Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

# Bakalářská práce

# Vizuální konfigurace testů API

Plzeň 2018

Petr Volf

# Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 3. května 2018

Petr Volf

# Abstract

The topic of this bachelor thesis focuses on creation of specific application for dealing with visual configuration of testing application interfaces (APIs). The main goal of this application is to facilitate the definition of API testings using visual tools and to offer alternatives to already existing practices. The thesis compares already existing applications, highlights their disadvantages and proposes solutions. When designing a solution, main focus is aimed on simplicity and usability as the main mandatory character traits of the final version. The second part describes the development and implementation of this application. The work is implemented in JavaScript, React, TypeScript, NodeJS, Elasticsearch, Redis and PostgreSQL.

# Abstrakt

Tato bakalářská práce se zabývá vytvořením aplikace pro vizuální konfiguraci testů aplikačních rozhraní (zkr. API). Cílem této aplikace je usnadnit definici testů API pomocí vizuálních nástrojů a nabídnutí alternativy ke stávajícím řešením. Práce porovnává tyto aplikace, upozorňuje na jejich nevýhody a nabízí možnosti řešení těchto problémů. Při navrhování řešení je kladen důraz na jednoduchost a přehlednost, kterou by výsledná aplikace měla obsahovat. Druhá část práce popisuje vývoj a implementaci tohoto programu. Práce je implementována v technologiích JavaScript, React, TypeScript, NodeJS, Elasticsearch, Redis a PostgreSQL.

# Obsah

**1 Úvod** **7**

**2 Stávající řešení** **8**
2.1 Postman . . . 8
2.2 Runscope . . . 9
2.3 Assertible . . . 9

**3 Volba technologií** **10**
3.1 Analýza požadavků . . . 10
3.2 Single page application . . . 10
3.3 Serverová část . . . 11
3.4 Klientská část . . . 12
3.4.1 React . . . 12
3.5 TypeScript . . . 15
3.6 Databázové systémy . . . 16
3.6.1 PostgreSQL . . . 16
3.6.2 Redis . . . 16
3.6.3 Elasticsearch . . . 16

**4 Architektura** **18**
4.1 Popis architektury . . . 18
4.2 Databáze . . . 18
4.2.1 Tabulka User . . . 19
4.2.2 Tabulka App . . . 20
4.2.3 Tabulka Scenario . . . 20
4.2.4 Tabulka Test . . . 20
4.2.5 Tabulka TestDefinition . . . 20
4.3 Definice testů . . . 21
4.3.1 Struktura volání . . . 21
4.3.2 Struktura pravidel odpovědi . . . 21

**5 Implementace** **24**
5.1 Struktura projektu . . . 24
5.2 Společný modul . . . 24
5.2.1 Konfigurace . . . 24
5.2.2 Definice relační databáze . . . 26

5.3 Klientská část . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.3.1 Redux . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.3.2 Diagram scénáře . . . . . . . . . . . . . . . . . . . . . 33
5.3.3 Routování . . . . . . . . . . . . . . . . . . . . . . . . 35
5.3.4 TSLint . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.4 Serverová část . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.4.1 ExpressJS . . . . . . . . . . . . . . . . . . . . . . . . 38
5.4.2 Odchytávání chyb . . . . . . . . . . . . . . . . . . . . 39
5.4.3 Autentifikace uživatele . . . . . . . . . . . . . . . . . 42
5.4.4 Dotazy do relační databáze . . . . . . . . . . . . . . . 42
5.5 Scheduler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.6 Worker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.6.1 Provádění testů . . . . . . . . . . . . . . . . . . . . . 44
5.6.2 Uložení výsledků . . . . . . . . . . . . . . . . . . . . . 46
5.7 Logování chyb . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**6 Závěr** **48**

**Literatura** **49**

**Uživatelský manuál** **51**

**Seznam obrázků** **52**

**Seznam algoritmů** **53**

# Kapitola 1

# Úvod

Trendem dnešní doby v IT sektoru je zvyšování poptávky po programátorech. QA (quality assurance) je oddělení, které se ve těchto firmách zaměřuje na testování vyvíjeného produktu. Vzniká nátlak, aby testeři pracovali s jednoduchými nástroji. V důsledku toho se mohou lidé se znalostí programování zaměřit pouze na vývoj software.

U testování API lze velkou část definic a provádění testů zadávat přes uživatelské rozhraní (dále jen UI). To znamená, že tyto testy lze definovat bez potřeby psaní kódu. Tímto způsobem je například možné psát API testy pomocí nástroje Postman (popsaný v kapitole 2.1). Ten umožňuje pomocí UI definovat volání, spouštět je a zobrazovat výsledky. Velká část testerů s tímto programem pracuje a umí ho používat. Toho lze využít a vytvořit nástroj s podobným rozhraním, který ovšem splňuje jiné požadavky na provádění testů, než které nabízí Postman.

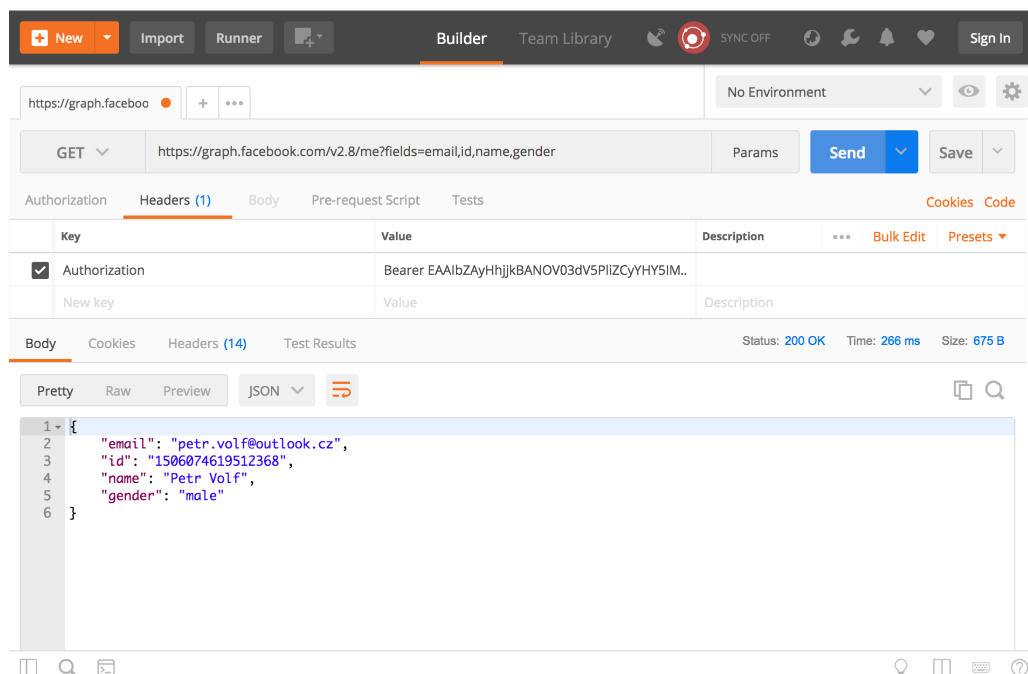
Cílem této bakalářské práce je vytvořit nástroj, pomocí kterého lze vizuálně definovat sady testů API (tzv. testové scénáře), určit očekávané výsledky, naplánovat testy, automaticky je provádět a vytvářet hlášení o průběhu těchto testů. Vše by se mělo snadno a vizuálně konfigurovat, následně i upravovat.

# Kapitola 2

# Stávající řešení

## 2.1 Postman

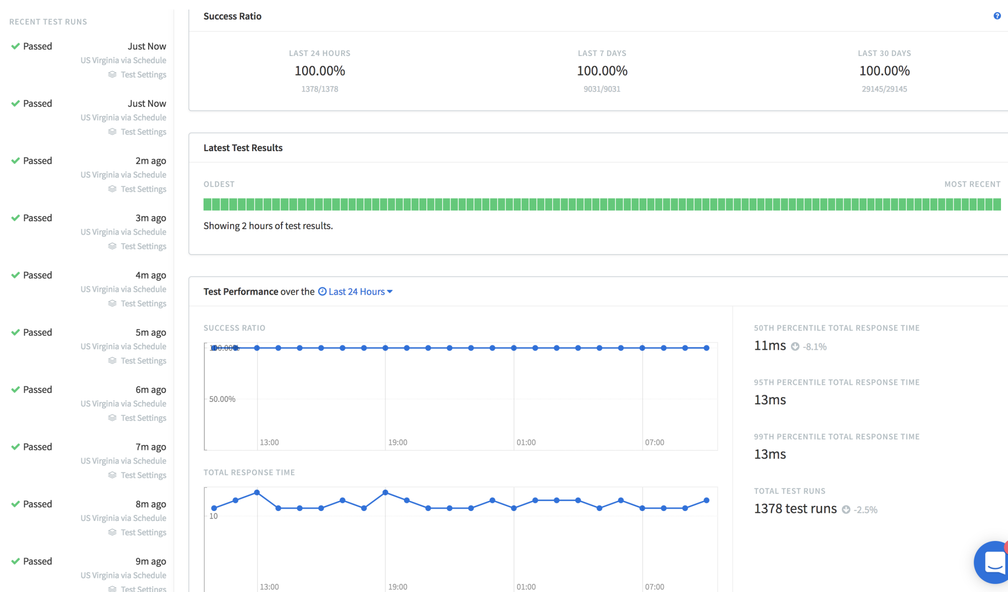
Postman je aplikace, ve které lze definovat a ukládat volání API. Nadefinovat lze například typ HTTP requestu, hlavičky a tělo požadavku. Po odeslání requestu se uživateli zobrazí odpověď serveru – návratový kód, hlavičky odpovědi a její tělo. Postman umožňuje vytvořit kolekce volání, které lze exportovat a například odeslat kolegovi. Lze si také vytvořit proměnné, které lze v definici volání používat a v odpovědi serveru nastavovat. Postman neumožňuje spouštět testy periodicky a tudíž pomocí něho nelze v reálném čase testovat očekávanou funkčnost a dostupnost API.




Obrázek 2.1: Grafická aplikace Postman.

