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

# Bakalářská práce

# Modulární monitorovací aplikace

Plzeň 2017 Jakub Šmaus

**Místo této strany bude zadání práce.**

# Prohlášení

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

V Plzni dne 4. května 2017

Jakub Šmaus

# Abstract

The goal of this thesis is a survey of the AngularJS, Node.js, Elasticsearch, and RIAK technologies, and the creation of a modular monitoring application for tracking the execution of applications based on the mentioned above technologies. The work includes a requirements specification document, according to which the application is created. The first part of the thesis consists of the introduction to the basics of the technologies mentioned above. The second part presents the design and the implementation of the monitoring application, as well as the evaluation of the results.



# Abstrakt

Cílem této práce je přehledové prostudování technologií AngularJS, Node.js, Elasticsearch a RIAK a vytvoření modulární monitorovací aplikace pro sledování běhu aplikací skládajících se z výše uvedených technologií. V rámci práce byl vytvořen dokument specifikace požadavků, podle kterého je aplikace vytvořena. První část práce tvoří seznámení s podstatou výše uvedených technologií. Ve druhé části práce je prezentován samotný návrh a implementace monitorovací aplikace a zhodnocení dosažených výsledků.

# Obsah

# 1 Úvod

Monitorování běhu aplikací je proces, který zjišťuje, zda softwarová aplikace funguje korektně. Tato technika průběžně měří a vyhodnocuje stav aplikace a následně poskytuje prostředky, kterými ukáže případné nedostatky či abnormality.

Aplikace pro monitorování poskytuje vývojářům a administrátorům v přehledné podobě data, díky nímž snadno odhalí a izolují problém, který má negativní dopad na běh celé aplikace. Když je monitorovaná aplikace složená z mnoha funkčních komponent, může to být obtížné a právě proto je velmi důležité poskytnout data o chybě co možná nejvíc přehledně.

Cílem této práce je vytvoření modulární monitorovací aplikace, která kontroluje běh aplikací založených na rozdílných technologiích. Modulární se rozumí to, že bude při návrhu kladen důraz na rozdělení funkčnosti na nezávislé, zaměnitelné moduly a tím bude usnadněno přidání další monitorované komponenty.

Dále bude v této práci věnována pozornost technologiím potřebným pro vytvoření takovéto aplikace. Mezi to patří především metodika tvorby SPA. Díky té je možné vytvoření webové aplikace podobné k aplikaci desktopové. Dalšími důležitými technologiemi jsou AngularJS, Node.js, Elasticsearch a RIAK.

# 2 Technologie

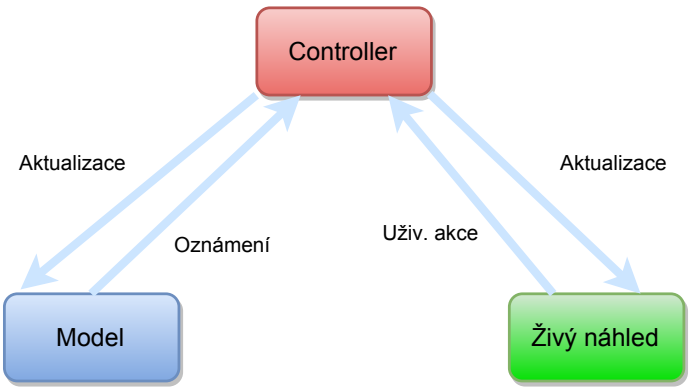
## 2.1 SPA

### 2.1.1 Struktura SPA

Single-page aplikace (dále **SPA**) je webová aplikace, kde není nutné při změně zobrazovaných dat stránku obnovovat, díky čemuž poskytuje lepší uživatelský komfort a zároveň i výkon. SPA řeší otázku výkonu oproti tradičnímu pojetí webových stránek jiným způsobem. Přesouvá veškerou logiku z databáze a serveru na klientskou část. Protože je tímto část webové aplikace přesunuta na prohlížeč, požadavky pro server jsou značně sníženy. [39]

### 2.1.2 Model View Controller

Model View Controller (dále **MVC**) je populární návrhový vzor, která rozděluje aplikaci na datový model, uživatelské rozhraní a řídicí logiku do tří nezávislých komponent tak, že modifikace některé z nich má jen minimální vliv na ostatní. [30]




Obrázek 2.1: MVC model

**Model** (česky model) – Doménově specifická reprezentace informací, s nimiž aplikace pracuje.

**View** (česky pohled) – Převod dat reprezentovaných modelem do podoby vhodné k interaktivní prezentaci uživateli.

**Controller** (česky kontroler) – Reaguje na události (typicky přicházející od uživatele) a zajišťuje změny v modelu nebo v pohledu.

### 2.1.3 AJAX

AJAX (zkratka z `asynchronous JavaScript and XML`) je soubor technologií používaných k tvorbě asynchronních webových aplikací. Odpovědí ze serveru je objekt, který má tyto atributy: `config` (objekt, jenž generuje požadavek), `data` (odpověď serveru), `headers` (obsahuje informace o headeru), `status` (číslo definující HTTP status) a `statusText` (textový řetězec definující HTTP status). [1]

Při vysílání AJAX požadavků je zapotřebí si vybrat, jakým způsobem má být tato komunikace řešena. V úvahu přicházejí dvě technologie, a to JSONP a CORS.

JSONP je starší a díky tomu podporuje starší verze prohlížečů. Jeho další výhodou je skutečnost, že dovoluje volání API i na jiné doméně, než má vlastní aplikace. Na druhou stranu je tímto výrazně ohrožena bezpečnost aplikace. Další nevýhodou je to, že je možné zasílat jen GET požadavky, a to pro potřeby této aplikaci nestačí. [18]

CORS ve zkratce definuje způsob, pomocí kterého se prohlížeč a dotazovaný server dohodnou na tom, zda mezi-doménové volání bude povoleno. Takto lze docílit sdílení zdrojů, zachování bezpečnosti aplikace a zároveň možnost použití ostatních požadavků. [18]

### 2.1.4 REST

REST (`Representational State Transfer`) je architektura, jenž se využívá pro přístup k datům implementujícím vzdálené volání procedur. V jistém smyslu je tato architektura bezstavová, protože žádná informace týkající se klientových stavů není uložena on server. Tímto je docíleno toho, že každý dotaz může být obsloužený rozdílným systémem. [37]

REST definuje čtyři základní metody pro přístup ke zdrojům, které jsou popsány svými URI.[1] Těmito metodami jsou *Create* (vytvoření dat), *Retrieve* (získání dat), *Update* (aktualizace dat) a *Delete* (mazání dat), souhrnně označované jako *CRUD*. Jsou implementovány jako klíčová slova HTTP protokolu a to v tomto pořadí: POST, GET, PUT a DELETE.

[1]URI neboli Uniform Resource Identifier je řetězec znaků, který jednoznačně identifikuje zdroj a umožňuje tak komunikaci se zdrojem.

Zde uvedu příklad: (klíčové HTTP slovo a URI – identifikátor zdroje)

- **GET /api/get/instance:id** – vrátí danou instanci
- **POST /api/post/instance:id** – vloží novou instanci
- **PUT /api/put/instance** – aktualizuje instanci
- **DELETE /api/delete/instance:id** – vymaže instanci

Tyto metody umožňující přístup ke zdrojům se nazývají *koncové body* (angl. endpoints).

### 2.1.5 Nevýhody SPA

#### Pomalé načítání

Ještě předtím, než se poprvé načte stránka SPA, musí se načíst všechny soubory frameworku. Vzhledem k tomu, že tyto aplikace většinou tvoří velké množství souborů, může to způsobovat jistý uživatelský diskomfort.

#### Omezená možnost pohybování se v historii

Už z definice SPA je zjevné, že adresa URL zůstává stále stejná, nezávislá na akcích uživatele nebo konkrétní zobrazené stránce. To způsobuje, že v historii prohlížeče se zaznamenává pouze tato jedna adresa a v rámci SPA se není možné vracet nebo jít dále pomocí tlačítek *Zpět* a *Vpřed*. Tento problém je způsoben hlavně tím, že technologie SPA vznikla až o mnohem později, než se vyvíjely webové prohlížeče. Částečně jej lze tedy odstranit používáním novějších technologií, například HTML5. Zde se jedná hlavně o metody `pushState()` a `replaceState()`, které dokáží měnit historii prohlížečů.

#### Automatické testování

Používání automatického testování u SPA problematizují především asynchronní požadavky na data. To zapříčiňuje také to, že je velmi pravděpodobné, že splnění daného testu bude trvat déle než u běžné webové stránky.

#### Nutnost jen jedné verze knihovny

V SPA lze mít pouze jednu verzi od každé knihovny. Ve chvíli, kdy potřebujeme kvůli nějaké komponentě určitou knihovnu aktualizovat, může to způsobit změny v chování různých částí aplikace, které jsou na ní závislé. Minimálně je vždy po každé aktualizaci potřeba celou aplikaci znovu otestovat. [3]

### 2.1.6 Výhody SPA

#### Rychlejší odezva

U klasické webové stránky je při načítání nové stránky a jejím přenačítání zapotřebí poměrně mnoho času na překlad na serveru a přenesení dat. Pokud však chceme přenačíst stránku u SPA, není potřeba se díky oddělení modelu a pohledu znovu dotazovat serveru na HTML kód. Stránku tedy načítáme pouze jednou a celkový objem potřebných dat je menší, než by byl u klasické webové aplikace. Díky tomu je i doba odezvy na akci uživatele kratší. [9]

#### Snazší vývoj mobilní aplikace

SPA má tu výhodu, že je velmi responzivní a dokáže se tedy lehce přizpůsobovat velikosti uživatelova přístroje. Mimo to se stránka velikosti zařízení přizpůsobuje pouze jednou, nemusí se znovu přizpůsobovat pokaždé, když uživatel chce zobrazit jinou část této stránky. To je dáno způsobem načítání popsaným výše.

V případě, že webová aplikace má mít i svoji mobilní podobu, lze u SPA využít stejný backend z webové aplikace. [4] [6]

#### Zobrazení offline

V některých případech lze SPA zobrazit i offline. Například když si uživatel jednou stránku načte a poté je bez přístupu k internetu, může stránku používat i nadále, jelikož jsou všechna důležitá data již načtena. [6]

## 2.2 AngularJS

### 2.2.1 Úvod do AngularJS

AngularJS je open-source JavaScriptový framework, který je určen pro především pro tvorbu webových single-page aplikací. Kód je organizován podle architektur Model–View–Controller (ref. 2.1.2) nebo Model–View–Viewmodel (MVVM)[2]. AngularJS také obsahuje mnoho komponent užívaných k tvorbě aplikací, které připomínají klasický desktopový software.

