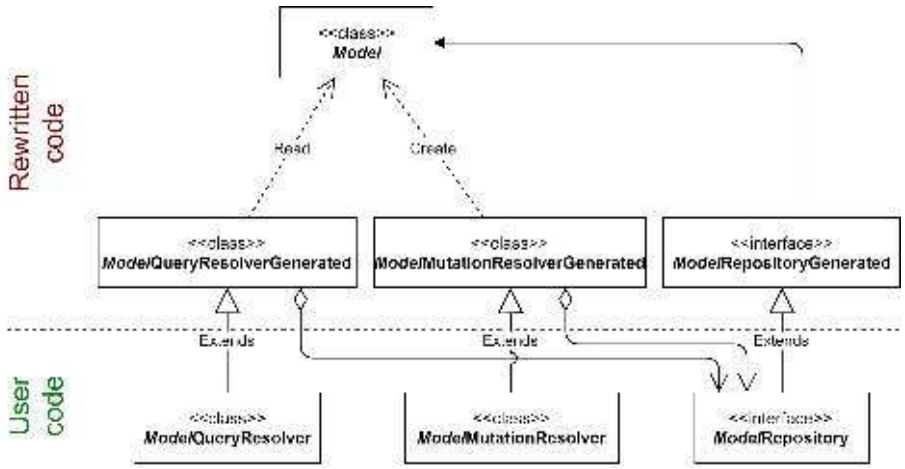
Pri prvom spustení generátor vytvorí **kostru** programu s využitím systému maven a nastaví atribúty projektu podľa vstupnej konfigurácie používateľa. Ide o súbory, ktoré nezávisia od vstupného modelu, teda triedy konfigurácie, `pom.xml`, `.gitignore`, `mvnw`, a podobne. Tieto súbory následne už nie sú prepisované pri ďalších spusteniach generátora.

Minimálna implementácia webovej služby spočíva vo vytvorení troch komponentov, respektíve tried

- **entita** reprezentujúca dátovú štruktúru modelu,

- **repozitár** slúžiaci ako rozhranie pre manipulovanie s dátovým úložiskom objektov danej entity,
- **kontrolér** ktorý sprístupňuje metódy pre manipuláciu so systémom.

V našom prípade sme ako rozhranie zvolili použitie jazyka GraphQL. GraphQL poskytuje **knižnicu** pre rámec Springboot, ktorá nahrádza posledný menovaný komponent kontrolér na základe definičného súboru `definition.graphqls`. Generovanie schémy je implementované v spoločnom module generátorov BFI, nakoľko ide o definíciu rozhrania, ktorá je znovupoužiteľná v iných generátoroch.



Obr. 5.1: Diagram tried vygenerovaného SpringBoot projektu pre Model

Na obrázku 5.1 je znázornený diagram tried, ktoré generujeme pre každý model. Všimnúť si môžeme ďalšie triedy QueryResolver a MutationResolver. Tieto triedy opisujú správanie programu po zavolaní príslušnej metódy webového rozhrania. Trieda **QueryResolver** obsahuje metódy pre čítanie objektov daného modelu vrátane ich filtrovania podľa požiadaviek používateľa. Toto filtrovanie je implementované pomocou Criteria API, ktoré poskytuje rámec Spring. Na tejto úrovni tiež prebieha overovanie autorizácie prihláseného používateľa pre vykonanie danej operácie v prípade, že používateľ v rámci BFI definoval potrebné roly. Trieda **MutationResolver** obsahuje metódy pre manipulovanie s objektami, teda ich vytváranie, aktualizovanie a mazanie.

Pre zachovanie možnosti dopĺňania novej funkcionality je využitá technika dedičnosti. Zatiaľ čo triedy modelov a triedy s príponou „Generated“ sú prepisované každým spustením generátora (**Rewritten code**), zvyšné triedy sú vygenerované len pri prvom spustení (**User code**). To programátorovi dovoľuje dopĺňať vlastnú

biznis logiku a vlastné metódy pre prácu s dátami ak je to potrebné. Tieto triedy neobsahujú žiadny vykonateľný kód a rozširujú triedy s príponou generated.

```
#user config start
okta.oauth2.issuer=https://dev-123456.okta.com/oauth2/default
okta.oauth2.client-id=0da8wd84a6d384axcw86
okta.oauth2.client-secret=das846x68sac4s8f64s6c84e6f87425f47asgfrh
okta.oauth2.groups-claim=groups
#user config end
spring.datasource.username=root
spring.datasource.url=jdbc:h2:file:./database;AUTO_SERVER=TRUE
spring.datasource.driverClassName=org.h2.Driver
server.port=8080
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.datasource.password=root
```

Zdrk. 5.14: Springboot konfiguračný súbor

Na obrázku 5.14 je uvedený konfiguračný súbor `springboot.properties`, ktorý je rozdelený na užívateľskú a generovanú časť súboru. Keďže v tomto prípade nemôžeme využiť techniku dedenia, bolo nutné navrhnúť iný systém, aby sme používateľovi umožnili **meniť** generovanú konfiguráciu. Špeciálnymi značkami vo forme komentárov (`#user config start/end`) sme ohraničili riadky, ktoré nie sú pri opätovnom behu generátora prepisované. Ak používateľ v tejto časti uvedie existujúci kľúč, ktorý sa normálne nachádza v generovanej sekcii, tento riadok pri následnom generovaní nebude doplnený. Vďaka tomu používateľ môže nielen rozširovať konfiguračný súbor ale aj prepisovať generovaný obsah.

Pri zavedení podpory riadenia prístupu (kapitola 3.3) sme vytvorili novú požiadavku, ktorú musí zohľadniť serverová aplikácia. Tou je podpora pre kontrola **autorizácie** používateľa a **filtrovanie** dát. Softvérový rámec Spring v rámci rozšírenia *security* ponúka pre tento účel anotácie `@PreAuthorize` a `@PostFilter`.

```
role User
    can manage Task where .user.name: #username
```

Zdrk. 5.15: Definícia roly User

Na ukážke zdrojového kódu 5.16 vidíme použitie týchto anotácií, ktoré zodpovedajú definícii roly (zdrk. 5.15).

```
26    @PreAuthorize(value ="(hasAuthority('User'))")
27    @PostFilter(value ="(hasAuthority('User') and filterObject.user.name
      ↪  == authentication.name)")
28    public List<Task> getTasks(final Integer page, final Integer size,
      ↪  final String name, final String description, final State state,
      ↪  final String dueDate, final Integer categoriesId, final Integer
      ↪  userId) {
```

Zdrk. 5.16: Príklad kontroly prístupu rámcom Spring

Tieto pravidlá sú vyjadrené doménovo-špecifickým jazykom SpEL („Spring Expression Language“).

5.1.3 Angular generátor

Pre generovanie klientskej časti aplikácie sme sa rozhodli **upraviť** existujúci generátor a pridať podporu nových rozšírení, najmä úpravy rozhrania. Ostala teda zachovaná podpora generovania HTML zo špeciálnej šablóny BFI, obsahujúcej XML elementy BFI (kapitola A.6.1).

Upravené bolo v prvom rade generovanie tried zodpovedajúcich za **komunikáciu** so serverom. Využili sme pri tom funkcionalitu jadra BFI pre generuje GraphQL dokumentov. Pridaním skupiny knižníc `graphql-codegen` a potrebných skriptov do konfiguračného súboru nástroja NPM `package.json` umožňujeme používateľovi z týchto dokumentov vygenerovať klienta príkazom **npm run gql:codegen**. Výhodou tohto prístupu je zjednodušenie implementácie nášho generátora, keďže sa spoliehame na existujúce a spoľahlivé riešenie, a tiež šetríme prácu programátorovi pri vytváraní nových metód. Nevýhodou je potrebná znalosť doménovo špecifického jazyka GraphQL SDL, pre ktorý mu ale predpripravené dokumenty môžu slúžiť ako vzor, čo uľahčuje jeho pochopenie. Spolu s týmito vygenerovanými službami sú vygenerované aj typy, preto sme z generovaných komponentov odstránili generovanie entít. Modul HTTP klienta bol nahradený klientom Apollo, ktorý je určený špeciálne pre komunikáciu pomocou GraphQL.

```
15    public nameFormControl = new FormControl(this.task.name,
      ↪  [Validators.pattern(/^[A-Za-z ]{1,20}$/), Validators.minLength(1),
      ↪  Validators.maxLength(20)]);
```

Zdrk. 5.17: Príklad FormControl

Druhou dôležitou funkcionalitou o ktorú bol generátor rozšírený, je podpora **validácie** dát (kapitola 4.2). Táto validácia je v tomto generátore riešená pomocou objektov **FormControl** (zdrk. 5.17). Tieto objekty môže následne programátor použiť v Angular HTML šablóne.

```
3   <div>
4    <label for="name">Task name</label>
5    <input type="text" name="name" id="name"
      ↪  [formControl]="nameFormControl">
6    <span *ngIf="nameFormControl.errors">
7     <ng-container *ngIf="nameFormControl.errors.pattern">
8      Must match '[A-Za-z ]{{"{"}}1,20{{"}"}}'
9     </ng-container></span>
10  </div>
```

Zdrk. 5.18: Príklad prekladu BFI input

V prípade, že používateľ využije špeciálnu BFI šablónu, atribút **bfi-input** zabezpečí po preložení automatické použitie FormControl, ako môžeme vidieť na ukážke 5.18.