## 2.2 Runscope

Runscope je online nástroj, pomocí kterého lze definovat API testy a jejich očekávaný výsledek. Testy lze již na rozdíl od Postman provádět periodicky a sledovat historii hlášení o jejich provedení. Definice řetězených testů ovšem nelze zadávat vizuálně. Nevýhodou je také vysoká cena (od $948 za rok).




Obrázek 2.2: Webová aplikace Runscope.

## 2.3 Assertible

Assertible je nástroj velmi podobný Runscope. Umožňuje také periodické opakování testů a ukládání jejich výsledků. Narozdíl od Runscope nabízí plán, který je zadarmo. Tento plán ovšem omezuje používání aplikace na pouze dva testovací scénaře.

# Kapitola 3

# Volba technologií

## 3.1 Analýza požadavků

Cílem aplikace je definovat testové scénáře REST API [16], které používá formát JSON. Aplikace musí být schopna nabídnout vizuální vytváření diagramů, které budou vyjadřovat posloupnost provádění testů v daném scénáři. Uživatelská data, definice testů a výsledky provádění musí být reprezentovány ve vhodné podobě (v rámci UI). Aplikace by měla být navržena tak, aby bylo umožněno její škálování. Je tedy nutné, aby aplikace byla schopna reagovat na zvýšení (nebo snížení) počtu požadavků na provádění testových scénářů – například tak, že provádění testů bude distribuováno na více serverech. Bude-li navržena vhodná architektura, bude možné použít stávající řešení na snadno škálovatelných cloudových službách, jako je Heroku nebo Amazon Web Services [5].

Následně je potřeba ukládat výsledky provedených testů do vhodného úložiště, ve kterém půjde rychle vyhledávat v posledních záznamech a agregovat statistiky. Poté půjde vyfiltrovat selhané testy, zobrazit graf dostupnosti a funkčnosti testovaného API.

## 3.2 Single page application

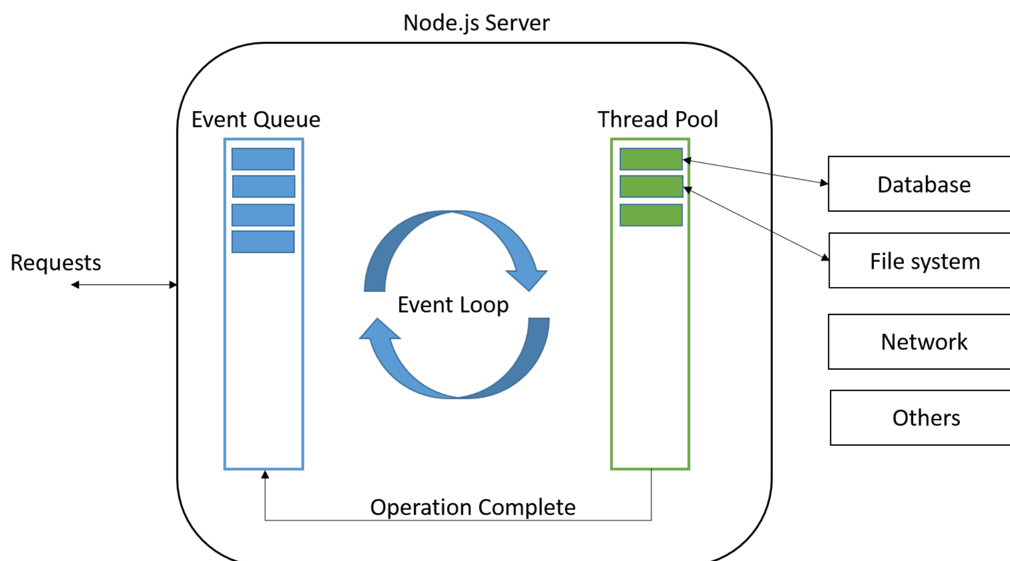
Single page application (SPA) [15], je trendem a obrovským hitem dnešní doby, který upozaďuje dřívější principy server-side-renderingu fungující na principu připravení HTML stránky na straně serveru a její následné vrácení. U SPA si uživatel při prvních několika HTTP požadavcích stáhne JavaScriptovou aplikaci, která se stará o základní věci jako například rendering, routing a stylování. Aplikace komunikuje se serverem pouze pomocí API rozhraní. Protože velká část logiky bude implementována na klientovi (diagramové knihovny, validace formulářů), je zde použití SPA vhodné.

## 3.3 Serverová část

Vzhledem k tomu, že základní modely aplikace jakými jsou Uživatel, Scénář nebo Test budou ve frontendu, backendu, plánovači a workeru stejné, bylo by vhodné tyto komponenty a i části kódu sdílet. Proto místo klasických jazyků pro vývoj serverových aplikací jakými jsou Java, Ruby či PHP jsem si vybral JavaScript (popř. jeho nadstavbu TypeScript, která je popsána v kapitole 3.5). Pro provádění JavaScriptu na serveru se využívá běhové prostředí Node.JS [6], za kterým stojí firmy jako například Microsoft nebo Google. Node.JS je v dnešní době velice populární a díky tomu je kolem něj velmi velká komunita a různé nástroje, které pomáhají při vývoji.

Node.JS je běhové prostředí, které séstává z V8 JavaScript engine od Google (to používá například internetový prohlížeč Chrome), Event loop (která zajišťuje asynchronní operace) a AsyncIO, což je knihovna umožňující souborové a síťové operace.

JavaScript využívá tzv. Event Loop [7]. To znamená, že JavaScriptový kód běží pouze v jednom vlákně. Blokující operace (jako například čtení ze souboru nebo síťové volání) se zaregistrují do fronty zvané ThreadPool – bez čekání na jejich provedení. Tyto operace zpravidla následně provádí nativní moduly. Na ně lze definovat obslužný kód (tzv. callback), který chceme provést po dokončení této operace. Jakmile se blokující operace dokončí (například pokud dostaneme po síťovém volání odpověď ze serveru), položka se přesune do fronty EventQueue včetně definovaného callbacku. Cílem Event Loop je vybírat čekající operace z fronty EventQueue a provádět je. Tento model je velmi výhodný při intenzivním zpracováním I/O požadavků. Node.JS server díky tomu dokáže obsluhovat více požadavků najednou, přestože běží pouze na jednom vlákně. Princip Event Loop lze vidět na obrázku 3.1.




Obrázek 3.1: Grafické znázornění event loop [10].

Stejně jako Java používá pro správu závislostí Maven nebo Gradle, či Ruby používá bundler, tak i Node.JS má vlastní balíčkový systém zvaný npm (*https://www.npmjs.com/*). Díky velké popularitě JavaScriptu lze v npm najít balíček téměr na cokoliv. Npm je v současnosti největší balíčkový systém, který obsahuje přes 350 000 balíčků [11]. Npm (stejně jako bundler či Maven) používá deklarativní model definic balíčku, který se definuje pomocí JSON [14] v souboru `package.json` (pobodné `pom.xml` v Maven). Nainstalování definovaných balíčků se provádí příkazem `npm install`.

## 3.4 Klientská část

Pro klientskou část aplikace jsem se rozhodoval mezi dostupnými frameworky pro SPA aplikace. Mezi nejrozšířenější patří Angular, React a VUE. Z důvodu větší flexibility jsem si zvolil React. Je to menší knihovna starající se pouze o jednu věc – zobrazování. Narozdíl od Angular.js si můžeme sestavit používané nástroje sami a díky komunitním knihovnám jako je například Redux, Mobx nebo React Router nahradí React vestavěné vlastnosti frameworku AngularJS a navíc poskytne vývojáři větší volnost při výběru a použití těchto nástrojů.

### 3.4.1 React

React je opensource projekt [12] od společnosti Facebook. React využívá deklarativní zápis a rozděluje části aplikace do komponent. Komponenta

je nezávislá a znovupoužitelná část kódu, která zpravidla definuje určitou část uživatelského rozhraní (*https://reactjs.org/docs/react-component.html*). Klasickým příkladem může být tlačítko, položka v menu či menu samotné. Tyto komponenty následně skládáme podle potřeby dohromady a při změnách stavu (vysvětleno dále) jsou automaticky přerenderovány[1]. React upřednostňuje model kompozice před dědičností.

### Renderování komponent

Renderování komponent řeší modul knihovny React s názvem *React DOM*. Každá React komponenta je neměnná. Po změně stavu se vytvoří nová React komponenta, kterou *React DOM* porovná s komponentou aktuální a do DOM[2] vykreslí pouze změny, které nastaly.

### JSX

Výsledkem vyrenderování React komponent je HTML stránka. JSX je rozšíření syntaxe jazyka JavaScript, který ulehčuje a zpřehledňuje zápis těchto komponent. Tato syntaxe se poté pomocí nástoje babel zkompiluje do syntaxe JavaScriptu. Příklad zápisu sytaxe JSX a jeho následnou zkompilovanou verzi lze vidět v kódech 3.1 a 3.2.

Kód 3.1: Definice React elementu zápisem syntaxe JSX.

```
1 const reactElement = (
2   <p>
3     Hello world
4   </p>
5 );
```

Kód 3.2: Definice React elementu zápisem syntaxe JavaScriptu.

```
1 const element = React.createElement(
2   'p',
3   {},
4   'Hello world'
5 );
```

[1] renderování = vykreslování

[2] DOM = Document Object Model

### State a props

Každá komponenta může pracovat s dvěmi zdroji dat – state (stav) a props. State je stav komponenty, který si řídí komponenta sama. V tomto stavu může být například informace o aktivním tlačítku v menu, nebo stav vyplněného formuláře. Props jsou data komponenty, které ji přiřadila komponenta nadřazená. Props uvnitř komponenty nelze modifikovat. Je běžným zvykem, že stav určité části aplikace řídí pouze jedna nadřazená komponenta, která tuto informaci předává svým pod-komponentám jako props. Existuje speciální prop s názvem `children`, který obsahuje pole komponent, které jsme aktuální komponentě přiřadili. Použití `children` lze vidět v kódu 3.3.