AngularJS funguje tím způsobem, že jsou do HTML kódu vloženy speciální formátovací značky (tzv. direktivy), které určují, jaké operace či data

[2]MVVM usnadňuje oddělnení backendu a frontendu. ViewModel poskytuje data a funkčnost, ale již neobsahuje její kód.

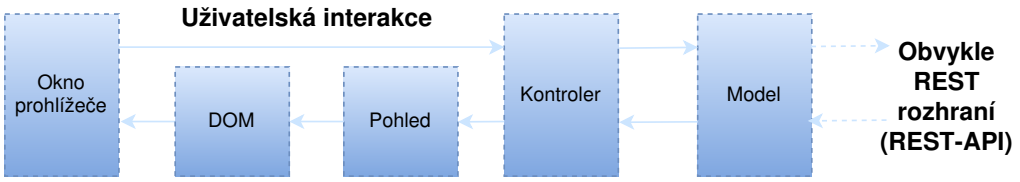
mají být na dané místo vloženy.[11]

Níže jsou uvedené vlastnosti a výhody, kterými tento framework disponuje.[7]

- Flexibilita: Díky rozdělení do MVC architektury jsou komplexní aplikace přehlednější a snadno se přidává další funkcionalita pomocí externích modulů.
- Testovatelnost: AngularJS podporuje jednotkové a end-to-end testování, které překonává tradiční formu testování webových aplikací vytvářením jednotlivých testovacích stránek.
- Standardizovanost: AngularJS pomáhá vytvářet webové aplikace, jenž používají nejnovější technologie.

### 2.2.2 Architektura

Protože AngularJS pracuje na prohlížeči, dochází k lehkému pozměnění MVC architektury.




Obrázek 2.2: MVC model AngularJS

Jak je patrné z obrázku 2.2, implementace strany klienta přejímá data ze serverové strany Application Programming Interface (dále **API**). Účel kontroleru je zpracování dat modelu a pohled pak tyto data zobrazuje. Toto zpracování má za následek manipulaci s rozhraním reprezentujícím HyperText Markup Language (dále **HTML**) dokumenty (dále **DOM**) a rovněž vytvoření a spravování prvků značkovacího jazyka HTML, se kterými uživatel interaguje. Tyto interakce jsou vráceny zpátky kontroleru. Tímto je smyčka interagující aplikace uzavřena. I když aplikace v AngularJS používají MVC architekturu, zásadní komponenty závisí na širší paletě stavebních bloků. Tyto další důležité pojmy budou probrány níže. [34]

### 2.2.3 Základní prvky AngularJS

Zde je uveden výsek základních prvků a jejich popis. [10]

Tabulka 2.1: Základní prvky AngularJS

| **Koncept** | **Popis** |
|---|---|
| Šablona | HTML s dodatečným značkováním |
| Direktiva | Rozšíření HTML s atributy a elementy |
| Model | Data, která jsou zobrazena uživateli a se kterými pracuje |
| Kontext | Kontext, kde je model uložen |
| Expressions | Přístup k proměnným a funkcím z kontextu |
| Kompilátor | Analyzuje šablonu a konkretizuje koncepty direktiv a expressions |
| Filtr | Formátuje hodnotu výrazu pro zobrazení |
| Pohled | To, co uživatel vidí (DOM) |
| Vázání dat | Synchronizování dat mezi modelem a pohledem |
| Kontroler | Logika fungující před pohledem |
| Vkládání závislostí | Vytváří a spojuje objekty a funkce |
| Injector | Kontejner pro vkládání závislostí |
| Modul | Kontejner pro rozdílné části aplikace, které konfigurují Injector |
| Služba | Znovupoužitelná logika |

#### Modul

Modul (angl. Module) aplikace obsahuje veškerou logiku kódu – kontrolery, služby, filtry, direktivy a další. Aplikace AngularJS může mít pouze jeden, ale i více modulů, kde každý specifikuje určitou funkcionalitu. Tímto je docíleno, že jednotlivé moduly mohou být načteny v jakémkoliv pořadí a tímto je zjednodušené testování, jelikož pro jednotkové testy stačí načíst jen potřebné moduly.

Pro deklarování modulu se používá funkce `module()`. Ta má dva parametry: prvním je název modulu a druhým je pole modulů, na kterých je závislý. Příklad deklarace modulu: [5]

Ukázka kódu 2.1: Vytvoření modulu

```
var mujAplikacniModul = angular.module('mojeAplikace', []);
```

Tato funkce vrací instanci nově vytvořeného modulu. Referencí názvu tohoto modulu v HTML kódu aktivujeme AngularJS aplikaci:

Ukázka kódu 2.2: Aktivace AngularJS

```
<body ng-app='mojeAplikace'>
```

### Kontext

Kontext (angl. Scope) je základním prvkem angularových aplikací. Je to objekt, který souvisí s aplikačním modelem. Jedná se o prostřední vrstvu propojující kontroler a pohled. Šablona pohledu se připojí ke kontextu ještě předtím, než se aplikace zobrazí. Pokud dojde k nějaké změně v datech na kontextu, AngularJS je s tím obeznámen a aktualizuje DOM. [35], [8]

### Kontroler

Kontroler (angl. Controller) ovládá data aplikace. Může být buď obsažen v HTML souboru, kde je označen tagy `<script></script>`, nebo (a to je typické pro větší aplikace) může mít svůj vlastní čistě JavaScriptový soubor. Je připojen k DOM pomocí direktivy `ng-controller`. Při inicializaci nového objektu komponenty Controller se vytvoří nový kontext. Je to v podstatě JavaScriptový objekt, který může být vytvořen například takovýmto konstruktorem:

Ukázka kódu 2.3: Kontroler

```
var mojeAplikace angular.module('mojeAplikace',[]);
mojeAplikace.controller('MujKontroler', ['$scope',
    function($scope){
   $scope.pozdrav = 'Ahoj, svete!';
}]);
```

Ukázka kódu 2.4: Použití v pohledu

```
<div ng-controller="MujKontroler"> {{ pozdrav }} </div>
```

Kontroler může obsahovat také metody. Jedná se vlastně o proměnné, kterým je přiřazena nějaká funkce. Zde je příklad jednoduché metody:

Ukázka kódu 2.5: Kontroler s funkcí

```
$scope.mojeMetoda = function() {
      var vysledek;
      //kod metody
      return vysledek;
    };
```

### Služby

Služby (angl. Services) jsou objekty, které umožňují organizaci kódu v aplikaci. Od každé služby existuje pouze jedna instance, říkáme tedy, že jsou jedináčky. To znamená, že všechny komponenty aplikace, které na dané službě závisí, dostávají odkaz na jednu a tu samou instanci oné služby. Instance se však vytváří pouze v případě, že je na službě závislá alespoň jedna komponenta.

AngularJS má některé služby již vestavěné – ty vždy začínají symbolem `$` (např. `$http`). Při tvorbě aplikace je však velmi často vhodné vytvářet služby vlastní. To lze celkem čtyřmi způsoby, zde naznačím jeden z nich.

Ukázka kódu 2.6: Pomocí metody service

```
var module = angular.module('mojeAplikace', []);
module.service('vlastniServisa', function(){
   this.atributJmeno = 'jmeno';
});
```

### Direktivy

Direktivy (angl. Directives) jsou značky, které HTML elementu přiřazují určité chování nebo tento element nějakým způsobem mění. Jedná se v podstatě o rozšíření HTML atributů. AngularJS má již mnoho direktiv vestavěných – ty začínají předponou `ng`. Příkladem může být `ngApp` (inicializuje aplikaci), `ngInit` (inicializuje data aplikace) nebo `ngModel` (synchronizuje data z HTML elementů s aplikačními daty). Lze však vytvářet i direktivy vlastní.

Na direktivy většinou odkazujeme pomocí jejich normalizovaného camel case[3] názvu (např. `ngHide`). Ale vzhledem k tomu, že HTML je case sensi-

[3]Camel case je způsob psaní několikaslovných názvů, které jsou psané bez mezery a s výjimkou úplně prvního písmene je každé počáteční písmeno slova psáno velkým písmenem, ostatní písmena jsou malá (např. „camelCase“).

tive jazyk, tak zde na direktivy odkazujeme pomocí upraveného kebab case[4] názvu (např. `ng-hide`).

Pokud tvoříme direktivu vlastní, musíme ji nejprve zaregistrovat na modulu pomocí `module.directive`. Ta má dva parametry – název direktivy zadaný jako camel case a její tovární funkce. Takováto direktiva je vytvořena ve chvíli, kdy je tento kus kódu poprvé přeložen. Příklad definice vlastní direktivy:

Ukázka kódu 2.7: Direktiva

```
angular.module('mojeAplikace', [])
   directive('vlastniDirektiva', function () {
   return {
   restrict: 'A',
   link: function (kontext, atributy) {
      console.log("Vypis logu");
       }
   }
}
```

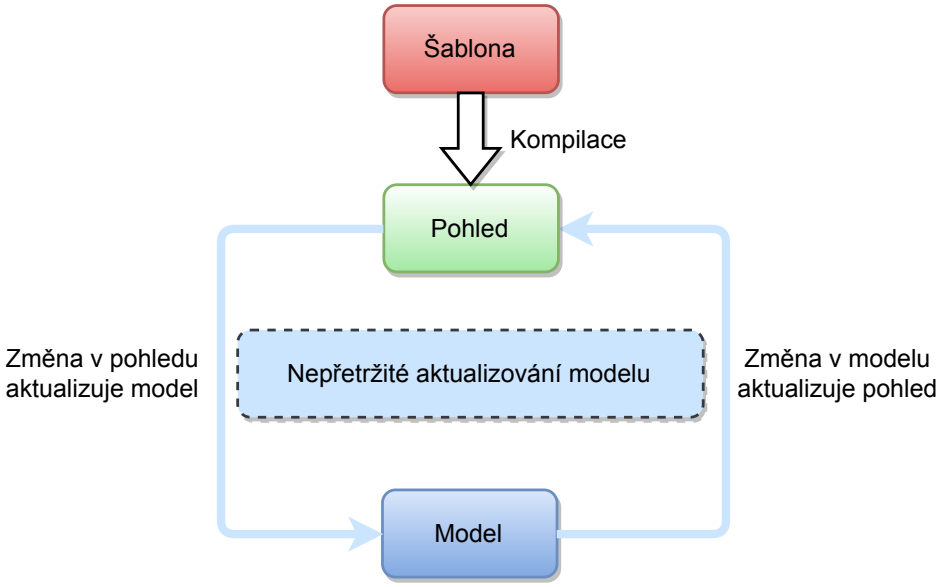
Na tomto příkladu je patrné, že při vytváření direktivy je nutné nastavovat určité parametry, v závislosti na tom, co od ní očekáváme.