5.1.4 Liquibase generátor

Nástroj Liquibase (kapitola 3.2.2) umožňuje **porovnávanie** databáz, aplikovanie a generovanie migračných skriptov. V kombinácii so softvérovým rámcom Hibernate sme schopní na základe modelu tried doplneného o anotácie obsahujúce metadáta databázových entít vytvoriť jednoducho rozdielové SQL skripty pre **migráciu** na korešpondujúcu verziu databázy s naším modelom. Generovanie týchto entít sme už implementovali v rámci generátora Springboot, a tak sme v tomto prípade kód recyklovali, pričom sme získali dôležitú funkcionalitu. Túto funkcionalitu sme nezahrnuli do generátora Springboot, pretože môže byť využitá aj v

kombinácii s inými serverovými aplikáciami. Návod pre vytvorenie takéhoto rozdielového skriptu sme uviedli v súbore s nápovedou k projektu `README.md`.

5.2 Zásuvný modul pre vývojové prostredia Jetbrains

Pre zjednodušenie vývoja v našom softvérovom rámci bolo podstatné zaviesť najmä **podporné** systémy pre tvorbu definície modelu v jazyku BFI (kapitola 4.2) a konfiguračného súboru (kapitola A.3)).

Naším cieľom je, aby používateľ s čo najmenšou námahou dokázal rýchlo začať pracovať s naším softvérovým rámcom a nemusel študovať rôzne príručky, dodržiavať rôzne postupy nutné pre uvedenie systému do prevádzky, inštalovať podporné aplikácie a knižnice, poznať syntax pre príkaz na spustenie generátorov a podobne.

Všetko to, čo pre naše potreby potrebujeme, je možné dosiahnúť prostredníctvom rozšírenia pre vývojové prostredie, v tomto prípade pre vývojové prostredia Jetbrains. Rozšírenia je možné inštalovať priamo v zabudovanom **obchode** s rozšíreniami, ktorý je prístupný v nastaveniach.

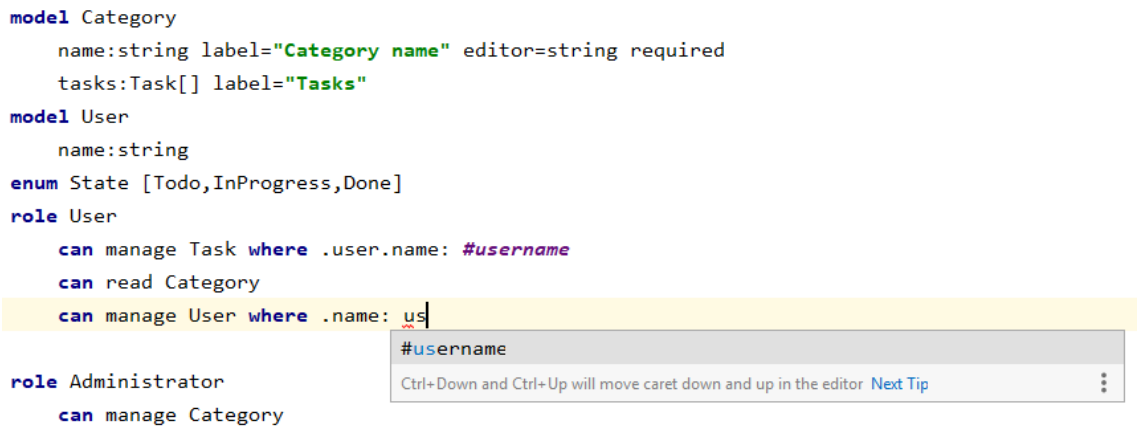
Našimi prioritami bude pomocou tohto zásuvného modulu priniesť:

- podporu editovania doménového modelu v jazyku BFI,
- podporu editovania konfiguračného súboru v jazyku YAML/JSON,
- jednoduché spustenie procesu generovania,
- prehľad o procese a možných chybách v procese generovania.

5.2.1 Podpora editovania doménového modelu v jazyku BFI

K tomuto účelu slúži balíček vývojových nástrojov pre platformu *IntelliJ Platform Plugin SDK*, ktorý obsahuje aj nástroje určené pre tvorbu doménovo-špecifických jazykov. S jeho využitím sme vytvorili rozšírenie, ktoré po pridaní do vývojového prostredia dokáže podľa prípony **identifikovať** definičné súbory a vniesť do editora príslušné funkcionality. Najzákladnejšou funkcionalitou je živá **kontrola syntaxe** editovaného súboru. Keďže tento balíček nástrojov nespolupracuje s nástrojom Antlr, boli sme nútení prepísať gramatiku z jazyka Antlr do formátu BNF. Pomocou *Grammar-Kit* od Jetbrains sme z tejto gramatiky podobne ako v prípade

Antlr vygenerovali Parser a Lexikálny analyzátor. Vďaka nemu prostredie automaticky dokáže **zvýrazniť** neočakávané reťazce a ponúknuť používateľovi **nápovedu** s očakávaným obsahom. Ďalšou funkcionalitou bolo pridanie farebného **formátovania** kódu, čo napomáha k rýchlej orientácii a prehľadnosti v kóde. Zvolili sme zvýrazňovanie kľúčových slov jazyka a textových reťazcov.



Obr. 5.2: Jazykové rozšírenie BFI

Ďalšou funkcionalitou je automatické bezkontextové **dokončovanie** používateľovho vstupu. Toto dokončovanie funguje pre kľúčové slová jazyka, nie však pre referencie na názvy modelov, respektíve atribútov modelov.

5.2.2 Konfigurácia generovania

V návrhu sme opisovali zavedenie konfiguračného súboru (kapitola A.3), ktorý obsahuje globálne nastavenia **projektu** a samotných **generátorov**. Pre zadanie tejto konfigurácie sme implementovali podporu jeho vyjadrenia v syntaxi JSON, pomocou modulu **Parser**. Na popísanie jeho štruktúry bol použitý jazyk **JSON Schema**, čo je jazyk pre definíciu štruktúry JSON dokumentu, obdobne k nástroju XML DTD. Implementáciou v tomto jazyku získavame tiež podporu jazykov YAML a TOML, ktoré vychádzajú práve z jazyka JSON a je ich možné ľubovoľne medzi sebou transformovať.

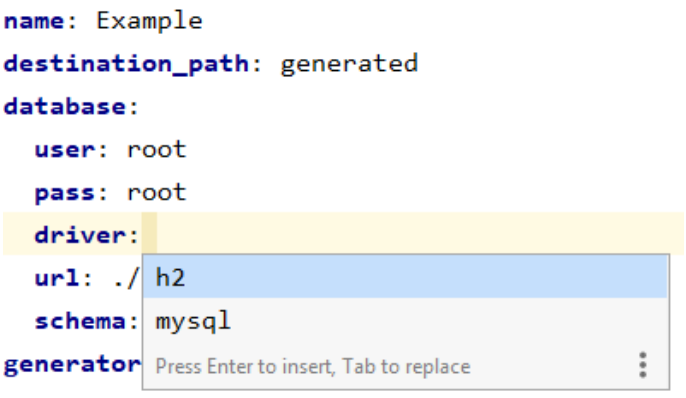
```
40    "type": "object",
41    "required": [
42      "name",
```

```
43      "generators",
44      "database"
45    ],
46    "additionalProperties": false,
47    "properties": {
48      "name": {
49        "$id": "#/properties/name",
50        "type": "string"
51      },
```

Zdrk. 5.19: BFI konfigurácia (časť) - Json schema

Na úseku kódu 5.19 vidíme napríklad definíciu koreňovej štruktúry dokumentu. Napríklad atribút **properties** definuje atribúty danej úrovne tvorené ich menami, typmi, identifikátormi, obmedzeniami a podobne. V prípade zloženého atribútu vzniká obdobne ďalšie vetvenie. Ako typ je možné použiť odkaz na definíciu definovanú v hlavičke dokumentu, alebo v inom dokumente. Atribút **required** definuje ktoré z týchto atribútov sú povinné.

Táto schéma plní dve funkcie. Použitím generátora *jsonschema2pojo-maven-plugin* vo forme maven pluginu je možné **vygenerovať** Java triedy, spolu s potrebnými anotáciami pre následné **namapovanie** obsahu konfiguračného súboru s použitím existujúcich knižníc ako Jackson ObjectMapper. Moderné vývojové **prostredia** umožňujú načítať tento súbor a ponúknuť podporu pri editovaní súborov (obrázok 5.3) v tomto formáte.



Obr. 5.3: Bfi konfigurácia

Bez potreby implementácie vlastného pluginu nám teda vývojové prostredie

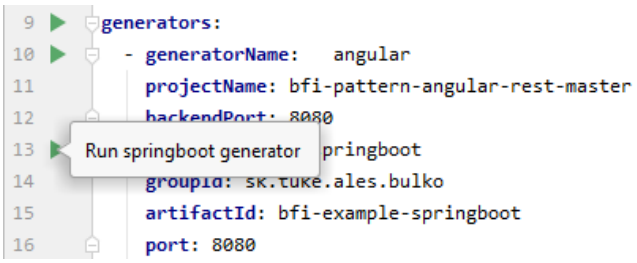
Jetbrains poskytne:

- kontrolu syntaxe,
- zvýrazňovanie,
- automatickú kompletizáciu slov.

Tento súbor popisujúci schému konfigurácie je umiestnený na vzdialenom úložisku a pri nainštalovaní nášho rozšírenia sa automaticky **pripojí** do IDE, ak sa tam už nenachádza.