Kód 3.3: Použití prop `children`.

```
const ListContainer = () => (
  <ItemList>
    <div>Item 1</div>
    <div>Item 2</div>
  </ItemList>
)

const ItemList = ({children}) => (
  <div>
    children.map(item => (
      {item}
    ))
  </div>
)
```

### High order component

HOC - high order component, je speciální typ komponenty, který obaluje a předává pod-komponentám určité vlastnosti a data. V celé aplikaci se maximálně snažím o to, aby základní prvky nebyly závislé na okolí a potřebné metody a data jim byly předávány. Často se pro tyto komponenty využívá název Container, který sám o sobě nic nerenderuje pouze připravuje prostředí a načítá data pro jiné komponenty.

### Server-side a client-side rendering

V React lze používat jak server-side, tak client-side rendering. Server-side rendering s každým požadavkem na server odešle klientovy vygenerovanou stránku – zpravidla tedy pouze to, co klient bude v danou chvíli potřebovat.

Výhodou je, že klient nemusí při prvním požadavku čekat, než si stáhne celou aplikaci. Další výhodou je konzistentní výkon a dobré SEO[3] bez větších nutných optimalizací [1]. Nevýhodou je větší zátěž serveru – při každém požadavku se musí odeslat velké množství dat.

Client-side rendering s prvním požadavkem na server stáhne celou aplikační logiku. Výhodou je měnší zátěž serveru a méně přenesených dat při používání aplikace. Server odesílá klientova pouze data o které klient žádá – neřeší už tedy samotné vykreslování aplikace.

## 3.5 TypeScript

TypeScript je preprocessor jazyka JavaScript, který ho rozšiřuje o statické typování. TypeScript je transpilován[4] do JavaScript normy ES5 (který podporuje většina prohlížečů) nebo ES6 kódu (který využívá Node.JS). Díky tomu ho lze využít pro jakékoliv JavaScript aplikace (včetně frameworku Node.JS a React). Díky statickému typování velmi zvyšuje přehlednost a přenositelnost kódu. V JavaScriptu je běžnou chybou, že se vývojář pokouší číst proměnnou z objektu, která není nastavena nebo neexistuje. TypeScript umožňuje tyto objekty přes rozhraní definovat, ověřit pak případné překlepy a umožňuje i lepší našeptávání v rámci IDE. Ukázku definice a použití rozhraní lze vidět na v kódu 3.4.

Velké množství externích JavaScript balíčků obsahuje mimo JavaScript kódu také definice rozhraní pro TypeScript, se kterými balíček pracuje. Díky tomu lze využít přednosti TypeScriptu i při práci s cizím kódem.

Kód 3.4: Definice TypeScript rozhraní a jeho použití.

```
1  interface Scenario {
2      id: string;
3      name: string;
4      lastRun: Date;
5      period: number;
6  }
7
8  function updateScenario(scenario: Scenario): Scenario {
9    scenario.lastRun = new Date();
10   return scenario;
11 }
```

[3]SEO = Search Engine Optimalization

[4]transpilace = překlad zdrojového kódu z jednoho jazyka do jazyka druhého

## 3.6 Databázové systémy

V aplikaci se bude pracovat s různorodými daty, na které jsou kladeny různé požadavky. Část dat bude možné dekomponovat do relační databáze, ale například data z výsledků testů nejsou pro relační databázi vhodné. Odpovědi serverového API nemají jednotnou strukturu a výsledků bude velké množství, ze kterých nás budou většinou zajímat pouze nedávné záznamy. Z toho důvodu jsem se rozhodl použít 3 databázové systémy – PostgreSQL - rdbms[5], Redis - key value storage a Elasticsearch - dokumentová databáze.

### 3.6.1 PostgreSQL

PostgreSQL využiji pro relační data. To jsou například informace o uživatelích a nadefinovaných testových scénářích.

### 3.6.2 Redis

Databáze Redis je díky ukládání dat v paměti velice rychlá. Není to pouze inmemory key-value databáze. Oproti rdbms nevyžaduje konzistenci a přesnou strukturu dat. Podporuje i další množství jiných funkcí, jakými jsou například fronty a atomické operace nad nimi – jako blokující pop. Do fronty v Redisu bude plánovač periodicky ukládat čísla scénářů pro provedení.

### 3.6.3 Elasticsearch

Elasticsearch použiji jako úložiště reportů provedených testů. Elasticsearch je NoSQL[6] úložiště, které je vhodné pro ukládání velkých množství dat. Výsledky celých scénářů budou obsahovat různorodá data, pro které je potřeba nalézt vhodnější úložiště než jakým je relační databáze. Pokud bychom chtěli přirovnat Elasticsearch k relační databázi, tak index je databáze a objekt je řádka. Oproti relační databázi se zde však můžeme setkat i se šablonami, které definují jakou ukládané objekty budou mít strukturu. Založení indexu je v Elasticsearch automatická událost, která se děje při vložení prvního objektu. Proto v kombinaci se šablonami je možné pro každý měsíc výsledků založit nový index. Dotazy v Elasticsearch jsou koncipovány tak, že se můžeme dotazovat pomocí wildcard (*) symbolů. Například případě existence indexů `logs-2018-01` a `logs-2018-02` lze provést dotaz `GET /search/logs-*`.

[5]rdbms = Relational database management system
[6]NoSQL = Not only SQL

### Kibana

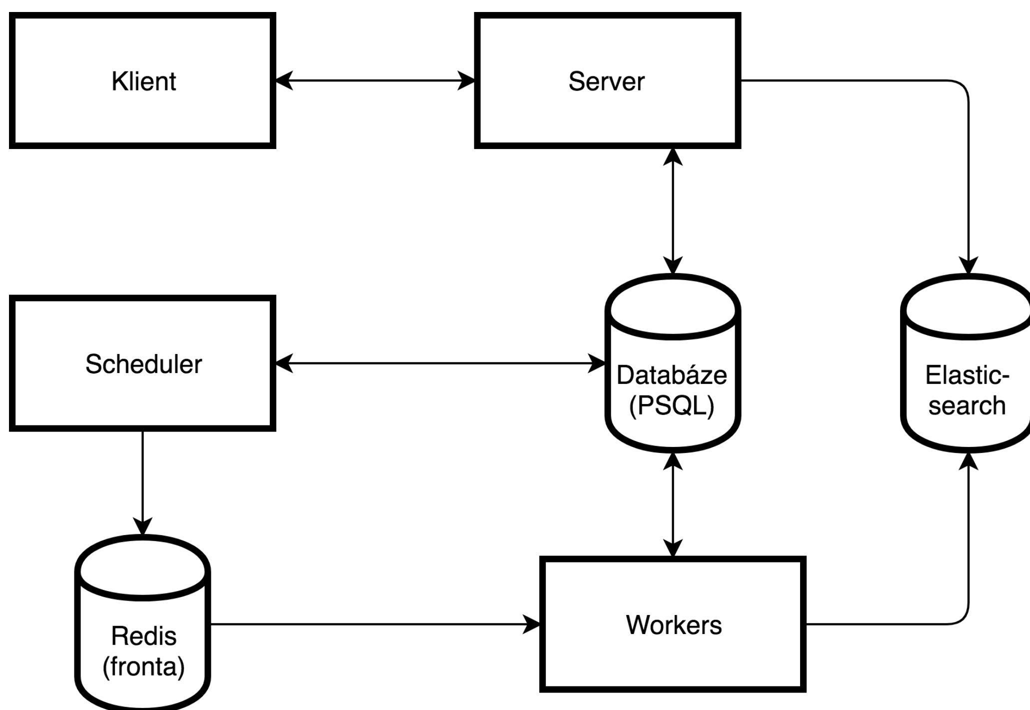
Kolem Elasticsearch existuje řada nástrojů – kromě Logstash je asi nejznámější Kibana. Kibana [8] je vizualizační nástoj pro Elasticsearch. Tento nástroj je možné využít pro prohlížení dat, vizualizaci reportů a případně i detekci anomálií.

# Kapitola 4

# Architektura

## 4.1 Popis architektury

Vykonávání testů bude probíhat opakovaně dle zadané periody. Zpracování těchto požadavků funguje na bázi producent-konzument. Tzv. Scheduler (producent) zjišťuje, které testovací sady čekají na provedení. Tyto testovací sady poté vloží do fronty v databázi Redis a z této fronty poté konzumenti (tzv. workeři) úkoly vybírají a provádějí. Po dokončení testů výsledky worker uloží do Elasticsearch.

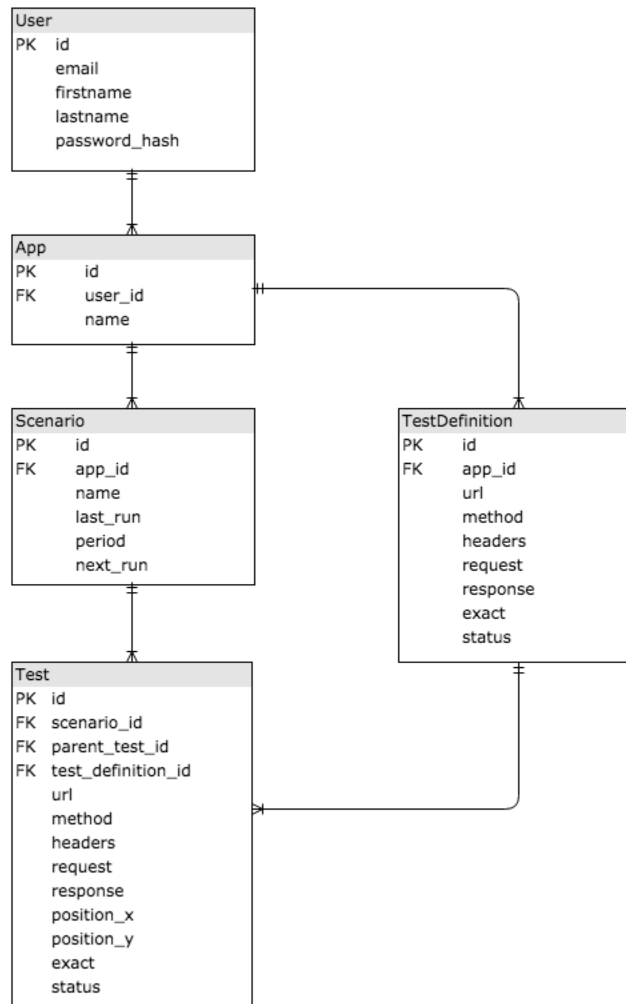



Obrázek 4.1: Architektura aplikace.