Například atribut `restrict`, jak bude direktiva používána. Má čtyři možné hodnoty, které je ale možno navzájem kombinovat (`A` - atribut, `E` - element, `C` - třída, `M` - komentář).

### Vázání dat

Vázání dat (angl. Data binding) je v aplikacích AngularJS automatická synchronizace dat mezi modelem a komponentami pohled. Pohled je projekcí modelu v každém okamžiku. Jakmile se model změní, projeví se změny i v komponentách pohled a naopak. Na obrázku 2.3 je toto přehledně ukázáno.

[4]Kebab case je způsob psaní víceslovných názvů, kdy jsou všechna písmena malá a jednotlivá slova jsou oddělena pomlčkou (např. „kebab-case“).




Obrázek 2.3: Vázání dat

Způsob, jak šablony pracují, je oproti klasickému vázání dat odlišný. Nejprve se šablona (což je nezkompilovaný HTML) zkompiluje na prohlížeči. Tímto se vytvoří pohled. Jakékoliv změny v pohledu jsou reflektovány v modelu a naopak. Tímto je zjednušena práce s modelem, protože vždy obsahuje taková data, jaké má aktuální stav aplikace. Proto se dá na pohled pohlížet jako na okamžitou projekci modelu. [2]

### Vkládání závislostí

Vkládání závislostí (angl. Dependency injection) je funkcionalita, která zajišťuje správné vkládání závislostí mezi jednotlivými komponentami. Můžeme ji zajistit například pomocí dvou nejpoužívanějších technologií – AngularJS a RequireJS. Je však důležité si uvědomit rozdíl mezi nimi. Systém vkládání závislostí, který je zabudovaný v rámci AngularJS, řeší závislosti mezi objekty a komponentami, zatímco RequireJS se zabývá závislostmi modulů a JavaScriptových souborů.

### Filtry

Filtry se používají k formátování dat. Jsou buď již implementované v Angularu, ale uživatel si může vytvořit i vlastní. Již implementované jsou například `filter` (vytváří podmnožinu pole), `lowercase` (převede řetězec na malé znaky), `orderBy` (řadí pole).

### 2.2.4 Pomocné technologie

#### RequireJS

RequireJS [27] je framework pro mapování závislostí mezi jednotlivými komponentami, z nichž každá je uložena ve svém vlastním souboru.

RequireJS nejprve načítá pouze závislé moduly. Teprve poté se vykonává funkční část kódu. Pokud nějaký modul chybí, nevykonává se žádný kód.

#### Lodash

Lodash [20] je knihovna pro zjednodušené psaní některých funkcí. Jedná se například o práci s řetězci (převod na malá písmena, ořezávání ...), poli (komprimace, dělení ...), kolekcemi (iterace, dělení, řazení ...), objekty (rozšíření, sjednocení ...) apod. Načteme si jej pomocí funkce *require()* do proměnné většinou pojmenované jako „_“.

## 2.3 Node.js

Node.js je platforma navržená pro psaní vysoce škálovatelných internetových aplikací, především webových serverů. Používá událostmi řízený, neblokující **vstupně/výstupní** (dále **I/O**) model, díky němuž se hodí pro datově náročné real-time aplikace, běžící na distribuovaných zařízeních. Node.js je postaven na renderovacím jádře užívající programovací jazyk JavaScript. Díky tomu aplikace napsané v této platformě využívají podobnou syntaxi, jako frontendové JavaScript aplikace, včetně objektů a funkcí.

Mezi vlastnosti patří:

- **Asynchronní a událostmi řízený** – Celé API Node.js knihovny je asynchronní, tím pádem neblokující. V podstatě to znamená, že takový server nikdy nečeká na vrácení dat v API. Všechny vstupně/výstupní operace, jako například připojení se k databázi nebo práce se soubory, neblokují hlavní vlákno, ale pokračují až tehdy, když získají od protistrany data. [38]

- **JavaScript** – Postavení Node.js na JavaScriptu poskytuje možnost mít jak frontend aplikace, tak backend ve stejném programovacím jazyku. Nemusí zde docházet ke střídání mezi technologiemi a navíc je možné i používat stejný kód nebo knihovny napříč celou aplikací.

Mezi nevýhody patří:

- **Využití CPU** – Node.js není vhodný pro úkony, jež silně využívají CPU. Jeho síla spočívá právě pro I/O aplikace, jako jsou webové servery.

- **Nestabilní technologie** – I když se jedná o novou, perspektivní technologii, nelze se vyhnout tomu, že se mezi knihovnami, jež jsou stabilní, dá nalézt mnoho knihoven, které jsou ještě nekompletní a v experimentálním stavu.

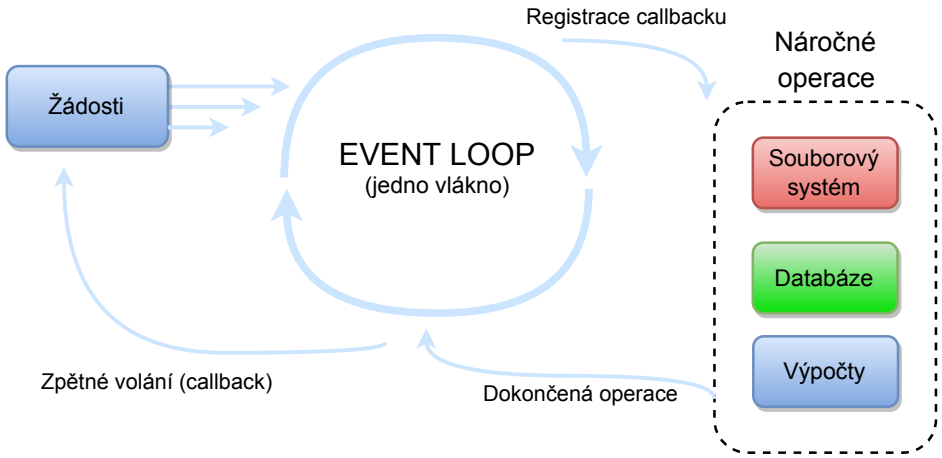
- **Problematická metodika** – Bez řádného pochopení volání a užívání událostí se může docílit nepřehledného a kvůli tomu i špatně napsaného kódu.

### 2.3.1 Událostní smyčka

To, čím se Node.js význačně liší od ostatních běhových prostředí a serverů je způsob reagování na příchozí události. Většina tradičních serverů (např. pro zapsání souboru nebo dotazování databáze) vnímá volání jako blokující. Tedy program se přeruší a vyčká se, až volání vrátí výsledek a tímto skončí. Po této události program znovu pokračuje. Současné servery řeší tuto problematiku vytvořením vlákna pro každou vzniklou událost, jenže toto řešení není optimální, když se jedná o aplikaci pro několik uživatelů. Node.js se vypořádává s tímto problém jiným způsobem.

Zpracovává každý požadavek pouze z jediného vlákna. Kód běžící v tomto vlákně běží synchronně, ale pokaždé, když nastane nový požadavek, bude tento požadavek přesunut do událostní smyčky, spolu s funkcí pro zpětné volání. Hlavní vlákno není uspáno a dále může zpracovávat požadavky. Jakmile je předchozí žádost dokončené, událostní smyčka spustí funkci pro jeho zpětné volání. [23]

Na obrázku 2.4 je toto přehledně ukázáno.




Obrázek 2.4: Událostní smyčka

### 2.3.2 Express.js

Express.js je webový framework, který se používá k vývoji aplikace na serverové straně. Samotný Node.js totiž nebyl původně vyvinutý pro vytváření webových stránek a neobsahuje tak pro tuto potřebu dostatek funkcionality. Tu Express.js dodává jako tzv. *middleware*. To je v obecné rovině software, který poskytuje funkce, jež nejsou běžně poskytované původní platformou. Například zde se jedná o již vytvořené komponenty JavaScriptu. Umožňují vývojáři použít pro svůj projekt již vytvořený a otestovaný kód, který řeší většinu základních problémů jako např. syntaktickou analýza HHTP požadavků nebo cookies[5], tvorba tras (angl. routes), vyjmutí parametru z *URL*, zvládání chyb a další. Express je založený na modulu *http* v Node.js a komponentách frameworku *Connect.js*. [36] [15]

Middleware jsou funkce, které se používají například ke spouštění kódu, ke změnám v objektech *response* a *request*, k ukončení cyklu spouštění dalšího middleware (tj. k tomu, že daná funkce po vykonání své činnosti nespustí funkci *next()*) nebo v pokračování ve vykonávání funkcí ze zásobníku. Mimo objekty *response* (res) a *request* (req) mají přístup i k funkci *next()*, která slouží k případnému spuštění dalšího middleware. Při každém zaslaném HTTP požadavku se spouští všechny middleware funkce, které jsou obsaženy v tzv. middleware zásobníku funkcí. Middleware lze připojit k aplikaci pomocí metod *app.use()* nebo *app.GET* a *app.POST*. [16]

[5]Cookie označuje v protokolu HTTP malé množství dat, které prohlížeč uloží do počítače.

Aplikace v Express.js se většinou skládá z těchto po sobě jdoucích částí:

- Vložení závislostí, popř. knihoven třetích stran
- Vytvoření objektu Express.js
- Konfigurace aplikace, nastavení proměnných aplikace
- Připojení k databázi
- Definování funkcí middleware
- Vytvoření tras
- Samotný start aplikace, nastavení portu a jména

### 2.3.3 NPM

NPM [22] neboli `Node Package Manager` slouží k šíření JavaScript kódu mezi vývojáři. Takovéto kódy se nazývají balíky (*packages*) nebo někdy také moduly. Pokud je v projektu využit takovýto „cizí“ balík, pomocí NPM je velmi snadné zkontrolovat, zda jsou k dispozici nějaké jeho aktualizace a pokud ano, tak jej i aktualizovat.

Balík je v podstatě adresář, který obsahuje jeden nebo několik souborů a zároveň i soubor `package.json` s informacemi o balíku. Hlavní ideou npm je, že z těchto malých bloků řešících vždy jeden problém lze sestavit velké množství funkčnosti aplikace.

### 2.3.4 Swagger

Modul `Swagger` obsahuje nástroje pro návrh, testování a překlad celého API v rámci Node.js. Jeho pomocí je možné popsat strukturu API tak, že je možné ji „přečíst“. Díky tomu je možné automatiky vytvořit interaktivní dokumentaci, jež poskytuje informace o funkcích rozhraní, které je dále možné ručně testovat. [29]

### 2.3.5 Node-schedule

Tento modul [24] je určen pro vykonávání určité funkcionality v určitý čas. Jeho použití je silně podobné tzv. `Cronu`, což je softwarový démon, který v operačních systémech automatizovaně spouští v určitý čas nějaký příkaz resp. proces.