5.2.3 Spúšťanie generátorov

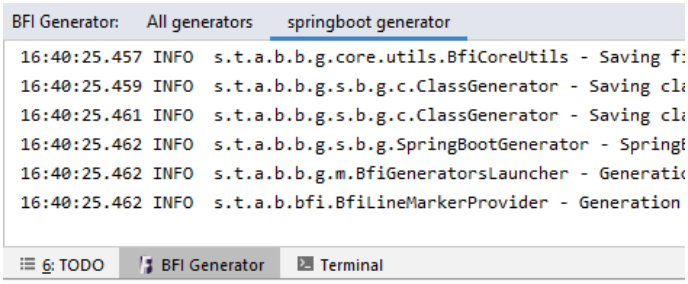
Ďalšou oblasťou z pohľadu použiteľnosti, ktorú bolo nutné vylepšiť bolo samotné spúšťanie generátorov. Daný spúšťač sme zabalili priamo do zásuvného modulu. Pre jeho spustenie sme zvolili doplnenie akcie do *Gutter* panela („odkvap“) v podobe zelenej **značky** spustenia.



Obr. 5.4: Spustenie generátora Springboot

Pri vytvorení potrebného konfiguračného súboru sa používateľovi tieto značky automaticky zobrazia pri elemente *generators* a pri elementoch jednotlivých generátorov na riadku obsahujúcom element *generatorName*. V druhom prípade sa spustí len **príslušný** generátor, a naopak v prípade že používateľ klikne na ikonu na riadku elementu *generators*, spustia sa postupne **všetky** uvedené generátory.

Priebeh procesu generovania môže používateľ sledovať v špeciálnom okne nástrojov, zobrazenom štandardne na spodnej lište prostredia. Každá spomínaná značka pre generovanie vytvára záznamy do **samostatného** okna, podľa názvu generátora, prípadne okna *All generators* v prípade hromadného spustenia.



Obr. 5.5: Okno nástroja BFI

5.2.4 Dočasné ukladanie sťahovaných súborov

Vo vývoji zásuvného modulu sme niekoľko súborov umiestnili na internetové úložisko a náš modul sa teda stáva závislý od internetového **pripojenia**. Pri jeho vývoji, ale aj pri vývoji samotných generátorov sme dbali na zachovanie čo najmenšej veľkosti pre ľahkú distribúciu, a vyvarovali sa používaniu rozsiahlych knižníc, či celých rámcov. Aj napriek tomu, že veľkosti generátorov a zásuvného modulu sa pohybujú medzi 2 až 4 MB, čo je pri rýchlosti dnešných internetových pripojení možné preniesť rádovo v priebehu pár sekúnd, štahovanie predstavuje zbytočné **oneskorenie** generátorov. Preto sme do jadra BFI implementovali možnosť automatického dočasného **uchovávaniu** na disku počítača počas preddefinovanej doby. Tieto súbory sú automaticky ukladané do štandardnej zložky pre aplikačné dáta, do podzložky s názvom *.bfi*. Internetové pripojenie je teda nutné len pri prvom spustení generátora.

6 Vyhodnotenie a zdokonalenie prvého prototypu

Po návrhu a implementácii rozšírení systému bude cieľom tejto kapitoly vyhodnotiť prvý prototyp. Vyhodnocovanie budeme realizovať na základe **pozorovania** respondentov pri používaní nášho rámca, zhodnotením výslednej aplikácie, ktorú implementujú, a pomocou krátkeho **interview**. Cieľom prvej iterácie tohto testovania bolo odhaliť čo najväčšie množstvo nedostatkov vyhotoveného riešenia, ktoré sa následne pokúsime odstrániť.

6.1 Testovanie prototypu

Pre testovanie sme vybrali dvoch používateľov, ktorým sme pripravili scenár na vyhotovenie jednoduchej webovej aplikácie, ktorú by malo byť možné implementovať v priebehu dvoch hodín.

6.1.1 Profily respondentov

Profily respondentov boli nasledujúce. Prvý respondent pracuje rok ako softvérový **vývojár prezentačnej vrstvy** softvérových aplikácií. Používa na pravidelnej báze softvérový rámec Angular, pričom pracuje výhradne na tejto vrstve a nestretáva sa s operačnou vrstvou. Má teda nadpriemerné skúsenosti so zobrazovaním dát od štruktúrovania, štýlovania a navigácie v grafickom používateľskom prostredí. Ovláda na vysokej úrovni jazyky Typescript, HTML, CSS, a jazyky PHP, a Java na mierne pokročilej úrovni. Je teda predpoklad, že náš systém dokáže oceniť najmä vďaka **doplneniu** svojej sady zručností o vývoj operačnej vrstvy a systémov pre nasadenie aplikácie.

Druhým respondentom je developer, ktorý vyvíja **rôznorodé** typy aplikácií, a je zároveň autorom predošlej práce. Ovláda jazyky Java, PHP, Javascript, Typescript, a Bash. Vo svojom zamestnaní pracuje prioritne s jazykom Java, respektíve rámcom Springboot, a s Javascript knižnicou React. Dobre sa orientuje vo sfére DevOps, teda v nástrojoch podporujúcich vývoj. Rozumie terminológii kontainerov a kontainerových klastrov, a aktívne pracuje s nástrojmi Gitlab CI/CD, Docker a Kubernetes, či správou cloud infraštruktúry AWS. Náš systém mu teda potencionálne môže pomôcť **urýchliť** prácu o zautomatizovanie často vykonávaných úloh pri počiatočných fázach tvorby nových webových aplikácií.

6.1.2 Postup testovania a spôsob vyhodnotenia

Pre respondentov sme pripravili **postup** ktorým sme ich viedli k vytvoreniu vzorovej aplikácie od jej počiatku, pričom mali k dispozícii len samotný zásuvný modul. Mali pritom otestovať **základné** funkcionality nášho rámca, teda generovanie klientskej a serverovej časti aplikácie spolu so systémom autorizácie. Týmto postupom sme ich viedli prostredníctvom telefonátu, pri ktorom nám respondenti zdieľali svoju obrazovku, aby sme mohli pozorovať ich správanie a v prípade komplikácií ich naviesť k riešeniu.

Prvým krokom bude nainštalovanie ľubovoľného najnovšieho IDE od Jetbrains pre zaručenie kompatibility, v prípade, že ním už nedisponovali. Následne je nutné pomocou integrovaného obchodu s rozšíreniami nainštalovať naše rozšírenie s názvom **BFI Language**. Popíšeme im funkcionalitu tohto rozšírenia, a požiadame o vytvorenie **dátového modelu** pre vyvíjanú aplikáciu. Ide o vytvorenie elektronickej žiackej knižky, kde si žiaci budú môcť prezerať svoje známky a učitelia navyše pridávať hodnotenia. Výsledkom prvej časti scenáru by malo byť vytvorenie zobrazovacej vrstvy pozostávajúcej z:

- jednoduchej **tabuľky** známok, kde bude uvedený žiak, učiteľ, predmet a známka,
- aspoň jedného **filtra**,
- **formulára** pre vytvorenie záznamu.

Popíšeme im atribúty, ktoré môžu v danom modelovom súbore využiť. Následne inštruujeme používateľov k vytvoreniu **konfiguračného** súboru. Očakávame, že

rozšírenie v IDE bude dostatočne nápomocné pri vytváraní tohto konfiguračného súboru a používateľom zdelíme len názvy generátorov, ktoré majú použiť, a to `sprigboot` a `angular`. Po následnom spustení generátora očakávame, že respondenti budú schopní svojpomocne spustiť oba projekty, pričom nápovedu majú k dispozícii v súboroch README v jednotlivých projektoch.

Druhá časť scenára je zameraná na implementovanie **autorizácie** s využitím prvku *role*, ktorý je súčasťou súboru definujúceho model v jazyku BFI. Študentom pristupujúcim do žiackej knižky je nutné **obmedziť** prístup len na nevyhnutné akcie. Študent by si mal vedieť zobraziť zoznam predmetov, zoznam učiteľov, a zoznam svojich známok. Na druhú stranu by nemal vedieť žiadne z týchto objektov vytvárať, upravovať, či mazať, čo má byť úlohou učiteľa, prípadne riaditeľa, či administrátora. Keďže kvôli zachovaniu univerzálnosti systém negeneruje implementáciu pre konkrétne autentifikačné služby, túto implementáciu musí používateľ pridať sám. Vo všeobecnosti si implementáciu autorizácie môžeme rozdeliť do nasledujúcich krokov:

1. vytvorenie **účtu** v službe poskytujúcej správu používateľských identít a vykonanie prvotných **nastavení** pre danú aplikáciu,
2. implementovanie podpory služby v **serverovej** časti aplikácie (zvyčajne minimálne, prostredníctvom knižnice a jej konfigurácie),
3. **kontrola** privilégií a **filtrovanie** dát poskytovaných serverovou časťou,
4. implementovanie podpory služby v **prezentačnej** vrstve aplikácie, ktoré pozostáva z:
   (a) pridania **knižnice**,
   (b) pridania tlačidla pre **prihlasovanie**, ktoré presmeruje používateľa na prihlasovací formulár poskytovateľa služby,
   (c) automatického pripojenia získaného prístupového kľúča (z angl. „**token**“) k požiadavkám smerujúcim na server.