## 4.2 Databáze

Databáze obsahuje 5 tabulek, které reprezentují uživatelská data, definici testů a definici scénářů. Protože samotná definice dotazu a i odpovědi je

v JSON formátu, používám datový typ JSON, který případně otevírá možnosti vyhledávání v definicích či zanořených strukturách. Pokud bychom rozšiřovali podporu pro formát XML nebo SOAP, museli bychom datový typ migrací změnit na text. Definice migrací je popsána v kapitole 5.2.2.




Obrázek 4.2: Struktura databáze.

### 4.2.1 Tabulka User

Tabulka `User` uchovává všechny informace o zaregistrovaném uživateli. Sloupec s unikátním klíčem `email` vyjadřuje přihlašovací identifikátor uživatele. Sloupec `password_hash` uchovává heslo uživatele zašifrované hashovací funkcí Bcrypt [17].

### 4.2.2 Tabulka App

Tabulka app reprezentuje testovanou aplikaci. Jedna aplikace může obsahovat více testovacích sad. Tyto sady definuje tabulka `Scenario`.

### 4.2.3 Tabulka Scenario

Tabulka Scenario obsahuje informace o testovacím scénáři (sadě testů), který reprezentuje testovanou aplikaci. Sloupec `last_run` vyjadřuje poslední čas spuštění sady testů. Sloupec `period` označuje čas v sekundách, po kterém se má volání testu periodicky opakovat. Sloupec `next_run` reprezentuje další čas spuštění testu. Pomocí `app_id` je vazba na aplikaci, ke které patří.

### 4.2.4 Tabulka Test

Tabulka `Test` obsahuje definici jednoho testu z dané sady testů, které reprezentuje tabuka `App`. Tato tabulka obsahuje všechny potřebné informace k provedení jednoho testu, včetně jeho validačních pravidel. Sloupec `method` rozděluje typ požadavku na typ POST, GET, PUT a DELETE. Sloupec `headers` obsahuje hlavičky požadavku, `request` reprezentuje tělo požadavku. Struktura těla požadavku je specifikována v kapitole 4.3.1. Sloupec `response` vyjadřuje validační pravidla odpovědi. Definice těchto pravidel je popsána v kapitole 4.3.2. Sloupec `exact` definuje, zda se v odpovědi mohou nacházet klíče objektů, které nejsou v pravidlech definovány. Položka `status` udává, jaký návratový HTTP kód odpovědi očekáváme. Soupce `position_x` a `position_y` reprezentují souřadnice umístění testů v diagramu scénáře. Sloupec `parent_test_id` vyjadřuje předchozí test v sadě. Pokud je hodnota tohoto sloupce `NULL`, znamená to, že test je první v pořadí.

### 4.2.5 Tabulka TestDefinition

Tato tabulka je velmi podobná tabulce `Test`. Obsahuje definici testů napříč aplikací. Uživatel tedy má nadefinované společné testy pro každou aplikaci, které následovně může v jednotlivých testových scénářích používat. Toto řešení je vhodné z důvodu společných vlastností aplikace, jako je například autentifikace uživatele.

## 4.3 Definice testů

Při definování testů se určují následující parametry:

- Název testu
- Cílová URL
- Metoda HTTP požadavku
- HTTP hlavičky
- Struktura volání
- Struktura pravidel odpovědi

Struktury volání a pravidel odpovědi mají speciální zápis, který je popsán v této kapitole.

### 4.3.1 Struktura volání

V těle zprávy můze být využita proměnná, kterou jsme uložili z odpovědi v některých předchozích testech. V případě použití této proměnné se v hodnotě zprávy definuje následující sekvence: `fill:${<variable_name>}`. Proměnnou lze použít jak v těle zprávy, tak i hlavičce nebo URL. Pokud tedy například chceme přistopit na URL s číslem aplikace v proměnné `appId`, výsledek definice URL může vypadat následovně:
`http://www.veripi.com/apps/fill:${appId}/`. Definice volání může vypadat následovně:

```
1 {
2   "city": "fill:${city_name}",
3   "day": "tomorrow"
4 }
```

### 4.3.2 Struktura pravidel odpovědi

Předpokládejme, že server na požadavek o předpověď počasí vrátí následující odpověď:

```
1  {
2      "data": [
3        {
4            "city": "Plzen",
5            "forecast": "23,5"
6        },
7         {
8            "city": "Praha",
9            "forecast": "27"
10       },
11     ],
12     "day": "tommorow"
13 }
```

Pro tento požadavek můžeme nadefinovat následující pravidla:

```
1  {
2      "type": "object",
3      "item": {
4          "data": {
5              "type": "array",
6              "item": {
7                  "type": "object",
8                  "item": {
9                      "city": {
10                         "type": "string"
11                     },
12                     "forecast": {
13                         "type": "string"
14                     }
15                 }
16             }
17         },
18         "day": {
19           "type": "string",
20           "saveto": "requestedDay"
21         }
22     }
23 }
```

Klíč `type` vyjadřuje, typ validované odpovědi. Může nabývat několika hodnot (dle všech podporovaných typů zápisu JSON):

- `string`, `number`, `boolean`, `null` – Primitivní datové typy, pravidlo validuje přítomnost daného typu.

- `object` – Očekáváme objekt. Na stejné úrovní jako klíč `type` se musí nacházet klíč `item`, který definuje strukturu objektu. Hodnoty klíčů objektu `item` vyjadřují typ klíče v odpovědi.

- `array` – Stejně jako u `object` očekáváme klíč `item`, jehož hodnota definuje typ položek v poli.

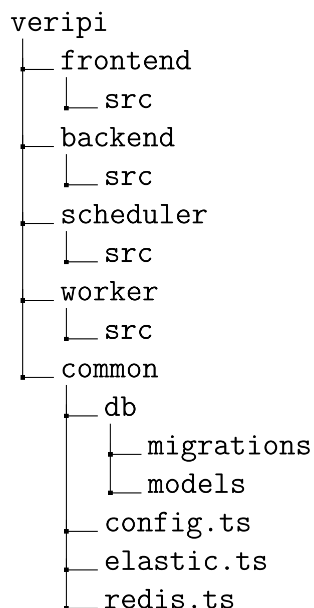
Na úrovni klíče `type` lze použít klíč `saveto`. Typ této položky musí být jakýkoliv primitivní typ. Účelem tohoto klíče je uložit výsledek volání do jakési proměnné, které se použije v dalších voláních. Příkladem může být například autentizační token uživatele, který získáme po přihlášení a budeme ho chtít používat v následujících voláních. Hodnota klíče `saveto` je string, pomocí kterého později k této proměnné přistoupíme.

# Kapitola 5

# Implementace

## 5.1 Struktura projektu

Základní strukturu projektu lze vidět na obrázku 5.1.




Obrázek 5.1: Základní struktura projektu.

## 5.2 Společný modul

Společný modul obsahuje obecné definice, které využívají ostatní moduly aplikace. To je například konfigurace projektu a databázových systémů.

### 5.2.1 Konfigurace

Konfigurace projektu je realizována prostým JSON souborem, který se nachází v souboru `config/config.json`. V tomto souboru lze definovat různá nastavení. Pro každé prostředí lze definovat například port Node.JS serveru,

konfigurace připojení do databází Redis a Elasticsearch, secret token pro knihovnu JWT[1], která spravuje autentizační tokeny pro autentifikaci uživatelů, nebo konfiguraci Raven (pro logování chyb v Sentry, které je popsané v kapitole 5.7). Konfigurace databáze PostgreSQL je definována ve vlastním JSON souboru, který musí být oddělený od zbytku konfigurace (protože Sequelize načítá konfiguraci sám při provádění migrací). Tyto konfigurace lze vidět v kódech 5.1 a 5.2.

Kód 5.1: Ukázka souboru config.json ve kterém je konfigurace prostředí development.

```
{
    "development" : {
      "port" : 3000,
      "jwtSecret": "tajnyjwtsecret",
      "raven": "https://b05d3a312d1212gc7b639a1a176@sentry.io
         /11443265",
      "tokenExpiration": "7d",
      "redis": {
        "engine": "redis",
        "ttl": 3600,
        "port": 6379,
        "host": "localhost",
        "queue": "tests"
      },
      "elastic": {
        "host": "localhost",
        "port": "9200",
        "index": "reports"
      }
    }
}
```

[1]JWT = Json Web Token (*https://jwt.io/*)

Kód 5.2: Ukázka konfigurace databáze PostgreSQL pro prostředí development.

```
1 {
2   "development": {
3     "username": "veripi",
4     "password": "veripi",
5     "database": "veripi_test",
6     "host": "127.0.0.1",
7     "dialect": "postgres"
8   }
9 }
```

### 5.2.2 Definice relační databáze

#### Definice modelů

Tato část obsahuje definici modelů knihovny *Sequelize*. Každý model představuje mapování jedné databázové tabulky do jedné JavaScript třídy. Tato třída dědí od Sequelize třídy `Model`, která nabízí abstrakci od databázového systému. Sequelize nepodporuje pouze PostgreSQL a dokáže komunikovat i s jinými databázemi. Kdyby přestal být PostgreSQL dostačující, lze bez větších potíží přejít na jiný systém.

Pro definici Sequelize modelů jsem si vybral knihovnu *sequelize-typescript*. Ta přidává typové definice, anotace a podporu TypeScriptu, kterou originální knihovna nemá. Tato knihovna také nabízí možnost definice Sequelize modelů pomocí TypeScript tříd a dekorátorů, které zatím JavaScript engine Node.JS nenabízí. Dekorátory jsou aktuálně v TypeScriptu experimentální vlastnost, kterou je potřeba v konfiguraci TypeScriptu explicitně nastavit [2].