Každý plánovaný proces v modulu node-schedule je reprezentován objektem `Job`. Tento objekt je možné vytvořit manuálně a poté zavolat funkci pro plánování a nebo je možné použít pohodlnější funkci `scheduleJob()`, která již objekt Job vytvoří sama. Nyní je nutné pouze určit, v jakém časovém rozpětí daný proces proběhne. To je možné jak způsobem, který používá Cron, tak i určením přesného času a nebo pravidla, podle kterého bude plánovaní spouštěné.

Pro ukázku vytvořeného plánovače jsem použil plánovaní používané v Cronu.

Ukázka kódu 2.8: Ukázka plánované akce

```
var planovac = require('node-schedule');

var j = planovac.scheduleJob('*/30 * * * *', function(){
  console.log('Tento radek se vypise kazdych tricet minut');
});
```

### 2.3.6 Testování a validace dat

#### Metodika testování

Standardy používané pro testování aplikací psaných v JavaScriptu jsou rozděleny především na metodiky Test-driven development (`TDD`) a Behavior-driven development (`BDD`). Proto je vhodné definovat, jaké jsou mezi nimi rozdíly. TDD, neboli ve volném překladu programování řízené testy a BDD, neboli chování řízené testy.

První a nejzásadnější rozdíl je samotný účel metodik. Programování řízené testy slouží k otestování implementace a chování řízené testy, jak už název napovídá, k otestování chování.

Druhým rozdílem je způsob zápisu. BDD se snaží připodobnit co nejvíce anglickému jazyku a díky tomu se stává lépe čitelnějším. V zásadě je metodika chování řízené testy vytvořena k eliminování problémů, při běžném vývoji nastat i přes opatření TDD.

U testování není nutné a ani není vhodné, aby obsahovalo pouze jedinou metodiku. Obě se vzájemně doplňují. [25]

### Mocha

Mocha [21] je knihovna, jež slouží jako prostředí pro vytváření a rovněž spouštění testů. Základním prvkem pro vytvoření testu je funkce `it()`. Ta obsahuje v parametrech svůj název a funkci, která se provede.

Pro seskupování jednotlivých testů slouží `describe()`. To může být libovolně do sebe vnořováno a díky tomu je možné vytvářet vhodně organizované testy.

Neméně důležité jsou funkce `before()` a `after()`, které obvykle připravují podmínky nutné pro úspěšné provedení testu. Jsou používány vždy před prvním zavoláním následujícího `describe()` a nebo `it()`.

### Chai

Chai [12] patří mezi několik knihoven JavaScriptu určených pro testování. Obsahuje funkce, které usnadňují ověřování výsledků testů. Jeho výhodou je oproti jiným, že poskytuje jak porovnání ve stylu BDD, tak ve stylu TDD. Pro porovnání chování se používá rozhraní `Expect` a `Should`. Pro porovnání implementace slouží rozhraní `Assert`.

### Joi – validace

`Joi` [19] je modul určený pro validaci dat. Dokáže ověřovat jednoduché datové typy až po složité, do sebe vnořené objekty JavaScriptu. Celý proces obsahuje čtyři kroky. V prvním kroku je nutné vytvořit objekt (schéma), který popisuje hodnoty, jež jsou očekávány. Oproti těmto hodnotám bude probíhat validace. Nyní je nutné získat testovaný objekt. Ten se následně otestuje proti schématu. Pokud nastane chyba a ukáže se, že testovaný objekt neodpovídá, Joi oznámi, proč chyba nastala.

## 2.4 Elasticsearch

Elasticsearch je nástroj sloužící jako úložiště dat a jako distribuovaný vyhledávací software pracující v reálným čase. Je to vyhledávací stroj založený na opensource knihovně Apache Lucene. Často se používá i jako NoSQL databáze, protože integruje fulltextové vyhledávání, databázový přístup a analytiku v reálném čase. Je používána pro fulltextové vyhledávání, strukturované vyhledávání a analýzu. To, čím je Elasticsearch zajímavý a zároveň revoluční, nejsou výše zmíněné části, ale jejich kombinace tvořící koherentní

aplikaci.

Většina databází je překvapivě neobratná v extrahování konkrétních znalostí z dat. Sice dokážou vyfiltrovat data podle určitých hodnot, ale již nedokáží zacházet se synonymy a nebo získat dokumenty podle relevantnosti. To a další odlišuje Elasticsearch od klasických databází. [32]

### 2.4.1 Architektura

#### Klustr

Seskupení jednoho nebo více uzlů, které drží všechna data pohromadě a umožňuje hromadné indexování a vyhledávání napříč všemi uzly. Klustr je pojmenován unikátním názvem. Toto jméno je důležité, protože uzel může být součástí klustru jen tehdy, kdy je uzel napojen na klustr správným jménem.[40]

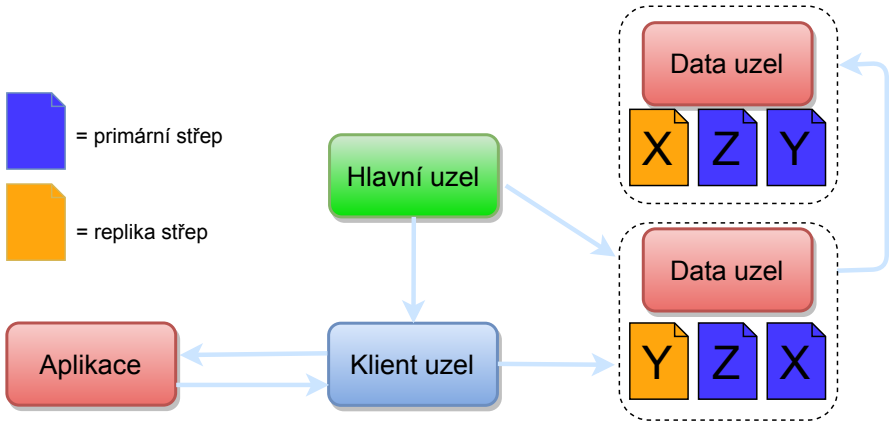
#### Node (uzel)

Jedná se o jednu běžící instanci, patřící jednomu klustru. Ukládá data a spolupracuje v indexování klustru a vyhledávání. Při startu hledá klustr, se kterým by se mohl spojit. Podobně jako klustr je uzel identifikován jménem, které je ve výchozím stavu určeno podle UUID[6], které je přiděleno uzlu při startu. Tento název je důležitý pro administrační účely, kdy je zapotřebí identifikovat servery náležící uzlům v klustrech.[40] Rozdělujeme tři typy uzlů:

- Hlavní uzel – je zvolen automaticky a řídí celý klustr.
- Data uzel – obsahuje data a vykonává operace nad nimi, jako například jejich vyhledávání, vytvoření, čtení, úpravu nebo agregaci.
- Klient uzel – používá se k přeposílání požadavků na úrovni klustru a požadavků souvisejících s daty do hlavní uzlu.
- Ingest uzel – používá se k předzpracování dokumentů před vlastním zpracováním [14]

Ukázka celého procesu je uvedena na obrázku 2.5

[6]UUID neboli Universally Unique IDentifier je 128-bitové číslo, jedná se o standard používaný pro jednoznačnou identifikaci objektu v softwarovém inženýrství




Obrázek 2.5: Elasticsearch proces

### Index

Index je kolekce dokumentů, jež mají určitým způsobem podobné charakteristiky. Index je identifikován názvem, na který je odkazováno při provádění indexace, vyhledávání, aktualizace a mazání dokumentů.
V podstatě se jedná o ekvivalent databáze v relačních databázích. Index obsahuje mapování, které definuje různé typy dokumentů. Index je rozprostřen na jeden nebo více *primárních střepech* a dále na žádný nebo více *replik střepů*. Jejich definice popsána v následující sekci. [40]

### Střep

Index může mít potenciálně uložen velké množství dat, které může převyšovat limit hardware pro jeden uzel. Aby tento problém nevznikal, Elasticsearch poskytuje další dělení indexu na více kousků zvaných střepy (shards). Při vytvoření indexu se dá definovat počet těchto střepů. Jejich vytváření se provádí především kvůli následujícím důvodům:

- Poskytuje horizontální dělení obsahu.
- Dovoluje rozdělení a paralelizací operací napříč střepy a tímto zvýšení výkonu.

Aby bylo myšleno i na chyby, které mohou nastávat, je v Elasticsearch mechanismus, který dokáže vyřešit problém, kdy se například uzel stane do stavu offline a nebo zmizí z jakéhokoliv důvodu. Tento mechanismus se nazývá replikace. Při ní je možné vytvořit jednu či více kopií střepu do tzv. replik střepů. Replikace je významná především díky následujícím schopnostem:

- Poskytuje vyšší možnost využití dat, když střep/uzel selže.
- Umožňuje zvýšení výkonu vyhledávání, když je možné vyhledávat ve všech replikách paralelním způsobem.

Každý index může být rozdělen na několik střepů. Jejich počet je určen uživatelem, kterému je dovoleno je přidávat i odstraňovat. Index může být replikován na žádný a nebo více replik střepů. Když je replikován, každý index získá primární střep (to jsou originální střepy, které byly replikovány) a repliky střepu (čili kopie primárního střepu). Defaultně je pro každý index alokováno 5 primárních střepů a 1 replika střepu. Tím pádem pokud jsou alespoň dva uzly v klustru, pak má index 5 primárních štrepů a dalších 5 replik střepů (1 kompletní replika). V součtu obsahuje 10 střepů pro jeden index. [40]

### Typ dokumentu

V indexu je možné definovat jeden i více typů. Typ sdružuje dokumenty se stejným významem, tím pádem stejnou nebo velmi podobnou strukturou ve stejném indexu. Každý typ obsahuje mapování, které definuje analyzování a indexování položky dokumentu. [40]

### Mapping (mapování)

Určuje, jaké položky bude dokument obsahovat a jak budou zpracovávány při indexaci. To může být definováno nebo vytvořeno automaticky při vložení prvního dokumentu.

### Dokument

Dokument je základní jednotkou informace, jež může být indexována. Dokument má specifikcký význam. Odkazuje ke kořenovému objektu, který je dále serializován (převod libovolně složitého objektu do jeho sériové podoby) do dokumentu formátu JSON[7]. [13]

Uvnitř indexu se může uložit tolik dokumentů, kolik si určíme. I když dokument náleží konkrétnímu indexu, musí být specifikován typem. Z toho vyplývá, že identifikátorem dokumentu je kombinace index-typ-ID. [40] [31]

[7] JavaScriptová objektová notace – JavaScriptový zápis objektu a také forma pro přenos dat ve stylu klíč-hodnota.