Náš systém zabezpečuje len **tretí** z menovaných bodov, ktorý je ako jediný závislý od vstupného modelu. Keďže zvyšné úkony nie sú predmetom testovania, používateľom **poskytneme** hotové časti kódu, ktoré bolo nutné vložiť do existujúcej aplikácie. Respondentom vytvoríme používateľský účet v použitej platforme

s rolou *Student*. Po prihlásení s týmto účtom by mal používateľ, študent, vidieť len svoje známky. Pre overenie je nutné naplniť databázu testovacími záznamami. Respondenti budú mať na ukončenie scenára predom stanovený maximálny čas **2 hodiny**.

6.1.3 Výsledky pozorovania

V priebehu dvoch hodín vrátane inštruktáže a inštalácie chýbajúcich nástrojov obaja respondenti vytvorili a úspešne spustili jednoduchú stránku, ktorá obsahovala **tabuľku** so známkami. S fázou inštalácie zásuvného modulu a definovaním modelu a konfiguračného súboru sa nevyskytli problémy.

Prvý respondent bez komplikácií zadefinoval dátový model, vygeneroval a spustil aplikácie. Na rozdiel od druhého respondenta sa rozhodol nevyužívať špeciálnu BFI HTML šablónu. Po stanovenom čase mala výsledná aplikácia v tomto prípade implementovanú **tabuľku** zobrazujúcu známky študentov, ako aj jednoduchý **filter** podľa známok. Známky bolo tiež možné vytvárať pomocou **formulára**, ktorý sa zobrazil po kliknutí na tlačidlo. Respondent teda v časovom rozmedzí dvoch hodín úspešne ukončil prvú časť scenára, avšak druhú časť scenára neabsolvoval, z dôvodu vypršania času.

Druhý respondent počas tvorby modelu definoval neexistujúci dátový typ, pričom **ignoroval** syntaktickú kontrolu prostredia a spustil generovanie. Chybová hláška zobrazená v okne s výstupom sa ukázala ako **nedostatočná** a používateľa sme museli naviesť na korekciu tohto typu. Tento respondent neskôr v priebehu vývoja niekoľko krát model menil, avšak vyskytovali sa problémy s nekorektným postupom pri opätovnom nasadzovaní aplikácie. Používateľ využíval pre spustenie aplikácie *docker compose*, pričom nesprávne reštartoval kontajner, ktorý v dôsledku neobsahoval potrebný aktualizovaný kód. Používateľ mal taktiež problém s použitím špeciálnych značiek BFI, ku ktorému nemal k dispozícii dostatočne detailnú **príručku**. Po dvoch hodinách výsledná aplikácia zahŕňala tabuľku zobrazujúcu známky všetkých študentov, čo plne nesplnilo ani očakávania prvej časti scenára.

V **oboch** prípadoch sme si všimli, že používatelia očakávali vyššiu úroveň **automatizácie**, a to že budú automaticky spustené potrebné príkazy pre sťahovanie závislostí, spúšťanie druhotných generátorov a kompiláciu. Špeciálne BFI HTML **značky** sa ukázali ako nepraktické, z dôvodu nejasností pri ich fungovaní.

V závere sme sa opýtali na dojmy aké mali z práce s týmto rámcom. Respondenti sa zhodujú v tom že tento systém dokáže **ušetriť** čas pre tvorbu novej aplikácie. Podľa druhého respondenta je možné uskutočniť takýto vývoj rýchlejšie v spolupráci so **Symphony** pre PHP, ak by sme uvažovali o webovej aplikácii ako **monolitu**. V tomto prípade Symphony dokáže generovať aj jednoduché **UI komponenty**, avšak v tomto prípade nedelíme aplikáciu na prezentačnú a operačnú vrstvu, čo je dnes už dobrým zvykom. Nemôžeme využiť napríklad **validáciu** vstupu, k čomu sa hodí práve Angular. Pre tohto používateľa bolo novinkou rozhranie **GraphQL**, ktoré dokázal rýchlo **pochopiť** a cez jeho používateľské rozhranie naplniť databázu testovacími dátami. Toto rozhranie je integrované v serverovej časti aplikácie. Prechod na tento štandard teda hodnotíme pozitívne v porovnaní s REST. Z dôvodu že sa používatelia nedostali do druhej častu testovania, počas interview sme im odprezentovali použitie definovania ABAC pravidiel pomocou BFI, čo hodnotil ako veľký prínos. Pre vylepšenie systému prvému respondentovi ako vývojárovi prezentačnej vrstvy chýbalo definovanie **verzie** angular, alebo aj výber **formátu** pre definíciu štýlov. **Kostra** programu teda odporúča generovať namiesto aktuálneho riešenia, ktoré kostru len skopíruje zo svojho obsahu. Generovanie takéhoto projektu v rôznych verziách je však z hľadiska **kompatibility** knižníc ťažko realizovateľné.

6.2 Náprava zistených nedostatkov

Výstupom testovania prototypu boli odhalené **vady** a navrhované **vylepšenia** od respondentov.

6.2.1 Kontrola formátu zadávaných dát

Implementácia kontroly vpisovaných dát na strane prezentačnej vrstvy na základe použitého regulárneho výrazu sa zdala ako nedostatočná, pretože respondenti nepoužili pre odosielanie formulára element `input` typu `submit`, ako bolo pôvodne očakávané. Túto kontrolu sme teda re-implementovali s využitím Angular **ReactiveFormsModule**, ako je popísané v kapitole 5.1.3. Zatiaľ čo predošlá implementácia si vyžadovala len pridanie atribútu `pattern` do elementu `input`, v tomto prípade je nutné okrem špeciálneho atribútu vytvoriť objekt **FormControl** s príslušnými validátormi. Keďže tento objekt musí byť vytvorený v komponente,

a regulárny výraz vo validátore bude používateľ podľa potreby meniť, rovnako sa bude meniť aj konštrukcia tohto objektu. Podobne, ako v prípade služieb pre prácu s databázou, sme boli nútení oddeliť používateľskú a generovanú časť kódu pomocou hierarchie. To znamená, že sme museli dodatočne vytvoriť nové komponenty s príponou **Generated**, ktoré budú obsahovať premenné FormControl.

6.2.2 Implementácia generovania jednoduchých komponentov

Respondenti ďalej stratili významný čas pri implementovaní **tabuľky** pre zobrazovanie dát, čo však do určitej miery dokážeme automatizovať. Rozhodli sme sa teda pridať generovanie **vzorových** komponentov, ktoré zahŕňajú tabuľku, formulár, a tiež prvok výberu.

Problém pri implementovaní **tabuľky** nastáva pri zobrazovaní atribútov, ktoré predstavujú komplexné objekty. V tom prípade nie je jasné ktorú informáciu chce používateľ zobraziť, a preto budeme znázorňovať štandardne **identifikátor** daného referovaného objektu. To následne môže používateľ zmeniť. Spolu s HTML šablónou dokumentu generujeme aj logiku komponentu, ktorá pozostáva predovšetkým z **načítania** dát. Po použití vygenerovaného komponentu bude teda používateľovi hneď dostupná tabuľka s dátami danej triedy.

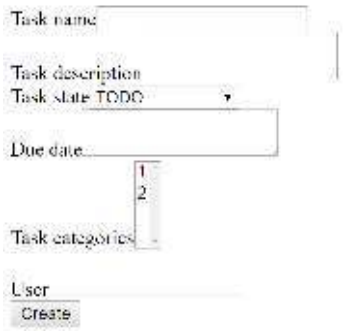
Task name	Task description	Task state	Due date	Task categories	User
Do dishes	Dont forget to do dishes	TODO	1	3	
Prepare presentation	Dont forget to add pictures	TODO	2	3	

Obr. 6.1: Jednoduchá vygenerovaná tabuľka

Okrem generovania komponentu tabuľky sme pridali aj generovanie komponentu **formulára**, s tlačidlom *Create* pre odoslanie dát. Tento komponent poskytuje funkcionalitu pre **vytvorenie** nového záznamu, spolu s **validáciou** vstupných dát a zobrazenia elementu s **chybovou** hláškou.

Pre vytvorenie formulára bolo nutné vytvoriť aj potrebné **select** komponenty, ktoré slúžia na vybratie referovaných objektov, prípadne zoznamu objektov pri vytváraní nových záznamov. Podobne ako v prípade komponentu tabuľky, predvolene zobrazujú databázové identifikátory daných objektov, pretože generátor nemá vedomosť o tom, ktorá informácia daného objektu môže najlepšie slúžiť ako identifikátor pre ľudského používateľa. To je možné následne zmeniť prepísaním

názvu atribútu v danej HTML šablóne vygenerovaného komponentu. Tieto komponenty nám následne môžu slúžiť nielen pre potreby formulára, ale napríklad aj pre vytvorenie filtrov.



Obr. 6.2: Jednoduchý vygenerovaný formulár

Vygenerované komponenty nevyzerajú používateľsky priateľsky, avšak je potrebné si uvedomiť, že je to zapríčenené absenciou štýlov, ktoré budú podľa očakávaní doplnené programátorom, prípadne dizajnérom.

6.2.3 Automatické spúštanie nadväzujúcich akcií

Pre zjednodušenie spúšťania vygenerovaných **projektov** sme zahrnuli do generátorov aj požadované akcie, ktoré zvyčajne predchádzajú spusteniu projektu. V prípade Springboot generátora po dokončení generovania vykonáme príkaz `mvn package`, ktorý stiahne potrebné závislosti a vytvorí archív JAR. V prípade Angular generátora je to spustenie skriptu `npm install`, a následne skriptu na generovanie GraphQL entít a služieb pomocou `npm run gql:codegen`. Výpisy z týchto príkazov samozrejme vidíme v okne výstupov daných generátorov. Vykonávanie daných skriptov je možné vypnúť v konfiguračnom súbore BFI.