Kód 5.3: Ukázka definice modelu pomocí knihovny sequelize-typescript.

```
@Table({
  tableName: 'apps',
  timestamps: true,
  underscored: true
})
export default class App extends Model<App> {

  @Column({    // definice sloupce
      type: DataType.STRING,
      allowNull: false
  })
  name: string;

  @ForeignKey(() => User)     // definice ciziho klice
  @Column({ field: 'user_id' })
  userId: number;

  @BelongsTo(() => User)     // definice relace 1:N
  user: User;
}
```

### Migrace

Další součástí společného modulu jsou definice migrací databáze. Knihovna Sequelize podporuje definici migračních záznamů a obsahuje nástroj pro správu a spouštění těchto migrací. Migrace se nedefinují jazykem SQL, ale používáním funkcí definovanými knihovnou Sequelize, které jsou nezávislé na databázovém systému (stejně jako ostatní funkce Sequelize). V definici migrace je nutno uvést i postup pro vrácení (*revertování*) změny. Příklad migračního souboru lze vidět v kódu 5.4. Klíč `up` v tomto kódu definuje změnu sturktury databáze při provádění migrace, klič `down` reprezentuje vrácení této změny. Sequelize se stará o to, aby spuštění každého migračního záznamu proběhlo pouze jednou. V migracích lze také naplňovat nebo mazat data v tabulkách, případně i přímo na disku. V takovém případě nelze pokaždé přesně definovat postup vrácení změny, protože data již mohou být ztracena.

Kód 5.4: Definice migrace pomocí knihovny Sequelize.

```
1  module.exports = {
2    up: (queryInterface, Sequelize) => {
3      return queryInterface.addColumn('scenario', 'next_run', {
4          type: Sequelize.DATE,
5              allowNull: true,
6              defaultValue: null
7      })
8    },
9    down: (queryInterface, Sequelize) => {
10     return queryInterface.removeColumn('scenario', 'next_run'
         )
11   }
12 };
```

## 5.3 Klientská část

V rámci implementace klientské části jsem využil několik nástrojů, které mi usnadnily nebo zpřehlednily vývoj.

### 5.3.1 Redux

Redux je knihovna pro globální state management [13], který využívám v rámci celé frontendové části aplikace. Základní koncept této knihovny je takový, že veškerý stav aplikace je uložen v jediném stromu JavaScript objektů, který uchovává takzvaný Redux store. Redux není vázaný pouze k frameworku React a lze ho využít v jakékoliv JavaScript aplikaci. Přesná struktura stavu aplikace musí být předem známá (vysvětleno dále). Stav nelze přímo měnit, lze z něho přímo pouze číst. Všechny změny stavu lze provádět pouze předdefinovanými objekty zvanými actions.

#### Actions

Redux action je objekt, který definuje změnu globálního stavu aplikace. Tento objekt definuje pouze typ akce a data, která se zpravidla uloží do Redux store. Akce nedefinují samotný proces transformace stavu. Tuto transformaci provádí takzvané reducers funkce. Funkce, která vytváří akci se nazývá *action creator*.

Kód 5.5: Redux action creator.

```
1 const showUiLoading = (message: String) => ({
2   type: 'SHOW_UI_LOADING',
3   message
4 });
```

Typ akce definuje, jakým způsobem se bude transformovat globální stav. Vhodnější, než řetězcový zápis je definice výčtového typu v zápisu TypeScript, který definuje možné typy akcí. Poté lze využít možností IDE a v projektu se snáze orientovat (včetně našeptávání akcí).

Kód 5.6: Redux action creator s využitím TypeScript enum.

```
1 enum TypeKeys {
2   SHOW_UI_LOADING = 'SHOW_UI_LOADING',
3   HIDE_UI_LOADING = 'HIDE_UI_LOADING'
4 }
5
6 const showUiLoading = (message: String) => ({
7   type: TypeKeys.SHOW_UI_LOADING,
8   message
9 });
```

Odesláním akce se rozumí použití funkce `dispatch()` nad Redux store, která je součástí knihovny redux. Argumentem této funkce je akce, kterou chceme provést.

Kód 5.7: Odeslání Redux akce.

```
1 const loadingAction = showUiLoading('Creating scenario');
2 dispatch(loadingAction);
```

Odesílání Redux akcí je synchronní událost bez side effects[2]. Pro použití akcí se side effects je použita knihovna *Redux thunk* (popsána dále).

### Reducer

Změnu stavu provádí funkce zvaná reducer. Tato funkce je analogií k metodě `reduce` dostupná v JavaScript třídě `Array`. Funkce obsahuje 2 argumenty –

[2]side effect = jakákoliv interakce mimo rámec aktuální funkce (například síťové volání)

akumulátor a odeslanou akci. Akumulátor představuje předchozí stav aplikace. Úkol této funkce je vrátit nový transformovaný stav na základě typu odeslané akce. Tato funkce nesmí modifikovat předchozí stav aplikace, jeho změna by způsobila nefunkčnost několika vlastností této knihovny, jako například *time travelling*, který lze využít při debugování aplikace [9]. Reducer funkci lze vidět v kódu 5.8. Reducer `uiReducer` spravuje ukládání stavu o zobrazení indikátoru načítání (a zprávy, kterou indikátor obsahuje).

Kód 5.8: Reducer funkce.

```
const initialState: UiData = {
  loading: false,
  loadingMessage: ''
};

const uiReducer = (state: UiData = initialState, action:
    ActionTypes) => {
  switch (action.type) {
    case TypeKeys.HIDE_UI_LOADING:
      return {
        ...state,
        loading: false
      };
    case TypeKeys.SHOW_UI_LOADING:
      return {
        ...state,
        loading: true,
        loadingMessage: action.message
      };
    default:
      return state;
  }
};
```

V případě neexistence předchozího globálního stavu je nutno stav inicializovat. V případě, že v aktuální reducer funkci není definovaná akce, která odpovídá odeslané akci, je nutno vrátit předchozí stav. Pro zvýšení přehlednosti můžeme reducer funkce zanořovat. Poté nám vznikne více reducer funkcí, kde jedna z nich bude funkce vstupní, která bude postupně volat funkce vnořené.

Kód 5.9: Použití vnořených reducer funkcí.

```
1 const rootReducer = (state: RootData = {}, action:
    ActionTypes) => {
2     return {
3       ui: uiReducer(state.ui, action),
4       userData: userReducer(state.user, action)
5     }
6   }
7 };
```

### Redux thunk

Redux thunk je middleware[3] knihovny Redux, který umožnuje psát funkce actions creators, které vrací funkci místo objektu. Do této funkce poté vloží jako argument funkci `dispatch()`, kterou lze použít pro odeslání nové akce. Redux thunk pouze pozastaví vykonávání `dispatch()` funkce, pokud je akce asynchronní. Middleware umožní její provedení až v případě, kdy se odesílá akce bez side efektů. Výhodou tohoto přístupu je možnost odesílat asynchronní akce, které uvnitř mohou odesílat další akce po nějaké události (například po získání dat ze serveru).

Kód 5.10: Příklad použití Redux thunk.

```
1  loginUser = (email, password) => {
2      return async (dispatch) => {
3          // odeslani akce o zacatku nacitani
4          dispatch(showUiLoading('Logging in'));
5
6          // blokujici serverove volani
7          const response = await ApiClient.request('post', '
               users/login', {
8              email,
9              password
10         });
11
12         // odeslani akce v zavislosti na vysledku serveroveho
               volani
13         if (response.error) {
14             dispatch(loginUserFail(response.message));
15         } else {
16             dispatch(loginUserSuccess(response.token,
                   response.id));
17         }
```

[3]Redux middleware = obalení funkcinality metody `dispatch`. Middlewares lze řetězit.

```
18
19          // skryti nacitaciho indikatoru
20          dispatch(hideUiLoading());
21      };
22  };
```

### Použití s frameworkem React

Redux nabízí nástroj pro snazší integraci do frameworku React s názvem *React-Redux*. Tento nástroj není součástí npm balíčku Redux a je ho potřeba přidat dodatečně. *React-Redux* nabízí pomocnou funkci `connect()`, která vrací high-order komponentu. Funkce `connect()` obsahuje 3 argumenty – `mapStateToProps`, `mapDispatchToProps` a `mergeProps`.

- `mapStateToProps` – Definuje transformaci globálního stavu aplikace na props komponenty.
- `mapDispatchToProps` – Definuje transformaci funkce `dispatch()` na props komponenty.
- `mergeProps` – Definuje transformaci props rodiče, výstupu `mapStateToProps` a `mapDispatchToProps` na props výsledné komponenty. V případě neuvedení tohoto argumentu se všechny složky sloučí do jednoho objektu.

Příklad použití této funkce je v kódu 5.11. Komponenta `Signup` v tomto případě dostane navíc props `user` a `signupUser`. Tato prezentační komponenta není nijak propojená s knihovnou *Redux* a v případě nutnosti použití jiného zdroje dat není nutné tuto komponentu jakkoliv modifikovat.

Kód 5.11: Příklad použití funkce connect knihovny React Redux.

```
1  import Signup from './signup';
2
3  function mapStateToProps(state: RootState) {
4    return {
5      user: state.user
6    };
7  }
8
9  function mapDispatchToProps(dispatch) {
10   return {
11     signupUser: (email, password) => dispatch(signupUser(
         email, password))
12   };
13 }
14
15 export default connect(mapStateToProps, mapDispatchToProps)(
     Signup);
```

### 5.3.2 Diagram scénáře

Interaktivní diagram pro definici scénáře jsem implementoval pomocí knihovny *Storm React Diagrams* (*https://www.npmjs.com/package/storm-react-diagrams*).