### 2.4.2 Srovnání Elasticsearch a relační databáze

V tabulce 2.2 je uvedeno srovnání klasické relační databáze a Elasticsearch.

Tabulka 2.2: Popis tabulky

| **Elasticsearch** | **Relační databáze** |
|---|---|
| Index | Databáze |
| Shard | Shard |
| Mapping | Tabulka |
| Pole | Pole |
| JSON Objekt | Tuple |

### 2.4.3 Základní koncepty

Elasticsearch klade důraz na snadnou použitelnost a horizontální škálovatelnost. Snadno použitelný je jak jako jedna pracující instance tak i při fungování více instancí jako jedna. Nastavené výchozí hodnoty umožňují začít používat Elasticsearch ihned po instalaci bez potřeby dalších zásahů.

#### Horizontální škálovatelnost

Horizontální škálovatelností (možnost rozšíření za účelem zvýšení efektivity) je myšleno souvislé navyšování kapacit jak z pohledu úložné kapacity, tak z pohledu rychlé odezvy na dotazy.

#### Peer-to-peer

Využitím peer-to-peer architektury (instance komunikují mezi sebou) se instance automaticky navzájem spojují s ostatními instancemi v rámci systému kvůli výměně dat a vzájemnému monitoringu.

#### Vyhledávání v reálném čase

Díky jeho distribuované povaze se odehrává vyhledávání téměř v reálném čase, ale nelze se vyhnout prodlevě v komunikaci mezi uzly. Tato prodleva je však minimální, protože od doby, kdy se dokument naindexuje, se stane vyhledávatelným běžně během 1 vteřiny. K tomu pomáhá verzování dat a udržování přehledu o stavu dat v čase.

## 2.5 RIAK

RIAK je NoSQL key/value databáze, která distribuuje data přes mnoho serverů, které jsou nazvány uzly (*nodes*). Tyto uzly dohromady utváří tzv. *klustr*. Tento klustr má kruhovou strukturu, jenž je popsána v podkapitole Struktura níže.

### Komponenty

Komponenty jsou základními jednotkou ukládání dat. Existuje pět následujících komponent. *Key*, *value*, *bucket*, *clock* a přidružená *meta data*. Programátor definuje hodnotu key. Key je unikátní a dá se chápat jako adresa bydliště, která je uložena v RIAK.

```
hashtable["key"] = "value"
```

Klíč může mít binární podobu, nebo se může jednat o řetězec. Value může mít různou podobu (soubor, řetězec, seznam a podobně). Každá value má také svoji verzi, která obsahuje čas, kdy byla nahrána na server. Různé verze konkrétní hodnoty se nazývají *sourozenci*.

To, že je klíč unikátní, neznamená, že může být použit jen jednou. Do key je možné uložit jinou hodnotu, pokud se již předchozí nepoužívá. Zároveň se i v jednom objektu může objevovat více unikátních klíčů a to díky jmenným prostorům (namespaces), které se nazývají buckety. V rámci jednoho objektu tedy můžeme takto definovat hodnoty se stejným klíčem:

```
prvni_namespace["stejny_key"] = "prvni_value"
druhy_namespace["stejny_key"] = "druha_value"
```

Každý klíč náleží bucketu, neexistuje zde nic jako globální bucket. Proto bychom měli klíč definovat spíše jako `bucket/klíč`, aby nedošlo k jejich záměně.

Od verze Riaku 2.0 můžeme i buckety organizovat do tzv. bucket types. Jedná se v podstatě o buckety bucketů. Definovat jej můžeme takto:

```
bucket_type["prvni_bucket"][stejny_key] = "prvni_value"
bucket_type["druhy_bucket"][stejny_key] = "druha_value"
```

Výhoda tohoto přístupu je, že skupiny bucketů mohou mezi sebou sdílet vlastnosti. Např takto:

```
bucket_type.props = {"search_index": "skupina_bucketů"}
```

Díky tomu je možné zejména to, že bucketům se dají přiřazovat vlastnosti hromadně. V případě jejich použití je vhodné definovat klíč jako `type/bucket/klíč`. [33]

### 2.5.1 Operace

Z důvodu, že se jedná o key/value databázi, většina interakce je nastavování nebo získávání value. Operacemi jsou: GET, PUT, POST a DELETE.
Pokud chceme zjistit hodnotu konkrétního keys bucket, použijeme následující příkaz:

```
GET /riak/bucket/key
```

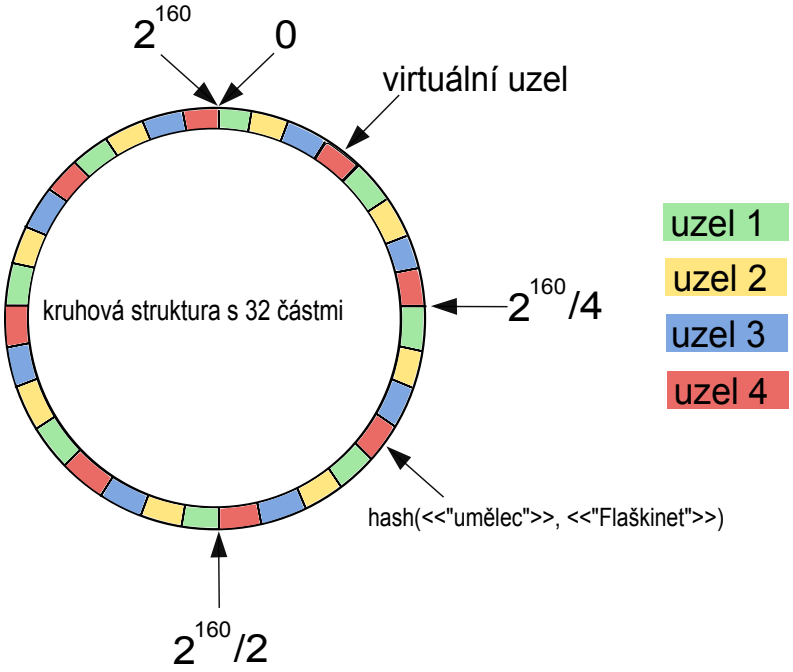
Typickým požadavkem k uložení hodnoty do bucket je následující:

```
PUT /riak/bucket/key
```

### 2.5.2 Struktura

RIAK mapuje objekt do kruhově tvarovaného klustru používáním techniky konzistentního hashování. Maximální hashovací hodnotou klustru je $2^{160}$ a ta je v kruhu rovnoměrně rozdělena do *příček*. Každý uzel v klustru se stará o virtuální uzle, které jsou zodpovědné za ukládání určitého množství klíčů do klustru. Každý virtuální uzel deklaruje „příčku“ v kruhu. Počet virtuálních uzlů je vypočítán následovně:

```
počet virtuálních uzlů = počet příček / počet uzlů
```




Obrázek 2.6: RIAK - klustr (kruh)

Počet příček je v základu 64. Příčkami se dokáže rozdělit sada key do odlišných uzlů. Pro příklad pokud se přiřadí sada 200 čísel na dva servery, RIAK to může zprostředkovat, že první polovinu uloží do uzlu A a zbytek do uzlu B. [26]
Dělení klustru do příček spolu s technikou *replikace* umožňuje vysokou dostupnost a konzistentnost dat a snižuje pravděpodobnost, že by mohla být data nedostupná. V tabulce 2.3 je ukázka srovnání relační databáze, reprezentované Oracle, a key-value databází Riak. [33]

Tabulka 2.3: Popis tabulky

| **Oracle** | **Riak** |
|---|---|
| databázová instance | riak klustr |
| tabulka | bucket |
| řádka | key-value |
| id řádky | key |

### 2.5.3 Replikace

Replikací rozumíme kopírování dat mezi několika servery. To přináší zásadní výhodu v případě výpadku některého ze serverů. V tomto případě mohou

požadavky vyřídit ostatní servery, které obsahují replikovaná data. Nevýhodou je ovšem velká paměťová náročnost.
Uzly, do kterých jsou data kopírována, se označují buď jako *primary* nebo *secondary replicas*. Uzel označovaný jako primary replica je buď jeden, nebo jich může být více, v závislosti na zvoleném typu replikace (viz dále). Právě tento uzel je zodpovědný za poskytování dat. Pokud tento uzel selže, data jsou stále dostupná na uzlech označovaných jako secondary replicas. Těch může být libovolné množství.
Zároveň rozdělujeme dva základní typy replikace podle způsobu zasílání požadavků – aktivní a pasivní replikaci. Riak využívá upravené aktivní replikace.
Při aktivní replikaci se všechny uzly chovají jako primární a také dostávají stejnou sérii požadavků od klienta. Problém nastává v případě, že uzly dostanou požadavky v jiném pořadí. Může pak dojít k porušení konzistence dat a to je také hlavní nevýhodou tohoto přístupu.
Pasivní replikace se od aktivní liší tím, že zde je za primární považován pouze jeden uzel, který vyřizuje všechny požadavky klienta. Pokud tento uzel selže, jeden ze sekundárních uzlů převezme jeho roli. Vzhledem k tomu, že jsou všechny požadavky nejprve vyřizovány pouze na primárním uzlu, nedochází zde k porušení konzistence dat. Tento přístup má ovšem také nevýhodu. Může zde dojít k situaci, kdy primární uzel selže, ale nestihne předat všechny potřebné informace sekundárnímu uzlu. Musí se zde také ošetřit případ, kdy dojde k obnovení činnosti selhaného uzlu, tedy dva uzly se současně chovají jako primární. Konzistence ve vyřizování požadavků musí zůstat zachována.[28]

# 3 Návrh

Tato kapitola se věnuje návrhu modulární monitorovací aplikace. Nejprve je popsána analýza problému zabývající se modularitou řešení. Další část navrhne samotnou strukturu aplikace. Zde je popsáno, jak jsou jednotlivé části do sebe zapojeny. Následující podkapitola se zabývá modelem instance a co bude jeho obsahem. V neposlední řadě dojde k navrhnutí REST-API. Poslední podkapitola osvětlí návrh realizace pro uživatelské rozhraní.

K aplikaci byl vytvořen Dokument specifikace požadavků. Ten se nachází jako příloha A.

## 3.1 Analýza problému

Z důvodu existujícího Dokumentu specifikace požadavků je mnoho možných témat k analýze problému vyřešeno. Jak je v dokumentu uvedeno, požadovaným programovacím jazykem je JavaScript a jeho frameworky. AngularJS pro frontend aplikace a Node.js pro backend aplikace. Už ale není uvedeno např. jakým způsobem budou monitorované instance ukládány. Nebo také jakým způsobem bude zajištěna modularita aplikace.