7 Záverečné testovanie a vyhodnotenie použiteľnosti riešenia

V záverečnom testovaní sme po odhalení a náprave nedostatkov z prvej iterácie testovania vybrali **ďalšieho** respondenta, ktorému sme predostreli **totožný** scenár. Očakávame, že v rámci stanoveného času vyhradeného na testovanie v tomto prípade používateľ dokáže vytvoriť aplikáciu **rýchlejšie** a splniť aj druhú časť úlohy. Následne sa znovu opýtame na **názor** používateľa za pomoci interview, v ktorom budeme zisťovať **mieru použiteľnosti** nášho systému pre koncových používateľov.

7.1 Priebeh testovania a odozva používateľov

Narozdiel od testovania prvého prototypu, tentokrát respondent s porovnateľnými znalosťami dokázal úspešne splniť **obe** časti scenára. Generovanie jednoduchých komponentov výrazne **urýchlilo** implementovanie celého systému. Dokázal bez komplikácií úspešne definovať model, vygenerovať obe časti aplikácie, spustiť serverovú časť pomocou *docker compose*, a vytvoriť jednoduchú prezentačnú vrstvu spĺňajúcu podmienky testu. Zobrazovala ľudsky zrozumiteľnú tabuľku známok, zrozumiteľný formulár a možnosť filtrovania zobrazovaných dát podľa zvoleného predmetu. V prvej časti zvolil naplnenie databázy pomocou webového grafického rozhrania GraphQL, čo sa mu podarilo napriek absencii predošlých skúseností s týmto systémom.

Čo sa týka dojmov používateľa z testovania, negatívne hodnotil veci ako spomínané generovanie UI komponentov, pretože riešenie sa mu zdalo príliš špecifické, vyhotovené pre daný scenár. Z nášho pohľadu však používateľ nie je viazaný tieto prvky využiť, a preto do budúcna budeme uvažovať skôr o doplnení mož-

nosti pre vypnutie tejto funkcionality. Pri porovnávaní systému **GraphQL** so systémom OpenAPI pre špecifikovanie REST rozhraní hodnotil pozitívne jednoduchosť dokumentu, ktorý tieto rozhrania definuje. Systém sa stále javí ako mierne **nestabilný** v dôsledku defektov, s ktorými sa používateľ stretol v priebehu práce s ním. Tak isto by si systém vyžadoval kvalitnejšiu **dokumentáciu**. Používateľ použil pre spustenie serverovej časti nástroj Docker compose a podpora **Devops** nástrojov sa tým pádom ukázala ako prospešná. Stretli sme sa s pozitívnym ohlasom ku **konceptu** aplikácie, avšak nedostatočné sa ukázalo najmä jej prevedenie. Zaujímavou pripomienkou bolo porovnanie tohto generatívneho postupu s postupom IDE v procese automatickej **refaktorizácie**. Ak teda používateľ pracuje s triedami, ktorých pôvod je v modeli BFI, pri premenovaní atribútov generátor nedisponuje vyhľadávaním a automatickou úpravou referencií, ako v prípade refaktorizácie pomocou IDE. Generatívny prístup však prináša výhodu v zmysle premenovania tohto atribútu naprieč projektami, avšak bez premenovania referencií v používateľovom špecifickom kóde. Doplnenie tejto funkcionality je realizovateľné, avšak obtiažne, aj v prípade, že by sme chceli využiť dostupnú funkcionalitu prostredia.

7.2 Vyhodnotenie záverečného testovania

Vo vyhodnotení najprv zhrnieme jednotlivé časti systému, ktoré sme navrhovali a voľby, ktoré sme uskutočnili, a zhodnotíme ich správnosť.

GraphQL Z pohľadu vývoja nových generátorov bolo kľúčové **ujednotiť** komunikáciu pomocou jednoducho definovateľného rozhrania, čo nám GraphQL umožnilo. Ako prijateľná alternatíva k REST sa javí aj používateľom. Nevýhodou bolo, že je tento štandard **málo rozšírený** a nikto z respondentov s ním nemal praktické skúsenosti. Napriek tomu, že svet informačných technológií sa považuje za dynamický, softvéroví vývojári len s ťažkosťami nahrádzajú staré postupy a technológie novými, najmä ak sú staré zvyky postačujúce. Respondenti teda váhali pri otázke, či by pri vývoji novej aplikácie chceli namiesto zaužívaného REST použiť systém GraphQL.

DevOps V rámci práce sme pridali podporu pre systémy Docker, Gitlab, Docker swarm, a Kubernetes vo forme jednotlivých definičných súborov. Dvaja z troch

používateľov **použili** Docker pre spustenie serverovej časti aplikácie, čím sa vyhli možným problémom s nekompatibilitou prostredia. Tak isto potvrdili, že tieto systémy používajú v zamestnaní a ich konfiguráciu pri tomto type aplikácie kopíruju naprieč projektami. Preto túto podporu **uvítali**.

Generátory Stretli sme sa s pozitívnym ohlasom najmä na spôsob **spúštania** generovania prostredníctvom akcii priamo v IDE. Tento spôsob bol dostatočne **intuitívny** a bezproblémový. Po vyriešení problémov v generovanom kóde, ktoré sme odhalili v prvej iterácii testovania bol generovaný kód ihneď spustiteľný. Čo sa týka výsledných aplikácií, je veľký **priestor** na zlepšovanie ich implementácie. Jeden z generátorov nebol používateľmi otestovaný, a mal slúžiť najmä pre demonštráciu možnosti generovať aj aplikácie iného charakteru ktoré vychádzajú z daného modelu. Išlo o generátor nástroja Liquibase, ktorý je schopný vygenerovať SQL skript pre migrovanie databázovej štruktúry. Podľa vyjadrení používateľov ide o reálny problém, ktorý tento generátor dokáže **riešiť** a vedia si predstaviť jeho použitie.

Koncept Vo všeobecnosti bol pokladaný systém za prospešný z hľadiska **ušetrenej** práce, a používatelia si dokážu predstaviť aj jeho použitie pri evolúcii aplikácie, teda nielen pre prvotné vyhotovenie prototypu, ale aj jeho následné využívanie pri následnom rozširovaní v **produkčnom** prostredí.

8 Záver

Pri interview sa ukázalo, že programátori nepoužívajú a majú **slabý prehľad** o možnostiach generovania kódu. Sú zvyknutí implementovať jednotlivé častí systému bez presného definovania rozhraní, ktoré si odovzdávajú len formou komunikácie, prípadne dokumentácie. Naša práca ukazuje **koncept**, ako by sa mohlo generatívne programovanie stať používateľsky **prijateľnejšie** vďaka jeho jednoduchej dostupnosti a použiteľnosti.

Podobne ako prevládali prvé pocity pri nástupe cloudu, používatelia váhajú aj v tomto prípade. Používatelia cítia menší pocit **kontroly** nad ich prácou. Prevláda strach z faktu, že sa generovaný kód nebude správať podľa ich predstáv, a nebudú disponovať nástrojmi, ako tento problém vyriešiť. Všeobecne naše riešenie rovnako ako v predošlom stave stále trpí množstvom malých **nedostatkov**, ktoré prispeli k jeho negatívnemu obrazu.

Používatelia však pomocou neho napriek tomu v relatívne krátkom čase dokázali implementovať jednoduchú webovú aplikáciu vrátane kontroly prístupu v relatívne krátkom čase. To naznačuje, že takýto prístup k jej tvorbe je **efektívny**, a množstvo práce, ktorú je potrebné pri vývoji vykonať, je možné **zautomatizovať**. Prínosom je, že prácu sa nám podarilo **verejne** sprístupniť prostredníctvom obchodu s rozšíreniami vo vývojových prostrediach spoločnosti Jetbrains.

Naviac sme zaviedli istú formu **modularity**, vďaka ktorej si vývojári dokážu jednoducho vytvoriť vlastné generátory a použiť ich pridaním odkazu do konfiguračného súboru. Pre implementovanie generátora je v zásade potrebné použiť knižnicu a vytvoriť triedu, ktorá implementuje rozhranie *Generator*.

Pri vývoji sme teda do úvahy brali nielen okamžitý prínos pre programátorov, ale snažili sme sa na projekt pozerať aj z perspektívy jeho **životaschopnosti** po skončení tejto práce. Z tohto pohľadu projektu chýba verzionovanie jadra, respektíve modelu systému, a riešenia pre zabezpečenie spätnej kompatibility so staršími

generátormi. Aj napriek tomu, že šance na prežitie takéhoto projektu sú malé, spočíva prínos tejto práce v **demonštrácii** potenciálu generatívneho programovania v oblasti webových aplikácií. Ukázali sme, že oproti ostatným podobným systémom, ako napríklad Swagger, je možné vychádzať priamo z **doménového** modelu aplikácie namiesto modelu **rozhrania**, čo nám umožní zájsť ďalej v oblasti funkcionalít, ktoré potencionálne dokážeme generovať. Tento prístup nám umožnil generovať aj nepriamo súvisiace časti systému, ako je nástroj pre generovanie databázových migračných skriptov. Systém je potencionálne možné použiť aj pre čiastočné generovanie **natívnych** aplikácií, ktoré taktiež využívajú rovnaké prístupové body pre prácu s dátami ako webové rozhrania.

Bibliografia

[1] Vincent Horváth. *Doménovo-špecifický jazyk pre vývoj webových aplikácií*. 2019.