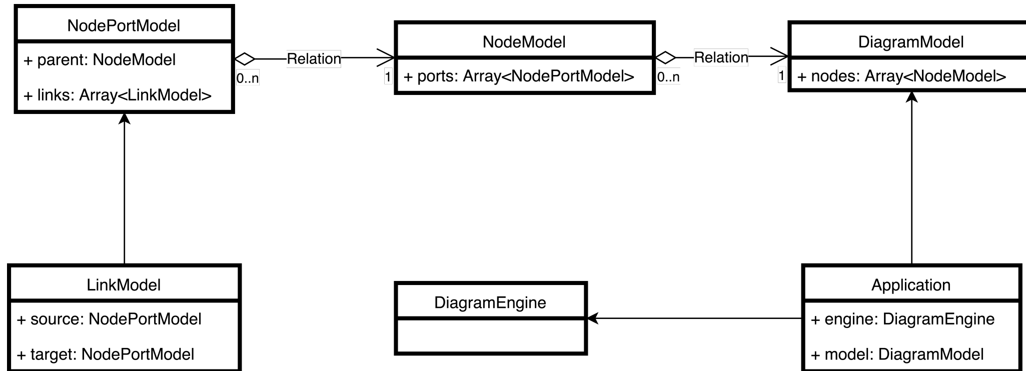
#### Komponenty diagramu

Diagram je rozdělen na několik modelů – `DiagramModel`, `NodeModel`, `NodePortModel` a `LinkModel`. Všechny tyto části jsou rozšířitelné a ve výsledném diagramu jsem některé z těchto částí musel modifikovat.

- `DiagramModel` – Uchovává všechny uzly i spojení diagramu a definuje aktuální stav diagramu.
- `NodeModel` – Definuje uzel diagramu. Uchovává informaci o portech uzlu, ve kterých lze realizovat spojení mezi uzly. V mé aplikaci navíc obsahuje data o testu, který uzel reprezentuje.
- `NodePortModel` – Definuje jeden port možného spojení mezi uzly.
- `LinkModel` – Definuje zdrojový a cílový port jednoho spojení mezi uzly.

Vykreslení jednotlivého uzlu diagramu řeší React komponenta `NodeWidget`, která jako props přijíma `NodeModel` daného diagramu (je to tedy klasická prezentační komponenta).

Zobrazování celého diagramu má na starost `DiagramEngine`, který je také součástí knihovny. Spojením všech těchto definovaných modulů aplikace vzniká funkční React diagram komponenta.




Obrázek 5.2: Struktura komponent diagramu.

### Diagram tray

Přidávání nového uzlu do diagramu probíhá vyběrem testu z bočního panelu. Tento panel obsahuje data z databázové tabulky `TestDefinition`. Cílem je usnadnit skládání jednotlivých testů v scénáři. Testy se zde před přidáním do diagramu předdefinují. V tomto panelu lze vidět výběr všech možných testů, které lze do aplikace přidat. Přidání testu do panelu probíhá přes modální okno formuláře. Testy v panelu jsou stejné pro všechny scénáře ve zvolené aplikaci. Při editaci testu v panelu lze zvolit, jestli chceme provedené změny přepsat i do již existujících testů v diagramu.

Tento panel reprezentuje v aplikaci komponenta `DiagramTray`. Předávání informace o testu diagramu je implementováno pomocí událostí `onDragStart` a `onDrop`. Ty lze vidět na ukázce v kódech číslo 5.12 a 5.13.

Kód 5.12: Příklad události onDragStart.

```
<div
    draggable={true}
    onDragStart={(event) => {
        /* Ulozeni informace o testu do udalosti tahnuti */
        event.dataTransfer.setData("node-data", JSON.
            stringify({...testItem}));
    }}
>
    <div>
        {testItem.name}
    </div>
</div>
```

Kód 5.13: Příklad události onDrop.

```
<div
    onDrop={
        (e) => {
            /* Ziskani dat z pri pretazeni testu do diagramu
                */
            const data = event.dataTransfer.getData("node-
                data");
            const payload = JSON.parse(data);
            this.addNode(payload)
        }
    }
    onDragOver={(event) => { event.preventDefault(); }}
>
    <DiagramWidget />
</div>
```

### 5.3.3 Routování

Routování jsem v aplikaci řešil pomocí knihovny *React Router*. Tato knihovna využívá deklarativní zápis cest, které jsou snadno čitelné a udržitelné. Ukázka definice cest je v kódu 5.14. Komponenta `Switch` zajistí, že se použije pouze první definice cesty, ke které najde shodu.

Kód 5.14: Ukázka definice rout knihovny React Router.

```
1  <Switch>
2        <Route path="/" exact={true} component={Home} />
3        <Route path="/login" exact={true} component={Login} />
4        <Route path="/signup" exact={true} component={Signup}
            />
5        <Route component={EnsureLoggedInContainer} />
6  </Switch>
```

Každá cesta se definuje použitím komponenty `Route` z této knihovny. Prop `path` specifikuje cestu, prop `component` definuje komponentu, která se má pro tuto cestu vykreslit.

### Routy pouze pro přihlášené uživatele

Routy, které náleží pouze přihlášenému uživateli spravuje komponenta `EnsureLoggedInContainer`. Tato HOC[4] komponenta zjistí, zda je aktuálně přihlášen nějaký uživatel a pokud ano, vloží další definice cest. Tento proces lze vidět v kódech 5.15 a 5.16.

Kód 5.15: Metoda render komponenty EnsureLoggedInContainer.

```
1  render() {
2      if (this.props.user) {
3          return (
4          <div>
5              <Route path="/app" component={Dashboard} />
6          </div>
7          );
8      }
9
10     return null;
11 }
```

[4]HOC = High Order Component

Kód 5.16: Definice rout přihlášeného uživatele komponenty Dashboard.

```
1 <Switch>
2       <Route component={Main} exact={true} path="/app" />
3       <Route component={Settings} path="/app/settings" />
4       <Route component={List} path="/app/list" />
5       <Route component={AppDetail} path="/app/detail/:id"
           exact={true} />
6       <Route component={Scenario} path="/app/detail/:id/
           scenario/:scenarioId" />
7 </Switch>
```

### Integrace s knihovnou Redux

*React Router* lze zintegrovat s knihovnou Redux pomocí npm balíčku *React Router Redux*. Díky tomuto balíčku lze pomocí Redux `dispatch()` funkce provádět přesměrování uživatele. Tato knihovna podporuje *time travelling* Reduxu. To znamená, že při použití *time travellingu* React Router aktualizuje svůj stav a vyrenderuje požadovanou komponentu dle aktuální cesty. Použití této knihovny je uvedeno v kódu 5.17.

Kód 5.17: Ukázka funkce action creator která po přihlášení přesměruje uživatele na výchozí stránku.

```
1  const loginUser = (email, password) => {
2      return async (dispatch) => {
3          const response = await ApiClient.request('post', '
              users/login', {
4              email,
5              password
6          });
7          if (!response.error) {
8            dispatch(push('/app'));
9          }
10     };
11 };
```

## 5.3.4 TSLint

TSLint je statický analyzátor kódu, který jsem používal při vývoji. TSLint je modifinovaná verze JavaScriptového nástroje ESLint pro potřeby syntaxe TypeScriptu. Jeho analogií je nástroj PMD, který využívá jazyk Java.

TSLint určuje striktní pravidla psaní kódu, která mají být dodržování v průběhu vývoje. Velmi užitečný je při týmové práci, kdy od všech programátorů TSLint vyžaduje identický styl zápisu kódu. Výsledkem toho je větší přehlednost a udržitelnost.

TSLint lze nakonfigurovat v konfiguračním souboru `tslint.json` v kořenovém adresáři projektu. Zde je možnost nadefinovat různá pravidla, které TSLint nabízí. Pro použití TSLintu lze použít plugin v IDE, nebo ho lze spustit v terminálu příkazem `./node_modules/.bin/tslint -c tslint.json 'frontend/**/*.ts'` (Pro všechny soubory s příponout `.ts` ve složce `frontend` v aktuálním projektu).

## 5.4 Serverová část

Přístupový bod aplikace je soubor `backend/src/server.js`, který inicializuje framework ExpressJS.

### 5.4.1 ExpressJS

Tato knihovna rozšiřuje funkcionalitu Node.JS modulu `http`.

#### Routing

Router frameworku ExpressJS definuje přístupové cesty rozdělené dle metody HTTP požadavku. Pro každou tuto cestu je možné definovat funkci, která požadavek obslouží a vrátí odpověď klientovi. V přístupové cestě lze definovat argument, který obslužné funkce mohou zpracovat. Tyto funkce jsou definovány v modulu `controllers` aplikace. Každý controller definuje určitou funkcionalitu aplikace a je ideálně vázaný právě k jednomu modelu aplikace. Cílem controlleru je pomocí modelové vrstvy uložit, modifikovat či získat data v závislosti na přijatém požadavku od klienta.

Kód 5.18: Definice cest frameworku ExpressJS.

```
1 router.get('/apps', AppsController.get);
2 router.post('/apps', AppsController.create);
```

#### Middlewares

ExpressJS middleware je funkce, kterou lze určit k dané přístupové cestě. Middlware funkce se zavolá před vykonáním obslužné funkce cesty a lze v ní

požadavek předzpracovat, nebo ukončit chybou. Příkladem použití middlewaru může být kontrola autentifikace uživatele.

V mé práci používám tyto middlewary právě k ověření uživatele a na validační účely (například pokud má uživatel právo přistoupit k danému testovacímu scénáři). Díky tomu mohou být obslužné funkce stručné a přehledné.

Kód 5.19: Příklad definice cesty pro získání informace o aplikaci včetně validace zda-li aplikace uživateli náleží.

```
async function findApp(req, res, next) {
    const app = await App.findOne<App>({ where: { id: req.
        params.appId, userId: res.locals.user.id }});
    if (!app) {
        throw new NotFoundError();
    }
    res.locals.app = app;
    next();
}

function show(req, res) {
    res.json({app: res.locals.app});
}

router.use('/apps/:appId', findApp);
router.get('/apps/:appId', show);
```

### 5.4.2 Odchytávání chyb

V případě, že v průběhu odbavování požadavku nastane chyba, je žádoucí klientovi odeslat vhodné chybové hlášení.

Nejjednodušší řešení je odesílat chybové stavy ve stejném formátu. Toto řešení není příliš vhodné z důvodu duplicity kódu – každá metoda by musela být obalena blokem `try catch` a v případě změny struktury chybového hlášení by bylo nutné upravit všechny výskyty formátování chyby.