Pro ukládání instancí existuje hned několik variant: např. databáze na disku *h2*, databáze *MySQL* nebo databáze Elasticsearch. Všechny zmíněné úložiště je možné použít. MySQL je mezi těmito kandidáty úložištěm nejstarším a velmi populárním. I přesto ale nedosahuje nejlepších výkonnostních parametrů, které jsou zapotřebí. Naopak Elasticsearch a databáze h2 toto poskytují. Obě dvě technologie jsou napsány v programovacím jazyku *Java* a obě rovněž poskytují velice rychlé vyhledávání, až realného času. Nakonec jsem se rozhodl použít databázi Elasticsearch, protože je nejpříhodnější použít technologii, která je již používána ve stejné sadě technologií monitorované aplikace a je pro tento úkol vhodná.

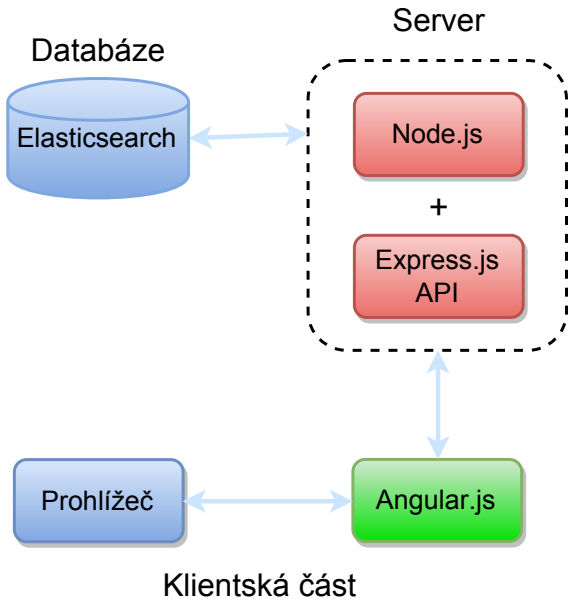
Zajištění modularity závisí na mnoha záležitostech. Prvotní je vhodná volba struktury modelu popisující monitorovanou instanci. Zde musí být umožněno snadné přidání nové komponenty. To bude moct obsahovat libovolnou strukturu. Dalším bodem je provádění operací nad instancí. Není vhodné, aby bylo toto prováděno nad jejími konkrétními komponentami. Pokud bude zapotřebí přidat novou komponentu, stačí přidat novou funk-

cionalitu pro zjišťování statusu celé instance instance a tím je zaručeno, že obměna podoby instance bude jednoduše změněna.

## 3.2 Struktura aplikace

Webová aplikace obvykle nestojí pouze na jedné konkrétní technologii, jako tomu může být u desktopových aplikací, ale je obvykle kombinovaná z několika, které danou problematikou řeší nejlépe. Jak bylo napsáno v dokumentu specifikace, hlavní platformou zodpovědnou za serverovou část aplikace je technologie Node.js. Pro jeho konfiguraci se použije aplikační framework Express.js. Ten se stará o vytvoření REST API a dokáže předkládat soubory, které jsou uloženy na datovém úložišti a na serveru. Pro úložiště monitorovaných instancí je určena dokumentově orientovaná databáze Elasticsearch.

Jako framework JavaScriptu pro vytváření uživatelského rozhraní aplikace a komunikace se serverem pomocí REST API je určen framework Angular.js.




Obrázek 3.1: Struktura aplikace

## 3.3 Model instance

Nyní je zapotřebí navrhnout model pro monitorovanou instanci. Protože se budou data o jednotlivých instancích uchovávat v databázi Elasticsearch, je zapotřebí vymyslet strukturu JSON objektu, se kterým bude zacházeno.

Protože vyhledávání záznamu v Elasticsearch je nejsnadnější pomocí unikátního ID záznamu, je třeba docílit toho, že bude mít instance stále stejné ID a svůj název. Změna ID v Elasticsearch ale není možná. První variantou je, aby se ID generovala sama a tak by zůstávala stejná a měnil by se pouze název instance. Tímto se ale ztrácí možnost získávání instance podle jejího aktuální jména, čímž se práce s modely instance komplikuje. Rozhodl jsem se pro druhou variantu, která vytváří a i aktualizuje ID instancí podle jejich jmen. Stačí, aby se při aktualizaci instance zjistilo, zda došlo ke změně jejího jména a pokud ano, tak se vytvoří instance s jiným názvem a následně smaže ta s názvem původním.

Dále bude model obsahovat boolean hodnotu, určující, zda má být instance monitorována či nikoliv. Nyní následují údaje o komponentách instance.

Protože jedna instance může obsahovat více frontendů a rovněž více backendů, bude zapotřebí tyto údaje vkládat do pole. V něm bude název a adresa. Pro složky Elasticsearch a Riak stačí hodnota typu string určující jejich adresu.

V neposlední řadě je zapotřebí určení mailových adres, kam se budou posílat maily. Tyto mailové adresy obsahují položku pro mailovou adresu a dále boolean hodnotu určující, jestli chtějí dostávat upozornění.

V příloze se nachází ukázka celého datového modelu instance. Příloha C

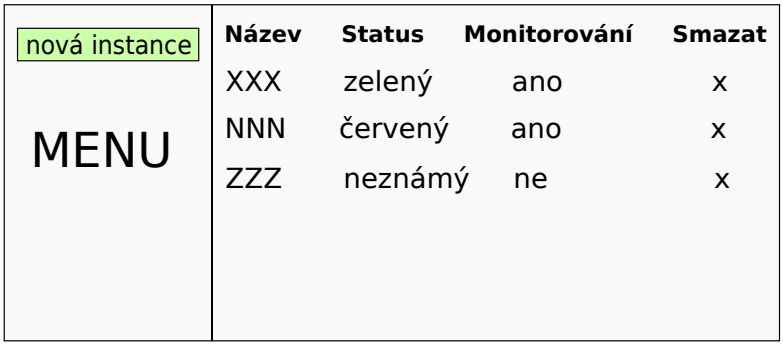
## 3.4 Aplikační rozhraní (API)

Funkcionalitou aplikačního rozhraní jsou operace nad databází, konkrétně nad modely instancí. Prvotní je nutné vytvořit základní funkce pro práci s instancemi. Těmi bude získání všech instancí, získání jedné konkrétní instance a vytvoření instance. Aby aplikace splňovala specifikaci požadavků, je nutné, aby bylo možné instanci rovněž měnit a i smazat.

## 3.5 Uživatelské rozhraní

Neméně důležitým faktorem, jako je správná funkčnost API, je rovněž přehledné a intuitivní uživatelské rozhraní. Úvodní obrazovka aplikace by měla ukázat uživateli výčet všech instancí, které monitoruje. Ty by byly zobrazeny v řádkách pod sebou a obsahovaly by i další důležité informace. Mezi ty patří, zda je instance monitorovaná, jestli má zapnuté posílání mailů a především celkový status instance. Viz obrázek 3.2

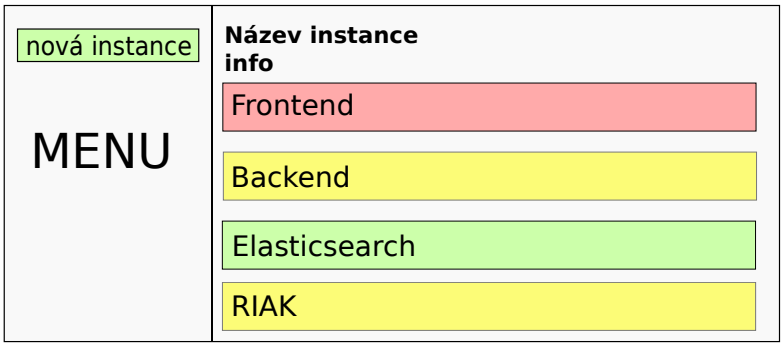
Na levé straně by se nacházel kontrolní panel, který obsahuje výčet instancí, díky kterému se uživatel rovnou dostane k dané instanci. Dále tlačítko pro zobrazení domovské stránky a nakonec tlačítko pro vytvoření nové instance. Toto menu by mělo být schopné se nechat skrýt, když by bylo pro uživatele rušivé.




Obrázek 3.2: Návrh úvodní obrazovky

Vytvoření nové instance není vhodné řešit jedinou šablonou pro vyplnění všech nutných údajů, ale spíš je výhodné rozdělit formulář do několika kroků. V každém kroku by se nastavila určitá komponenta a zároveň by byla validována konkrétní předávaná hodnota.

Pro detailní obrazovku zobrazující instanci je nutná přehlednost. Na první pohled by mělo být jasné, že má instance určitý status a v závislosti na kterých komponentách tento status má. Proto by jednotlivé komponenty měly být ohraničeny barvou značící jejich status a obsahovaly by informace o jejich stavu. Viz obrázek 3.3




Obrázek 3.3: Návrh detailní obrazovky

# 4 Implementace

## 4.1 Webový server - backend

Pro vytvoření aplikace je nutné úspěšně nakonfigurovat a spustit webový server, na kterém bude aplikace operovat.

Jak již bylo uvedeno výše, to je možné obstarat frameworkem Express.js. Ten dokáže naimportovat pomocné moduly a knihovny. Soubor, jež bude obsahovat konfigurační kód jsem nazval jednoduše `app.js`.

Na jeho počátku se postupně načítají pomocné moduly a i samotný Express.js. To vše za pomoci metody `require()`. Příkladný import vypadá následovně:

Ukázka kódu 4.1: Ukázka importů v app.js

```
var express = require('express');
var bodyParser = require('body-parser');
var cors = require('cors');
var responseTime = require('response-time');
var https = require('https');
var http = require('http');
var routes = require('./routes/index');
var nconf = require('./config.js');
```

Mezi další základními moduly k naimportování se použijí následující:

- *cors* – modul, díky němuž má server možnost kontrolovat, jací klienti se připojí a jakým způsobem se připojí.
- *body parser* – modul sloužící k extrahování příchozích HTTP datových požadavků. Jedná se například o data zaslaná pomocí metody POST, vytvořená po odeslání formuláře (např. vytvoření monitorované instance). Tyto data jsou poté přístupná v `req.body`.
- *http, https* – moduly zajišťující vytvoření serveru na protokolu HTTP a nebo na verzi HTTPS.
- *responseTime* – modul vytvářející middleware, který zapisuje čas odpovědi pro HTTP žádosti serveru. „Čas odpovědi“ je definován jako

uběhnutý čas od doby, kdy žádost dorazí k tomuto middlewaru do doby, kdy se vypíše odpověď klientovi.

Do proměnných se naimportují i potřebné soubory.