[2] Trygve Reenskaug. „Mvc xerox parc 1978-79“. In: *Trygve/MVC* (1979).

[3] Google Trends. *Vue.js,React,Angular*. Máj 2019. URL: https://trends.google.com/trends/explore?cat=31%5C&date=today%205-y%5C&q=Vue.js,React,Angular (cit. 15. 05. 2019).

[4] Hackerrank. *HackerRank 2019-2018 Developer Skills Report*. 2019. URL: https://info.hackerrank.com/rs/487-WAY-049/images/HackerRank_2019-2018_Developer-Skills-Report.pdf (cit. 15. 05. 2019).

[5] *JavaScript 2019 - The state of Developer Ecosystem in 2019 Infographic*. URL: https://www.jetbrains.com/lp/devecosystem-2019/javascript/.

[6] Alejandro Ugarte. *Building a Web Application with Spring Boot and Angular*. Mar. 2019. URL: https://www.baeldung.com/spring-boot-angular-web (cit. 15. 05. 2019).

[7] Bobby Brennan. „Libraries vs. Frameworks“. In: *Medium* (2017). URL: https://medium.com/datafire-io/libraries-vs-frameworks-626cdde799a7 (cit. 02. 06. 2019).

[8] Cory Gackenheimer. *Introduction to React*. Apress, 2015.

[9] Sergiy Pylypets. „Consuming REST APIs With React.js“. In: (apr. 2019). URL: https://www.baeldung.com/spring-boot-angular-web (cit. 02. 06. 2019).

[10] Stefan Krause. URL: https://www.stefankrause.net/js-frameworks-benchmark7/table.html (cit. 02. 06. 2019).

[11] Ibrahim Šuta. „React.js and Spring Data REST“. In: (aug. 2018). URL: https://developer.okta.com/blog/2018/08/27/build-crud-app-vuejs-netcore (cit. 02. 06. 2019).

[12] William Young a Janice Platt. *Database migration*. US Patent App. 09/922,032. Feb. 2003.

[13] E. Yuan a J. Tong. „Attributed based access control (ABAC) for Web services“. In: *IEEE International Conference on Web Services (ICWS'05)*. Júl 2005, s. 569. DOI: 10.1109/ICWS.2005.25.

[14] Vincent C Hu, David Ferraiolo a D Richard Kuhn. *Assessment of access control systems*. US Department of Commerce, National Institute of Standards a Technology, 2006.

[15] David Ferraiolo, Janet Cugini a D Richard Kuhn. „Role-based access control (RBAC): Features and motivations“. In: *Proceedings of 11th annual computer security application conference*. 1995, s. 241–48.

[16] Steven Tuecke et al. „Globus Auth: A research identity and access management platform“. In: *2016 IEEE 12th International Conference on e-Science (e-Science)*. IEEE. 2016, s. 203–212.

[17] Pedro A Castillo et al. „SOAP vs REST: Comparing a master-slave GA implementation“. In: *arXiv preprint arXiv:1105.4978* (2011).

[18] Brian Mulloy. *Web API design*. 2013.

[19] Robert Wideberg. *RESTful Services in an Enterprise Environment: A Comparative Case Study of Specification Formats and HATEOAS*. 2015.

[20] Erik Wittern, Alan Cha a Jim A Laredo. „Generating GraphQL-Wrappers for REST (-like) APIs“. In: *International Conference on Web Engineering*. Springer. 2018, s. 65–83.

[21] Google Trends. *GraphQL, Swagger, RAML*. Dec. 2019. URL: https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11cn3w0w9t,%2Fg%2F11g9nh3stq,RAML (cit. 08. 12. 2019).

[22] Christof Ebert et al. „DevOps“. In: *Ieee Software* 33.3 (2016), s. 94–100.

[23] Rajdeep Dua, A Reddy Raja a Dharmesh Kakadia. „Virtualization vs containerization to support paas“. In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, s. 610–614.

[24] Margaret Rouse a Alex Gillis. *What is Docker image? - Definition from WhatIs.com*. Júl 2018. URL: https://searchitoperations.techtarget.com/definition/Docker-image.

[25] M. Shahin, M. Ali Babar a L. Zhu. „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“. In: *IEEE Access* 5 (2017), s. 3909–3943. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2685629.

Zoznam príloh

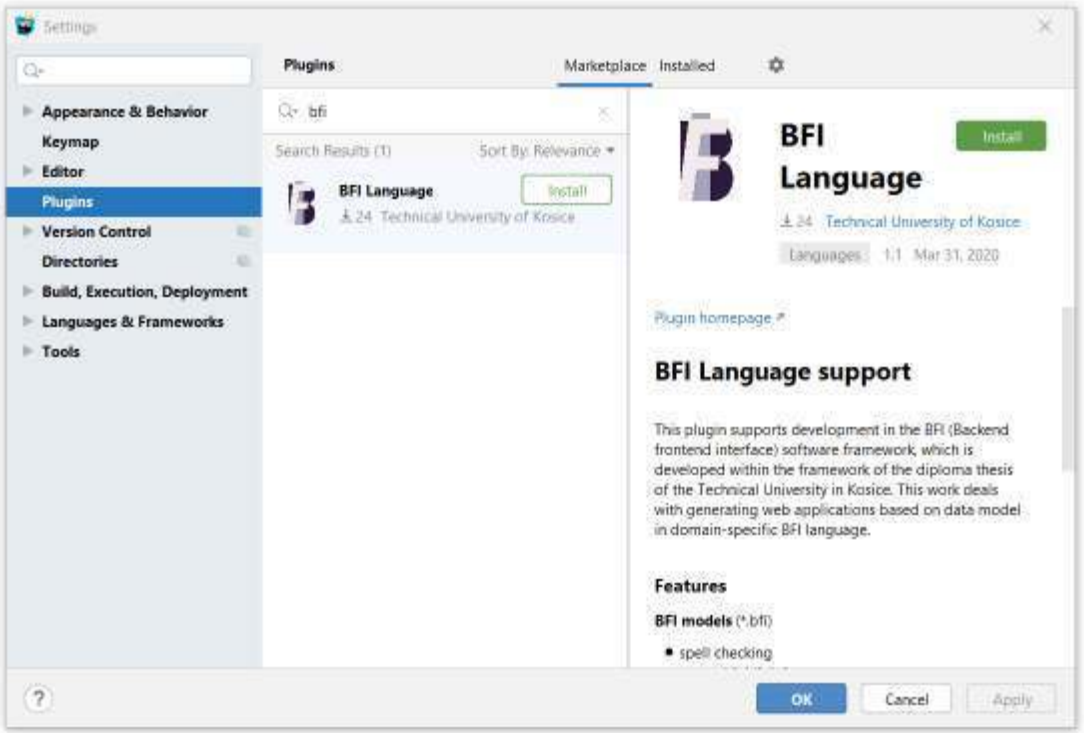
- **Príloha A** Používateľská príručka
- **Príloha B** Systémová príručka
- **Príloha C** CD médium - diplomová práca v elektronickej podobe, prílohy v elektronickej podobe, aplikácia BFI generátor, aplikácia BFI Language plugin, vzorové aplikácie

A Používateľská príručka

A.1 Inštalácia nástroja

Inštalácia nástroja prebieha prostredníctvom integrovaného obchodu z rozšíreniami v prostrediach Jetbrains. Rozšírenie je kompatibilné so všetkými prostrediami novšími, ako verzia 193, označovaná aj ako 2019.3.

install.png



Obr. A.1: Inštalácia rozšírenia

Po inštalácii rozšírenia bude IDE schopné rozpoznať súbory *.bfi a *.bfi.yaml.

A.2 Definovanie doménového modelu

Pre definovanie modelu je nutné vytvoriť aspoň jeden súbor s príponou *.bfi. Tento súbor je napísaný v špeciálnom doménovo špecifickom jazyku BFI. Príklad obsahu takéhoto súboru môžete vidieť na obrázku A.20.

```
1  model Task
2    name:string label="Task name" editor=string required matches="[A-Za-z
     ↪  ]{1,20}" min=1 max=20
3    description:string label="Task description" editor=textarea virtual
4    state:State label="Task state" editor=select
5    dueDate:string
6    categories:Category[] label="Task categories" editor=multiselect
7    user:User
8  model Category
9    name:string label="Category name" editor=string required
     ↪  matches="[A-Za-z ]{1,20}" min=1 max=20
10   tasks:Task[] label="Tasks"
11 model User
12   name:string
13 enum State [Todo,InProgress,Done]
14 role User
15   can manage Task where .user.name: #username
16   can read Category
17   can read User where .name: #username
18 role Administrator
19   can manage Category
20   can manage User
```

Zdrk. A.20: Príklad BFI súboru

Vidíme, že tento súbor sa skladá z 3 hlavných prvkov, ktorými sú **model**, **enum** a **role**. Prvok **model** definuje štruktúru určitej dátovej entity. Prvok **enum** definuje enumeračný typ, teda typ, ktorý má pevne dané pole hodnôt. Prvok **role** definuje používateľskú rolu a jej právomoci pomocou modelu ABAC.

```
model <model name>
  <field name>:<field type> label="<label>" editor=<editor type>
  ↪ matches="<pattern>" min=<minimum> max=<maximum> <flags>
```

Zdrk. A.21: Štruktúra definície modelu

Model je zložený z ľubovoľného počtu atribútov. Popis jednotlivých prvkov modelu A.21:

- **model name** (povinné) meno modelu,
- **field name** (povinné) meno atribútu modelu,
- **field type** (povinné) typ atribútu, môže nadobúdať hodnotu:
  - **string** krátky textový reťazec´,
  - **int** číselná hodnota,
  - **text** dlhý textový reťazec,
  - ***referencia*** referencia na iný typ pomocou jeho mena,
- **pattern** regulárny výraz, ktorý musí spĺňať hodnota daného atribútu,
- **minimum** minimálna dĺžka, respektíve minimálna hodnota atribútu,
- **maximum** maximálna dĺžka, respektíve maximálna hodnota atribútu,
- **flags** značky daného atribútu oddelené čiarkou, môžu byť,
  - **required** hodnota tohto objektu nesmie byť nikdy prázdna,
  - **hidden** *aktuálne nepoužívaná*,
  - **virtual** *aktuálne nepoužívaná*.

```
enum <enum name> [<enum values>]
```

Zdrk. A.22: Štruktúra definície enumeračného typu

Predpis enumeračného typu sa skladá z hodnôt v tvare *CamelCase* oddelených čiarkou.

```
role <role name>
  can <role action> <model name>
  can <role action> <model name> where <field path>: <field value>
```

Zdrk. A.23: Štruktúra definície roly

Rola je zložená z ľubovoľného počtu schopností, ktoré definujú čo používateľ s danou rolou môže vykonávať.