Kód 5.20: Příklad definice oblužné funkce cesty.

```
// obaleni obsluzne funkce
const create = errorCatcher(async (req, res) => {
    // ziskani zadane aplikace z databaze
    const app = await App.create<App>({
      name: req.body.name,
      description: req.body.description,
      userId: res.locals.user.id
    });
    // odeslani odpovedi
    res.json(app);
});
```

Mé řešení je middleware `errorHandler()`, který je zavolán v případě chyby a v závislosti na typu chyby odpoví klientovi vhodným způsobem. Obslužná funkce v případě chyby vytvoří instanci třídy chyby a zavolá ExpressJS funkci `next()` (s argumentem vytvořené instance chyby). Implementaci tohoto middlewaru lze vidět v kódu 5.21. V kódu si lze všimnout, že v případě vyvolání vyjímky typu `ValidationError` se odpověd klientovi přetransformuje na pole chyb s chybovou hláškou a cestou chyby. Tuto vlastnost využívám pro serverové validování odesílaných formulářů. Chybu typu `ValidationError` totiž vyvolává sama knihovna Sequelize při porušení definovaných validačních pravidel v průběhu ukládání nebo modifikování dat v databázi. Klient tedy snadno po odeslání nevalidního formuláře může zjistit, jaké položky formuláře nejsou validní a následně jej uživateli zvýraznit i s chybovou hláškou.

Kód 5.21: Implementace middlewaru errorHandler.

```
const errorHandlerMiddleware = (err, req, res, next) => {
  // vychozi status code
  let statusCode = 500;

  // v pripade ze chyba obsahuje status code, prepiseme ho
  if (err instanceof ErrorWithStatus) {
    statusCode = err.status || statusCode;
  }

  let message = err.message;

  // pokud je chyba typu ValidationError, upravime strukturu
     odpovedi
  if (err instanceof ValidationError) {
    statusCode = 422;
    const castedErr = <ValidationError> err;
    const errors = [];
    _.each(castedErr.errors, (error) => {
      errors.push(_.pick(error, ['message', 'path']));
    });
    message = errors;
  }

  // odeslani odpovedi
  res.status(statusCode);
  res.json({
    code: statusCode,
    message
  });
};
```

Toto řešení jsem ještě vylepšil obalovou funkcí `errorCatcher()`. Její implementace je na obázku 5.22. Tato funkce přijmá jako argument obslužnou funkci cesty, kterou následovně zavolá. Pokud zavolaná funkce vyvolala vyjímku, obalová funkce `errorCatcher()` vyjímku zachytí a zavolá ExpressJS funkci `next()` s argumentem chyby, která chybu předá následujícímu middleware. Při definici obslužné funkce je tudíž pouze nutné zavolat funkci `errorCatcher()` s argumentem chtěné osblužné funkce. Příklad této definice je v kódu 5.20. Můžeme zde vidět, že se vyjímka se nemusí vyvolávat bezprostředně v obslužné funkci, ale i z funkcí volaných z ní. Pokud v tomto případě nastane při volání funkce `App.create()` chyba (například při porušení Sequelize validačního pravidla), middleware `errorHandler()` ji vždy zachytí.

Kód 5.22: Obalová funkce obslužné metody routy zachycující vyhozené vyjímky.

```
1 const errorCatcher = (genFn) => {
2   return (req, res, next) => {
3     genFn(req, res, next).catch(next);
4   };
5 };
```

### 5.4.3 Autentifikace uživatele

Pro autentifikaci uživatele využívám knihovnu *passport*. To je modulární nástroj, který nabízí mnoho typů přihlašování (takzvaných strategií). Program, který používá knihovnu passport, lze velmi snadno rozšířit o nový způsob přihlašování z dostupných strategií, jako je například Facebook nebo Twitter [3]. Pro mou aplikaci využívám přihlašování pomocí uživatelského jména a hesla s využitím nástroje JWT (JSON Web Token). Proces autentifikace uživatele je implementován v souboru
`backend/src/modules/Authentication.ts`.

### 5.4.4 Dotazy do relační databáze

Při práci s relační databází používám knihovnu Sequelize. Definice modelů této knihovny jsou popsány v kapitole 5.2.1. Práce s modely (jako například založení nebo editace záznamu) jsou zobrazeny v kódech 5.23 a 5.24.

Kód 5.23: Vytvoření nového záznamu v databázové tabulce.

```
1 await TestDefinition.create<TestDefinition>({
2     name: req.body.name,
3     appId: res.locals.app.id,
4     url: req.body.url
5 });
```

Kód 5.24: Dotaz do databáze.

```
1 const testDefinitions = await TestDefinition.findAll<
    TestDefinition>(
2     {
3         where: {
4             appId: res.locals.app.id
5         }
6     }
7 );
```

## 5.5 Scheduler

Scheduler plánuje testové scénáře. V relační databázi hledá takové testy, které mají znovu proběhnout. Testy vybírá na základě položky `next_run` v tabulce `scenario`. Id těchto testů poté vloží do fronty Redis. Po vložení do fronty se vypočítá čas dalšího provedení a ten se uloží zpět do databáze. Smyčka, která spouští kód plánovače je zobrazena v kódu 5.25. V této smyčce je použit `setInterval`, který vykonává vloženou funkci periodicky po zadaném intervalu (narozdíl od `setTimeout`, který nejprve vykoná funkci a až poté čeká daný interval). Proměnná `processing` vyjadřuje, zda se právě plánují testy. Pokud se obslužná funkce `setInterval` zavolá před tím, než se stihnou testy vložit do fronty a přeplánovat, provedení následujícího plánovaní se přeskočí z důvodu zamezení zahlcení plánovače.

Scheduler je závislý pouze na relační databázi a Redisu. Díky tomu může být spuštěn na jiném stroji, než na kterém beží server. Celou implementaci lze nalézt v souboru `scheduler/src/scheduler.ts`.

Kód 5.25: Smyčka plánovače.

```
1 const intervalCallback = () => {
2   if (!processing) {
3     schedulerTick();  // Provedeni planovani
4   } else {
5     console.info('scheduler tick missed');
6   }
7 };
8
9 setInterval(intervalCallback, 1000);
```

## 5.6 Worker

Worker vybírá testy k provedení z databáze Redis, které mu připravil plánovač (viz kapitola 5.4.4).

Redis je jednovláknová aplikace, díky čemuž je zaručeno, že každá provedená operace je atomická [4]. Díky této vlastnosti může být spuštěno více workerů na více strojích, kteří budou vybírat testy ze stejné fronty.

Každý testový scénář probíhá následovně:

1. Získání testů v daném scénáři
2. Provedení testů
3. Uložení výsledků do databáze Elasticsearch

### 5.6.1 Provádění testů

Provedení jednoho testu v testovém scénáři probíhá dle následujících kroků:

1. Vyplnění URL, hlaviček a těla požadavku specifikovanými proměnnými
2. Odeslání požadavku na server a získání odpovědi
3. Validace odpovědi a uložení specifikovaných proměnných

#### Vyplnění proměnných v požadavku

URL, hlavičky nebo tělo požadavku může obsahovat proměnnou, která se definovala v jakémkoliv předchozím testu jako odpověď serveru. Vyplnění proměnných má na starost metoda `fillVariables()`, která nahrazuje výskyty proměnných jejich hodnotou. V případě definice proměnné v tělu požadavku rekurzivně prohledává výskyt použití proměnných v polích a objektech.

#### Odeslání požadavku na server a získání odpovědi

Jakmile máme nahrazené výskyty proměnných za jejich hodnoty, odesílá se požadavek na server. V případě, že volání neskončilo chybou nebo timeoutem, následuje validace odpovědi. Pokud nastane chyba, ukončuje se proces testování scénáře.

### Validace odpovědi a uložení specifikovaných proměnných

Struktura definice pravidel je popsána v kapitole 4.3.2. Validaci odpovědi řeší funkce `validateResponse()`. Tato funkce rekurzivně vyhodnocuje odpověď serveru dle specifikovaných pravidel. V případě primitivního datového typu tato funkce vrací chybu, nebo indikuje správnost validace. V případě definového `exact` validačního pravidla funkce navíc kontroluje, zda se v objektech nenachází jiné klíče, než které validační pravidla specifikují. U validace typu pole nebo objekt funkce rekurzivně kontroluje validitu vnořených typů. Funkce také uloží výsledek do proměnné, pokud se v definici validace nachází klíč `saveto`. Část implementace této funkcionality lze vidět v kódu 5.26.

U validace využívám knihovnu Lodash pro porovnávání typů (při kterém má JavaScript problémy).

Kód 5.26: Implementace funkce validateResponse.

```
function validateResponse(response, rule, variables, exact) {
    if (!rule || Object.keys(rule).length === 0) {
        return exact ? 'rule is missing (exact type)' : null;
    }
    let error = false;
    if (rule.saveto) {
        variables[rule.saveto] = response;
    }
    switch (rule.type) {
        case 'string':
            return _.isString(response) ? null : 'field is not a string';
        case 'number':
            return _.isNumber(response) ? null : 'field is not a number';
        case 'boolean':
            return _.isBoolean(response) ? null : 'field is not a boolean';
        case 'null':
            return _.isNull(response) ? null : 'field is not null';
        case 'array':
            return validateArray(response, rule.item, variables, exact);
        case 'object':
            return validateObject(response, rule.item, variables, exact);
        default:
            return 'Undefined rule type';
    }
```

```
25 }
```

### 5.6.2 Uložení výsledků

V průběhu provádění testů se postupně ukládájí požadavky a odpovědi volání. Po úspěšném dokončení testového scénáře (nebo po výskytu chyby) se uloží výsledky testů do databáze Elasticsearch. Tyto výsledky obsahují hodnoty proměnných i všechny požadavky a odpovědi serveru.

## 5.7 Logování chyb

K logování chyb byl použit nástroj `Sentry`. Ten v reálném čase zaznamenává chyby v aplikaci. Sentry nabízí možnost odesílání notifikací (email, sms, atd.) při výskytu závady. V přehledu chyby lze nalézt podrobné informace, jako například část kódu, kde vznikla chyba, obsahy proměnných, obsah těla požadavku na server nebo data o uživateli, u kterého chyba nastala. Tento přehled je konfigurovatelný a lze k němu přidat další informace.

Pomocí Sentry můžeme velmi rychle detekovat problémy v aplikaci a díky tomu nemusíme spoléhat na uživatele, kteří by tyto problémy hlásily. Naopak se můžeme s postiženým uživatelem spojit a poskytnout mu podporu.