- *routes* – představuje soubor, který vytváří *router* jako modul pro *směrování* (angl. routing). Samotný proces směrování je popsán v kapitole o tvorbě API.
- *nconf* – představuje modul, díky němuž je možné snadno zacházet s konfiguračním souborem pro celou aplikaci.

Po importování modulů a souborů je nutné vytvořit odkaz na aplikaci Express.js následujícím příkazem:

Ukázka kódu 4.2: Vytvoření aplikace

```
var app = express();
```

Dále se nakonfigurují načtené knihovny. Pro to se použije slovo *use*.

Ukázka kódu 4.3: Konfigurace modulů a souborů

```
app.use(bodyParser.json({limit: '50mb'})); // Parsovac
app.use('/api/v1', routes);        // Pridani routovani
```

Po nakonfigurování knihoven se vytvoří aplikační rozhraní (API). Posledním krokem je samotné spuštění webového serveru.

Ukázka kódu 4.4: Spuštění serveru na HTTP portu

```
var httpPort = nconf.get('server:http_port');
   http.createServer(app).listen(httpPort, function () {
       console.log("Server is started for http on port " +
          httpPort);
   });
```

## 4.2 Databáze - elasticsearch

Jednotlivé monitorované instance jsou uloženy a spravovány v databázi Elasticsearch. Pro komunikaci aplikace nad touto databází je vhodné použít modul podobným způsobem, který je obsažen při vytvoření webového serveru. K vytvoření klienta modulu `elasticsearch` je zapotřebí použít jeho konstruktor. V tom je možné nadefinovat libovolné množství výchozích hodnot. Zde je uveden výtažek z kódu aplikace, kde se nastaví hostitel, ke kterému se klient připojí a rovněž verze rozhraní, specifikující verzi elasticsearch.

Ukázka kódu 4.5: Vytvoření klienta pro elasticsearch

```
var elasticsearch = require('elasticsearch');
var instances = require('./elasticsearch/instances');
exports.instances = instances;

var client = new elasticsearch.Client({
    host: nconf.get('server:es:host'),
    apiVersion: nconf.get('server:es:apiVersion')
});
```

### 4.2.1 Práce s elasticsearch

Po úspěšném vytvoření klienta je nyní možné používat funkce, jež jsou nutné pro práci s instancemi uloženými v databázi. Opětovně jsou zde zastoupeny základní funkce pro správu databáze, funkce *CRUD*.

- **client.get** – získá JSON model z indexu v závislosti na dodaném id.
- **client.search** – získá dokumenty, které se shodují v požadovaných kritériích.
- **client.update** – aktualizuje část nebo celý dokument, který je určen svým id.
- **client.create** – vytvoří JSON model v určitém indexu. Pokud již databáze obsahuje model se stejným id, tak bude vyhozena výjimka.
- **client.delete** – vymaže JSON model v určitém indexu určeném pomocí id.
- **client.count** – získá počet dokumentů, které se shodují v požadovaných kritérií.

Příkladem bude opětovně uvedena funkce pro získání konkrétní instance.

Ukázka kódu 4.6: Získání konkrétní instance

```
function getInstance(id) {
    return client.get({
        index: instance_index,
        type: instance_type,
        id: id
    });
}
```

## 4.3 Tvorba REST-API

### 4.3.1 Routování v Express.js

Routování je způsob, jakým aplikace odpoví na požadavek uživatele na konkrétní koncový bod (viz. 2.1.4). Ten je definován řetězcem *URI* a zaslán specifickou HTTP metodou. Funkce na zpracování požadavku (angl. handlers) jsou spuštěny ve chvíli, když se URI shoduje s cestou v dané „routě“. Každá „routa“ může mít jednu či více zpracovávajících funkcí.
V Express.js se routa definuje jako *app.METHOD (PATH, HANDLER)*, kde *app* je instance Express.js, *METHOD* je typ HHTP požadavku, *PATH* je cesta na serveru, *HANDLER* je funkce, která je spuštěna, když se cesta v routě shoduje.[17]

Ukázka kódu 4.7: Zaregistrování trasy

```
router.get('/instances', process(instances.getInstance));
```

Ukázka kódu 4.8: Trasa obsahující swagger a její validaci

```
var es = require('../services/elasticsearch');

var resources = {};
resources.getInstance = {
    swagger: InstancesValidations.swagger.getInstance,
    validation: {
        path: InstancesValidations.joi.getInstance.path
    },
    endpoint: getInstance
};
```

Zde je příklad trasy, která podle předaného parametru *id* vyhledá odpovídající monitorovanou instanci a zašle ji zpět klientovi.

Ukázka kódu 4.9: Funkce routy

```
function getInstance(req, res) {
    var id = req.params.id;

    es.instances.getInstance(id).then(function (result) {
        var instance = result._source;
        res.send(instance);
    }).catch(function (err) {
        restUtils.sendError(err, res);
    });
```

### 4.3.2 Swagger a validace dat

Pro přehlednost toho, co REST-API dokáže a rovněž pro přehledné dokumentování jeho funkcí je použit zmíněný modul Swagger. Ukázka tohoto modulu je předvedena v příloze: E.4.

Pro robustní REST-API je rovněž důležité, aby byla ověřována správnost dat, která se budou předávat. K tomuto účelu byla použit modul Joi.

Ukázka kódu 4.10: Validace názvu instance

```
InstanceRequest.joi = {};
InstanceRequest.joi.schema = Joi.object().keys({
    name: Joi.string().trim().empty('').min(1).required(),

});
```

### 4.3.3 Specifikace REST-API

V aplikaci existuje celkem osm koncových bodů, jak je naznačeno v příloze D. Většina se používá k manipulaci s instancemi.
`GET /instances` je bez parametru a vrací všechny instance.
`GET /instances/count` je rovněž bez parametru a odpovědí je počet všech instancí.
`GET /instances/:id` požaduje jako parametr číslo id a vrací jednu instanci s daným id.
`GET /instances/status/:id` má rovněž jako jediný parametr id instance, ale zde vrací její zdravotní stav.
`POST /instances` se využívá k aktualizaci informací a v odpovědi obsahuje aktuální informace o instanci.
`PUT /instances/:id` vytváří novou instanci s id, které se zadává jako parametr. Pokud je parametrem id, které již existuje, instance se nevytvoří.
`DELETE /instances/:id` smaže instanci, jejíž id má jako parametr.
`POST /email` nemá žádný parametr a slouží k zaslání emailu.

## 4.4 Určování stavu komponent

Stav jednotlivých instancí je určen stavem jejich komponent. Následující výčet komponent osvětluje metriku, která byla použita pro určení jejich stavu.

- *Frontend aplikace* – určení jeho stavu je ověření, zda v daný moment frontend funguje. Posláním pingu na adresu se zjistí jeho odezva. Po-

kud se tato žádost úspěšně vyplní, bude mít frontend stav zelený. V opačném případě bude stav nastaven jako červený.

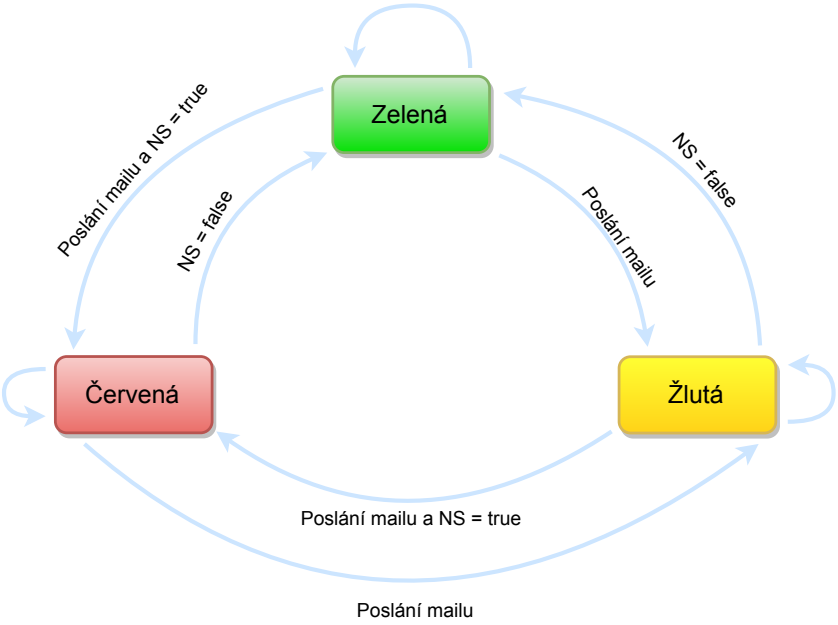
- *Backend aplikace* – je určen podobným způsobem, jako je uvedeno výše. Posláním žádosti na danou adresu a port, na kterém backend běží, je nastaven stav v úspěšném případě jako zelený, v jakémkoliv jiném případě na červený.
- *Elasticsearch* – již obsahuje svůj stav v odpovědi po vykonané žádosti na jeho adresu. Elasticsearch podporuje stejnou metriku pro určování stavu instance (zelený, žlutý a červený). Díky tomu je této komponentě rovnou nastavena tato hodnota.
- *RIAK* – u této komponenty dojde na základě její odpovědi k ověření, zda funguje. Na tomto základě je nastavena hodnota zeleného a nebo červeného stavu.

## 4.5 Automatické posílání mailů

Pro zajištění periodického běhu monitorování instancí je příhodné použít výše popsaný modul node-schedule. Při každém spuštění aplikace bude zavoláno vytvoření plánovače. Plánovač je nastaven tak, aby funkce pro kontrolu instance proběhla každou minutu.

Samotná logika posílání mailů je založena na hodnotě původního stavu instance, aktuálním stavu instance a hodnotě typu boolean značící zaslání mailu. Na základě těchto informací se buďto zašle mail oznamující změnu stavu a případně změna hodnoty značící zaslání mailu. Logika posílání mailů je přehledně ukázána na obrázku 4.1, kde NS značí zmiňovanou booleanskou hodnotu.

Veškerý obsluhovací kód pro automatické posílání mailů je uložen v souboru `statusChecker.js`.




Obrázek 4.1: Kontrola pro posílání mailů

## 4.6 Klientská část - frontend

Jedná se o část, jež komunikuje s webovým serverem a zároveň uživatelem, pracujícím s aplikací. Tato část je založena především na frameworku Angular.js spolu s HTML šablonami, jejichž vzhled je kombinací vlastních `CSS`[1] stylů a frameworku `Bootstrap`.[2]

Nejdůležitější součásti adresáře určeného pro frontend jsou složky app, styles a soubory index.html, bower.json a package.json. Poslední dva zmiňované slouží pro management závislostí. Bower.json je určen pro frontendové soubory a package.json pro Node.js soubory. Index.html slouží jako základní šablona a v souboru styles jsou uloženy CSS styly. Logika frontendu je obsažena ve složce `app`.