- **role name** (povinné) meno roly,
- **role action** (povinné) akcia, ktorú rola môže vykonať pre daný **model name**, môže nadobúdať hodnoty:
  - **read** povolenie čítania,
  - **write** povolenie zapisovania,
  - **delete** povolenie mazania,
  - **manage** povolenie všetkých akcií,
- **role action**
- **model name** (povinné) meno modelu pre ktorý je možné vykonať akciu **role action**,
- **field path** cesta k atribútu z daného modelu oddelená a začínajúca sa bodkou, ktorý vstupuje do podmienky pre platnosť schopnosti roly,
- **field value** hodnota atribútu, ktorá vstupuje do podmienky pre platnosť roly, môže nadobúdať hodnoty:
  - *"reťazec"* konštanta vo forme textového reťazca,
  - **username** špeciálny atribút reprezentujúci meno používateľa obsiahnuté v autorizačnom kľúči.

A.3 Konfiguračný súbor

Po zadefinovaní modelu je nutné vytvoriť konfiguračný súbor s príponou „.bfi.yaml“. Konfiguračný súbor A.24 je využívaný na zadanie vlastností projektu, definovanie generátorov, ktoré chce používateľ použiť, a spúštanie generátorov.

```yaml
name: test-todo
destination_path: C:\Project\Java\test-todo\generated
database:
  url: ./test
  schema: data
```

```
6     user: root
7     pass: root
8     driver: h2
9   generators:
10    - generatorName: springboot
11      groupId: sk.tuke.ales.bulko
12      artifactId: bfi-example-springboot
13      runMvnPackage: true
14      port: 8080
15    - generatorName: angular
16      projectName: bfi-example-angular
17      backendUrl: http://localhost:8080
18      backendDevUrl: http://localhost:8080
19    runNpmInstall: false
20    runGraphqlCodegen: true
21  kubernetes:
22    docker-repository:
23      server-url: git.kpi.fei.tuke.sk:4567
24      username: gitlab-ci-token
25      password: tfa5cas864fgas88c9sd
```

Zdrk. A.24: Príklad BFI konfigurácie

Skladá sa z:

- **name** (povinné) meno projektu
- **destination_path** cesta k adresáru kde bude projekt vygenerovaný, predvolene je to koreňový adresár, kde je umiestnený tento definičný súbor,
- **source_path** cesta k adresáru kde sa nachádzajú definičné súbory modelov, predvolene je to koreňový adresár, kde je umiestnený tento definičný súbor,
- **database** (povinné) definuje databázu, ktorú majú aplikácie používať, obsahuje:
  - **driver** typ databázy, aktuálne podporované „mysql“ a „h2“
  - **url** (povinné)adresa databázy,
  - **schema** (povinné) názov schémy,
  - **user** (povinné) používateľské meno pre prihlásenie,

  - **pass** (povinné) používateľské heslo pre prihlásenie,

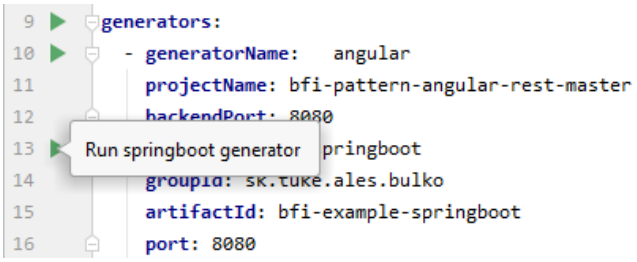
- **generators** (povinné) definuje aké generátory budú použité (popísané nižšie v paragrafe *generators* A.3)
- **kubernetes.docker-repository** prihlasovacie údaje do privátneho docker repozitára, pre účely vygenerovanie Kubernetes Secrets objektu, ktorý po nasadení slúži kubernetesu pre stiahnutie image

***generators** Generátory je možné špecifikovať dvoma spôsobmi:

- **generatorName** definuje meno generátora, aktuálne podporované „springboot“, „angular“, „liquibase“
- **customGenerator** definuje vlastný generátor pomocou
  - **uri** URI adresa kde sa nachádza .jar generátora,
  - **classpath** cesty ku triede, ktorá implementuje rozhranie *Generator*.

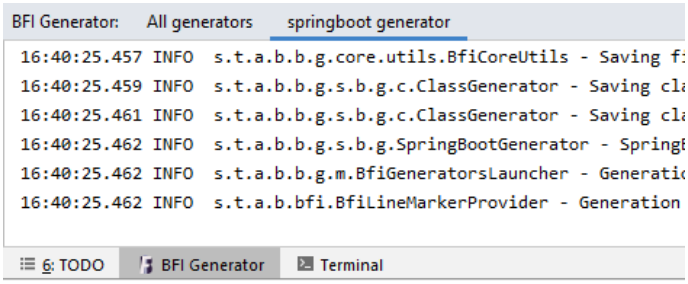
A.4 Spúšťanie generátorov

Generátor je možné spustiť pomocou akcie v konfiguračnom súbore. Táto akcia sa aktivuje kliknutím na zelenú ikonu spustenia na riadku, kde je definovaný generátor. Je možné spustiť naraz všetky generátory, alebo aj jednotlivo.



Obr. A.2: Spustenie generátora Springboot

Po spustení sa objaví v dolnej lište okno s názvom *BFI Generator*, kde bude zobrazovaný výstup generátora. Toto okno obsahuje karty s názvom akcie, ktorou boli spustené.



Obr. A.3: Okno nástroja BFI

A.5 Springboot aplikácia

Súborová štruktúra výslednej aplikácie je nasledovná.

```
\---main
    +---java
    |   \---sk.tuke.ales.bulko.bfi.example.springboot
    |                                +---config
    |                                +---graphql
    |                                |   +---mutation
    |                                |   |   \---generator
    |                                |   \---query
    |                                |       \---generated
    |                                +---model
    |                                |   +---entity
    |                                |   \---enums
    |                                \---repository
    |                                    \---generated
    \---resources
        |   application.properties
        |   schema.sql
        \---graphql
                definition.graphqls
```

Zdrk. A.25: Súborová štruktúra Springboot

Triedy typu mutation, query a repository je možné pomocou dedičnosti doplňať o špecifickú funkcionalitu. V prípade súboru *application.properties* je možné špecifický kód vpisovať do bloku ohraničeného komentármi *user config start* a *user*

config end. Po zadaní položky do tohto bloku sa už rovnaká položka mimo nevygeneruje.

A.6 Angular aplikácia

Návod pre spustenie aplikácie sa nachádza v súbore *README.md* v koreňovom adresári vygenerovaného projektu. Podobne ako v prípade Springboot aplikácie, je možné jednotlivé triedy pomocou dedičnosti rozširovať. Súbor environment.ts je možné rozširovať pomovou vpisovania do bloku ohraničeného komentármi *#user config start* a *#user config end*. V prípade Angular komponentov si môžete všimnúť 2 HTML súbory s príponou **.bfi.html* a **.generated.html*. Súbor **.bfi.html* obsahuje HTML kód, v ktorom môžu byť použité špeciálne atribúty rámca BFI, a ten je následne preložený do súboru **.generated.html*. Pre vypnutie tejto funkcionality v prípade, že chcete používať súbor **.generated.html* je možné zmazať komentár v prvom riadku tohto súboru.

A.6.1 Využívanie šablón pre generovanie pohľadov

Pre klientskú časť aplikácie je možné využívať HTML šablóny, pričom autor pre tento účel zaviedol špeciálne XML značky. V procese generovania generátor vyhľadá všetky šablóny (súbory s príponou *.bfi.html*) a následne vygeneruje HTML kód s nahradenými špeciálnymi značkami.

Značky na úrovni atribútov modelov

Nasledujúce špeciálne XML značky sa viažu na niektorý z atribútov modelu špecifikovaného pomocou BFI. Patria sem:

- **bfi-input** element, ktorý prijíma **vstup** podľa špecifikovaného editora pre daný atribút objektu,
- **bfi-output** element pre **zobrazovanie** hodnoty danej vlastnosti,
- **bfi-label** element pre zobrazenie **popisu** uvedeného vo vlastnosti label daného objektu.

Keďže sú tieto značky sú viazané na atribúty modelov, vyžadujú tieto atribúty:

- **bfi-model** názov modelu objektu,

- **bfi-field** názov atribútu objektu.