V aplikaci je použita Sentry v následujících případech:

- Nečekaný výskyt chyby na serveru
- Chyba při ukládání testového hlášení do Elasticsearch po provedení scénáře
- Chyba při práci s nástrojem Redis

Příklad zachycení chyby při ukládání dat do Elasticsearch je ukázán v kódu 5.27. Uživatelské rozhraní Sentry při výskytu chyby lze vidět na obrázku 5.3.

Kód 5.27: Zachycení chyby a její odeslání do Sentry.

```
// ziskani dat, ktere se budou ukladat do Elasticsearch
const data = getData();
elasticClient.index(
    {
        index: 'report',
        type: 'report',
        body: data
    },
    (err) => {
        if (err) {
            // zachyceni chyby a jeji zalogovani do Sentry
            Raven.captureException(err);
        }
    }
);
```

Obrázek 5.3: Uživatelské rozhraní Sentry.

# Kapitola 6

# Závěr

V rámci této práce jsem porovnal nástroje pro definice testů API – Runscope, Assertible a Postman. Popsal jsem jejich rozdíly a využití.

Dále jsem zpracoval požadavky, které jsou potřeba pro implementaci takové aplikace. Také jsem shrnul možnosti vývoje dynamických webových aplikací v jazyce JavaScript. Jsou zde popsány jeho výhody a využití jak v klientské, tak v serverové části. Zde je v neposlední řadě zmíněn JavaScript preprocesor TypeScript, který ho rozšiřuje o statické typování. Dalším požadavkem aplikace bylo vhodné úložiště pro různorodá data. V práci jsou navrženy 3 databázové systémy – PostgreSQL, Redis a Elasticsearch.

V implementační části práce jsou popsány moduly aplikace. Zmíněn je společný modul, který využívají všechny ostatní části aplikace. V rámci klientské části jsou popsány knihovny, které jsem při vývoji použil. Důraz je kladen na implementaci interaktivních diagramů, které se využívají pro definice testových scénářů. U implementace serverové části aplikace je zmíněna například knihovna ExpressJS, kterou jsem při programování použil. Dále je popsána implementace plánovače a workerů, které se starají o provádění testů.

Cílem práce bylo naprogramovat nástroj, který umožní vizuálně konfigurovat testové scénáře API. Tento nástroj jsem dle zadání úspěšně vytvořil. Program by bylo možné rozšířit o detailnější statistiky s grafy dostupnosti, nebo by bylo vhodné notifikovat uživatele při selhání testů.

# Literatura


[1] *The Benefits of Server Side Rendering Over Client Side Rendering* [online]. 2018. [cit. 23.4.2018]. Dostupné z: https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8.

[2] *TypeScript decorators* [online]. 2018. [cit. 23.4.2018]. Dostupné z: http://www.typescriptlang.org/docs/handbook/decorators.html.

[3] *Passport providers* [online]. 2018. [cit. 23.4.2018]. Dostupné z: http://www.passportjs.org/docs/facebook/.

[4] *The little Redis book, (s. 17)* [online]. 2018. [cit. 23.4.2018]. Dostupné z: http://openmymind.net/redis.pdf.

[5] *Amazon Web Services – auto scaling* [online]. 2018. [cit. 26.4.2018]. Dostupné z: https://aws.amazon.com/about-aws/whats-new/2018/01/introducing-aws-auto-scaling/.

[6] *Node.JS – JavaScript runtime engine* [online]. 2018. [cit. 26.4.2018]. Dostupné z: https://nodejs.org/en/.

[7] *JavaScript: The Event Loop* [online]. 2018. [cit. 26.04.2018]. Dostupné z: https://hackernoon.com/understanding-js-the-event-loop-959beae3ac40.

[8] *Kibana: Elasticsearch data vizualizer* [online]. 2018. [cit. 26.04.2018]. Dostupné z: https://www.elastic.co/products/kibana.

[9] *Time Travel in React Redux apps using the Redux DevTools* [online]. 2018. [cit. 26.04.2018]. Dostupné z: https://medium.com/the-web-tub/time-travel-in-react-redux-apps-using-the-redux-devtools-5e94eba5e7c0.

[10] *Understanding the Node.js event loop* [online]. 2018. [cit. 26.04.2018]. Dostupné z: http://abdelraoof.com/blog/2015/10/28/understanding-nodejs-event-loop.

[11] *Node.js's npm Is Now The Largest Package Registry in the World* [online]. 2017. [cit. 18.12.2017]. Dostupné z: https://developers.slashdot.org/story/17/01/14/0222245/nodejss-npm-is-now-the-largest-package-registry-in-the-world.

[12] *ReactJS* [online]. 2017. [cit. 18.12.2017]. Dostupné z: https://reactjs.org/.

[13] *Redux – Three Principles* [online]. 2018. [cit. 17.4.2018]. Dostupné z: https://redux.js.org/introduction/three-principles.

[14] *Specifics of npm's package.json handling* [online]. 2018. [cit. 23.4.2018]. Dostupné z: https://docs.npmjs.com/files/package.json.

[15] *Single-page application vs. multiple-page application* [online]. 2018. [cit. 23.4.2018]. Dostupné z: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.

[16] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* UNIVERSITY OF CALIFORNIA, 2000.

[17] Provos, D. N. M. *A Future-Adaptable Password Scheme.* USENIX Annual Technical Conference, 1999.

# Uživatelský manuál

## Sestavení projektu

Pro spuštění projektu je zapotřebí běhové prostředí Node.JS verze `8.4.0` a nainstalovaný nástroj `yarn`. Stažení závislostí probíhá spuštěním příkazu `yarn install` v kořenovém adresáři projektu.

### Konfigurace

Nejprve je nutné provést konfiguraci projektu. Ta se definuje v souborech `config/config.json` a `config/database.json`. Oba soubory mají ukázkovou konfiguraci v souborech se stejným názvem, které mají příponu `.example`. Před konfigurací je nutné mít tajný klíč k loggeru Sentry (popsaný v kapitole 5.7) a nainstalované nástroje `PostrgreSQL`, `Redis` a `Elasticsearch`, které se v těchto souborech konfigurují. Dále je nutné spustit příkaz `yarn elastic-mapping`, který nadefinuje strukturu Elasticsearch a `yarn migrate` pro inicializaci relační databáze.

### Spuštění

Spuštění modulů projektu probíhá následujícími příkazy:

- Server – Příkaz `yarn bstart`
- Klient – Příkaz `yarn fstart`
- Scheduler – Příkaz `yarn sstart`
- Worker – Příkaz `yarn wstart`

# Seznam obrázků

2.1 Grafická aplikace Postman. . . . . . . . . . . . . . . . . . . . . 8
2.2 Webová aplikace Runscope. . . . . . . . . . . . . . . . . . . . . 9

3.1 Grafické znázornění event loop [10]. . . . . . . . . . . . . . . . 12

4.1 Architektura aplikace. . . . . . . . . . . . . . . . . . . . . . . 18
4.2 Struktura databáze. . . . . . . . . . . . . . . . . . . . . . . . 19

5.1 Základní struktura projektu. . . . . . . . . . . . . . . . . . . . 24
5.2 Struktura komponent diagramu. . . . . . . . . . . . . . . . . . 34
5.3 Uživatelské rozhraní Sentry. . . . . . . . . . . . . . . . . . . . 47

# Seznam algoritmů

3.1 Definice React elementu zápisem syntaxe JSX. . . . . . . . . 13
3.2 Definice React elementu zápisem syntaxe JavaScriptu. . . . . 13
3.3 Použití prop `children`. . . . . . . . . . . . . . . . . . . . . . 14
3.4 Definice TypeScript rozhraní a jeho použití. . . . . . . . . . 15
5.1 Ukázka souboru config.json ve kterém je konfigurace prostředí development. . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.2 Ukázka konfigurace databáze PostgreSQL pro prostředí development. . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.3 Ukázka definice modelu pomocí knihovny sequelize-typescript. 27
5.4 Definice migrace pomocí knihovny Sequelize. . . . . . . . . . 28
5.5 Redux action creator. . . . . . . . . . . . . . . . . . . . . . . 29
5.6 Redux action creator s využitím TypeScript enum. . . . . . 29
5.7 Odeslání Redux akce. . . . . . . . . . . . . . . . . . . . . . . 29
5.8 Reducer funkce. . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.9 Použití vnořených reducer funkcí. . . . . . . . . . . . . . . . 31
5.10 Příklad použití Redux thunk. . . . . . . . . . . . . . . . . . 31
5.11 Příklad použití funkce connect knihovny React Redux. . . . 33
5.12 Příklad události onDragStart. . . . . . . . . . . . . . . . . . 35
5.13 Příklad události onDrop. . . . . . . . . . . . . . . . . . . . . 35
5.14 Ukázka definice rout knihovny React Router. . . . . . . . . . 36
5.15 Metoda render komponenty EnsureLoggedInContainer. . . . 36
5.16 Definice rout přihlášeného uživatele komponenty Dashboard. 37
5.17 Ukázka funkce action creator která po přihlášení přesměruje uživatele na výchozí stránku. . . . . . . . . . . . . . . . . . 37
5.18 Definice cest frameworku ExpressJS. . . . . . . . . . . . . . 38
5.19 Příklad definice cesty pro získání informace o aplikaci včetně validace zda-li aplikace uživateli náleží. . . . . . . . . . . . . 39
5.20 Příklad definice oblužné funkce cesty. . . . . . . . . . . . . . 40
5.21 Implementace middlewaru errorHandler. . . . . . . . . . . . 41
5.22 Obalová funkce obslužné metody routy zachycující vyhozené vyjímky. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.23 Vytvoření nového záznamu v databázové tabulce. . . . . . . 42
5.24 Dotaz do databáze. . . . . . . . . . . . . . . . . . . . . . . . 43
5.25 Smyčka plánovače. . . . . . . . . . . . . . . . . . . . . . . . 43
5.26 Implementace funkce validateResponse. . . . . . . . . . . . . 45
5.27 Zachycení chyby a její odeslání do Sentry. . . . . . . . . . . 47