Zobrazovanie tabuľky

Pre zobrazenie tabuľkovej reprezentácie dát jednotlivých objektov je možné využiť značku **bfi-tb**. Je možné ju používať analogicky so značkou **table** jazyka HTML a poskytuje teda nasledujúce alternatívne značky.

- **bfi-tb-header** zaobaľuje elementy definujúce hlavičku tabuľky,
- **bfi-tb-body** zaobaľuje elementy definujúce telo tabuľky
- **bfi-tb-th** zaobaľuje elementy hlavičky tabuľky
- **bfi-tb-tr** zaobaľuje elementy riadku tela tabuľky
- **bfi-tb-td** definuje dátovú bunku, prípadne jej názov v prípade použitia v hlavičke (využíva parametre **bfi-model** a **bfi-field**)

Použitie môžeme vidieť na príklade A.26.

```
1  <bfi-tb>
2      <bfi-tb-header>
3          <bfi-tb-th>
4              <bfi-tb-td bfi-model="Task" bfi-field="name"></bfi-tb-td
5              <bfi-tb-td bfi-model="Task" bfi-field="state"></bfi-tb-td>
6              <bfi-tb-td>Actions</bfi-tb-td>
7          </bfi-tb-th>
8      </bfi-tb-header>
9      <bfi-tb-body>
10         <bfi-tb-tr>
11             <bfi-tb-td bfi-model="Task" bfi-field="name"></bfi-tb-td>
12             <bfi-tb-td>{{task.state}}</bfi-tb-td>
13             <bfi-tb-td>
14                 <button (click)="showTaskReader(task)">Show</button>
15             </bfi-tb-td>
16         </bfi-tb-tr>
17     </bfi-tb-body>
18 </bfi-tb>
```

Zdrk. A.26: Príklad vytvorenia BFI tabuľky

B Systémová príručka

Táto príručka slúži pre implementovanie nových generátorov pre rozšírenie softvérového rámca BFI.

B.1 Implementácia nového generátora

Pre implementáciu nového generátora sú potrebné tieto kroky:

1. **stiahnutie** a nainštalovanie maven závisloti z repozitára `git@git.kpi.fei.tuke.sk:master-thesis-bfi/bfi-generator.git`,
2. pridanie **rodičovského** modulu podľa aktuálnej verzie stiahnutej knižnice (zdrk. B.27),
3. pridanie **závislosti** na modul core-semantics, alebo modul generator-utils (zdrk. B.28) (popis v kapitole 4.1),
4. **implementácia** rozhrania `sk.tuke.ales.bulko.bfi.generator.core.semantics.component.Generator` .

```
1    <parent>
2        <artifactId>bfi-generator-new</artifactId>
3        <groupId>sk.tuke.ales.bulko</groupId>
4        <version>${bfi.version}</version>
5    </parent>
```

Zdrk. B.27: Pridanie rodičovského modulu

```
1    <dependency>
2        <groupId>sk.tuke.ales.bulko</groupId>
3        <artifactId>generator-utils</artifactId>
4    </dependency>
```

Zdrk. B.28: Pridanie závislosti

Rozhranie **Generator** obsahuje jedinú metódu

```
void generate(GeneratorContext context)
```

, ktorá ako parameter obsahuje kontext generátora. Tento kontext je vypĺňaný pri spustení generovania podľa zdrojových súborov, teda modelu a konfigurácie. Skladá sa z:

- **projectContext** Objekt, ktorý obsahuje kontext celého projektu.
- **generatorProperties** Obsahuje premenné, určené pre daný generátor.

Premenná `projectContext` je tvorená:

- **models** Modely definované v súbore *.bfi.
- **roles** Roly definované v súbore *.bfi, alebo v súbore *.bfi.yaml.
- **enums** Enumeračné typy definované v súbore *.bfi.
- **generatorConfigs** Konfigurácie všetkých použitých generátorov.
- **kubernetesConfig** Konfigurácia potrebná pre systém Kubernetes.
- **databaseConfig** Konfigurácia pripojenia na databázu, ktorá by mala byť použitá v serverovej časti aplikácie.
- **runConfiguration** Informácie viažúce sa na spustenie procesu generovania, kde zatiaľ patrí len cesta ku konfiguračnému súboru.
- **name** Meno projektu.
- **targetPath** Cesta, kde majú byť aplikácie vygenerované.

Implementovaný generátor je možné testovať pomocou atribútu konfiguračného súbora *customGenerator* (kapitola A.3). Po dokončení generátora je potrebné autora práce pre pridanie generátora do oficiálne podporovaných generátorov prostredníctvom aktualizácie súboru so zoznamom generátorov umiestnenom na internetovom úložisku na adrese `https://s3.eu-west-3.amazonaws.com/bfi.framework/generator/generators-definition.yaml`.

B.1.1 Užitočné funkcie jadra

Súčasťou jadra sú funkcionality, ktoré môžu byť užitočné pri implementácii nového generátora.

BfiCoreUtils.java

Táto trieda obsahuje pomocné metódy:

- `List<File> findByExtension(Path searchPath, String extension)`
  - metóda slúži na vyhľadanie súborov podľa ich prípony v danom adresári, do úrovne 3 zložiek
- `Optional<Field> getBackreferenceField(Model source, Model target)`
  - metóda slúži na získanie atribútu obsahujúceho referenciu z modelu `target`, na model `source`, ak taká existuje
- `String transform(String value, Transformation... transformations)`
  - metóda slúži na transformovanie reťazca s použitím Transformátorov, popísaných nižšie
- `String indent(String multiRowValue, int indentTabs)`
  - metóda slúži na odsadenie textu `multiRowValue` zvoleným počtom tabulátorov `indentTabs`
- `String toString(InputStream inputStream)`
  - metóda slúži na prekonvertovanie vstupného prúdu údajov `inputStream` na textový reťazec
- `Optional<String> getProjectFile(Path projectPath, String... path)`
  - metóda slúži na získanie obsahu súboru, určeného relatívnou cestou `path` vzhľadom k `projectPath`
- `void rewriteFile(String content, Path projectPath, String... path)`
  - metóda slúži na prepísanie súboru určeného relatívnou cestou `path` vzhľadom k `projectPath` obsahom `content`

- `void saveIfNotExists(String content, Path projectPath, String... path)`
  - metóda slúži na uloženie súboru (ak neexistuje) určeného relatívnou cestou `path` vzhľadom k `projectPath` obsahom `content`
- `Field getFieldByName(Model model, String fieldName)`
  - metóda slúži na získanie atribútu modelu `model` podľa jeho mena `fieldName`
- `Model getModelByName(List<Model> models, String modelName)`
  - metóda slúži na získanie modelu z poľa `models` podľa jeho mena `modelName`
- `Enumeration getEnumByName(List<Enumeration> enums, String enumName)`
  - metóda slúži na získanie enumeračného typu z poľa `enums` podľa jeho mena `enumName`
- `void unzip(InputStream zippedFile, Path unzipLocation)`
  - metóda slúži na rozbalenie obsahu prúdu `zippedFile`, ktorý obsahuje archív formátu zip, do lokácie určenej cestou `unzipLocation`
- `int runCmd(Path projectPath, String... command)`
  - metóda slúži na spustenie príkazu `command` v zložke `projectPath` s návratovou hodnotou

DownloadUtils.java

- `String getUrlContent(URI uri, CachePreferences cachePreferences)`
  - metóda slúži na získanie obsahu z URI adresy `uri`, ktorého obsah môže byť dočasne uschovaný podľa preferencií `cachePreferences`
- `File download(URI uri, CachePreferences cachePreferences)`
  - metóda slúži na stiahnutie súboru z URI adresy `uri`, ktorého obsah môže byť dočasne uschovaný podľa preferencií `cachePreferences`

BfiCorePredicates.java

Obsahuje funkcie pre overovanie určitých podmienok pre typy z BFI modelu.

- isModel - overenie či je typ modelom
- isEnum - overenie, či je typ enumeračný
- isCollection - overenie, či je typ kolekcia
- isModelCollection - overenie, či je typ kolekcia modelov
- isEnumCollection - overenie, či je typ kolekcia enumeračných typov

Transformations.java

Je to enumeračný typ obsahujúci rôzne funkcie pre transformáciu textu:

GraphQLUtils.java

Táto trieda obsahuje pomocné metódy pre prácu s GraphQL.

- `String toGraphQlFieldType(Type type)`
  - metóda slúži na konverziu typu `type` na GraphQL typ
- `String toGraphQlRefType(Type type)`
  - metóda slúži na konverziu typu `type` na GraphQL typu, avšak v prípade že ide o referenciu, vráti typ Int

GraphQLDocumentGenerator.java

Táto trieda slúži pre generovanie GraphQL dokumentov.

- `void generateDocuments(String... relativePathToFolder)`
  - metóda slúži na vygenerovanie GraphQL dokumentov do zložky `relativePathToFolder`, ktorá je relatívna k ceste vygenerovaného projektu
- `void generateDocument(Model model, String... pathArray)`

  - metóda slúži na vygenerovanie GraphQL dokumentu modelu `model` do zložky `relativePathToFolder`, ktorá je relatívna k ceste vygenerovaného projektu

GraphQLSchemaGenerator.java

Táto trieda slúži pre generovanie GraphQL schémy.

- `void generateSchema(String... relativePath)`
  - metóda slúži na vygenerovanie súboru so schémou do zložky `relativePath`, ktorá je relatívna k ceste vygenerovaného projektu