### 4.6.1 Adresář app

Nejpodstatnějšími soubory v tomto adresáři jsou:

- *layout* – adresář, který obstarává rozvržení bloků aplikace.

[1] CSS neboli Cascading Style Sheets je jazyk popisující styl zobrazení elementů daných značkovacím jazykem (většinou HTML).

[2] Bootstrap je frontendový framework pro snadnou tvorbu vzhledu aplikace.

- *components/home* – adresář, který obsahuje kontroléry, šablony a direktivy spolu se službami, které se starají o hlavní obsah klientské části.
- *filters* – adresář obsahující vlastní filtry. Obsahuje filtry, které jsou obecnější a můžou být použity napříč aplikací.
- *services* – adresář obsahuje servisy, které mohou nalézt použití na různých místech aplikace.
- *app.js* – soubor připravující a spouštějící frontend aplikace.
- *includes.js* – určuje soubory, které budou načteny při spuštění aplikace.

Nyní bude popsána implementace tří základních konceptů, které byly nastíněny v sekci o návrhu uživatelského rozhraní.

### 4.6.2 Úvodní obrazovka

Úvodní obrazovka aplikace si dává za cíl přehledně zobrazit všechny vytvořené monitorované instance a ukázat jejich název, stav, zda jsou monitorovány a také nabízí možnost jejich odstranění.

#### Kontroler – category.js

Každý kontroler této aplikace obsahuje funkci `init()`. Tato funkce je zavolána vždy, kdy je uživatelem zobrazena stránka, která přísluší danému kontroleru. Akce, které se pro úvodní obrazovku vykonají jsou následující:

- Zisk všech instancí pomocí RESTového rozhraní a uložení do pole. Z tohoto pole budou zjišťovány jednotlivé stavy instancí bude po dobu zobrazení každých 30 sekund aktualizováno.
- Obsahuje také funkci obstarávající vymazání instance. Předtím, než ale tak učiní, vyskočí na obrazovku okénko, aby uživatel mohl potvrdit definitivní smazání instance.

### 4.6.3 Vytvoření instance

Pro vytvoření nové monitorovací instance byl kladen důraz na jednoduchost a přehlednost. Formulář bude rozdělen do několika po sobě jdoucích kroků, během kterých budou hodnoty validovány. Jakmile budou vyplněny všechna povinné hodnoty, uživatel bude vyzván, aby potvrdil, či vyvrátil vytvoření instance.

#### Kontroler – newInstance.js

Při inicializaci stránky pro vytvoření nové instance dochází pouze k nastavení výchozích hodnot pro zapnutí monitorování instance a pro nastavení, že ještě nebyla odeslána notifikace související s odesíláním mailů. Jediná funkce kontroleru, která bude spuštěna při potvrzení uživatele, je zaslání požadavku pro vytvoření instance z dat, které se získali.

### 4.6.4 Detail instance

Tato sekce klientské části obsahuje jak nejvíce prvků kódu HTML, tak rovněž nejvíce užité funkcionality. Opět je použita dvojice kontroler a její HTML šablony, avšak v kódu souboru `detail.html` se nachází nově definované znaky, kterými jsou direktivy. Tyto direktivy jsou rozděleny do složek o stejném názvu, obsahující soubor JavaScriptu a HTML soubor. V JavaScriptovém souboru se nachází jak vytvoření direktivy, tak i vytvoření příslušného kontroleru, jenž je využíván.

- `circularoPanel` – direktiva pro vytvoření panelu zobrazující frontendy a backendy aplikace.
- `elasticPanel` – direktiva pro vytvoření panelu zobrazující úložiště Elasticsearch. Obsahuje rovněž direktivu pro modální okénko zobrazující bližší informace o komponentě.
- `riakPanel` – direktiva pro vytvoření panelu zobrazující úložiště RIAK. Podobně jako elasticPanel, i zde je obsažena direktiva pro vytvoření modálního okénka pro bližší informaci.
- `elasticPanel` – direktiva pro vytvoření panelu zobrazující všechny maily, na které se mají zasílat mailová upozornění.

#### Kontroler – detail.js

Když je inicializována stránka zobrazující detail pro jednotlivou instanci, je spuštěna funkce pro zjištění stavu konkrétní instance. Dále se již při spuštění této stránky samo nic neprovede. Nyní následuje výčet funkcí obsažených v kontroleru.

- *deleteInstance* – kompletní vymazání instance.
- *saveInstance* – uložení změn instance a zobrazení výsledku monitorování s novými daty

- *addFrontendRow / removeFrontend* – přidání další položky pro frontend instance / odebrání vybrané položky.
- *addBackendRow / removeBackend* – přidání další položky pro backend instance / odebrání vybrané položky.
- *addMailRow / removeMail* – přidání dalšího mailu pro posílání notifikací / odebrání vybrané mailové adresy. aplikace.

Rovněž obsahuje funkce pro kontrolu vyplněných dat a jejich validitu.

# 5 Testování

## 5.1 Testování funkčnosti

Pokrytí aplikace testy bylo soustředěno na splnění sekce popisující testování v dokumentu specifikace požadavků. Pro testování byly použity frameworky `Mocha` a `Chai`.

### 5.1.1 Vytvoření testů

Pro testování je nutné nejdřív vytvořit testovací instance, na kterých budou jednotlivé koncové body testovány. Před započetím každého testu je nutné vytvořit takovou instanci. V případě jakékoliv chyby musí dojít k odstranění testové instance, aby nezůstali po proběhnutém testu instance určené k testování. Díky tomu je splněna podmínka testování, která zaručuje nezávislost testů.

## 5.2 Testování použitelnosti

Testování použitelnosti je testování uživateli podle daného scénáře, při kterém je důležitá především jejich spokojenost s používáním aplikace. To znamená, že by aplikace měla být intuitivní a pohodlně použitelná. Zároveň budu průběh tohoto testování pozorovat a budu si psát poznámky o způsobu, jakým byly jednotlivé úkoly plněny.
Scénář byl vytvářen jak s ohledem na co největší pokrytí funkčnosti aplikace, tak s vědomím, že by měl odrážet skutečné potřeby klienta.

### 5.2.1 Scénář

1. úspěšné vytvoření instance monitorované aplikace s předem danými pouze povinnými parametry
2. zobrazení detailu právě vytvořené instance
3. změna jejího názvu, přidání frontendu a backendu, změna adresy Elasticsearch
4. změna monitorování instance na false
5. zobrazení úvodní obrazovky – instance by již neměla být monitorována

### 5.2.2 Výsledek testování použitelnosti

Testování proběhlo na množině pěti uživatelů s rozdílnými zkušenostmi.

1. Vzhledem k tomu, že je tlačítko pro vytvoření instance velmi dobře viditelné, uživatelé neměli problém se dostat na daný formulář. Jelikož je formulář řešen po krocích postupným vyplňováním hodnot, nebyli uživatelé přehlceni množstvím informací. Jen ve dvou případech si uživatelé mysleli, že jsou všechna pole povinná a vyplnili je také.

2. S tímto krokem neměli uživatelé problém. Většina z nich, čtyři, si novou instanci zobrazili přes hlavní nabídku monitorovaných instancí a jeden uživatel si ji zobrazil přes postaní panel.

3. Tento krok byl zpočátku pro uživatele obtížný, jelikož se snažili instanci přejmenovat pomocí tlačítka s tužkou, které slouží jinému účelu. Po několika vteřinách se však uživatelé seznámili s možností editování řetězců jednoduchým poklikáním na ně. Poté název instance bez problému změnili. Přidání frontendu a backendu již bylo velmi rychlé díky intuitivnímu tlačítku se znakem plus a právě nabyté zkušenosti s editováním řetězců. Stejně tak nedělala problém změna adresy Elasticsearch.

4. Po zběžné pohledu na obrazovku a zjištění, že se zde přímo nenachází možnost změny monitorování na false, uživatelé využili možnosti další úprav přes tlačítko s tužkou. Zde již bylo možné pomocí jim známého způsobu tuto hodnotu změnit.

5. Tento krok všichni uživatelé zvládli. Většina intuitivně poklepáním na logo Monitoring, až na jednoho méně zkušeného uživatele, který volil cestu přes overview v postranním panelu.

### 5.2.3 Otestování zákazníka

Práce byla předána zadavateli. Aplikace byla ozkoušena na fungující instanci aplikace Circularo. Všechny požadované funkce podle specifikace požadavků fungovaly podle očekávání.

### 5.2.4 Předávací protokol

Na základě předání monitorovací aplikace byl sepsán předávací protokol, který se nachází v příloze B.

# 6 Závěr

Cílem této práce bylo vytvoření funkční monitorovací aplikace dle dokumentu specifikace požadavků.

V rámci této bakalářské práce bylo popsáno několik technologií, které jsou potřeba k vytvoření funkční a komplexní SPA. je založeno na mnoha odlišných technologiích, které je zapotřebí pochopit a vhodně propojit. Ty jsou nedílnou součástí pro vytvoření SPA určené pro monitorování běhu aplikací skládajících ze stejných technologií. V rámci práce byl rovněž vytvořen dokument specifikace požadavků, podle kterého je aplikace vytvořena.

Aplikace je modulární a dále rozšiřitelná. Další práce na projektu by zahrnovala důkladnější monitorování instancí, které by obsahovalo zejména výkonnostní zatížení všech komponent a vylepšení uživatelského rozhraní.

Po důkladném otestování byla aplikace předána zákazníkovi, který podepsal předávací protokol. Tímto bylo ověřeno, že aplikace splňuje zadání a je použitelná v běžném provozu.

# Seznam zkratek

**AJAX** Asynchronous JavaScript and XML – soubor technik používaných k tvorbě javascriptových aplikací

**API** Application Programming Interface – Rozhraní pro programování aplikací

**BDD** Behavior-Driven Development – Chování řízené testy

**CORS** Cross-origin resource sharing – mechanismus umožňující načíst požadované zdroje webové stránky mimo doménu požadavku

**CSS** Cascading Style Sheets – jazyk popisující styl zobrazení elementů daných značkovacím jazykem

**CRUD** Create, Read, Update, Delete – základní databázové funkce nad záznamem v trvalém úložišti

**DOM** Document Object Model – rozhraní, které reprezentuje HTML, XHTML a XML dokumenty jako stromovou strukturu

**HTML** HyperText Markup Language – Značkovací jazyk pro tvorbu webových stránek

**JSON** JavaScript Object Notation – JavaScriptová objektová notace

**JSONP** JavaScript Object Notation with Padding – způsob získávání dat ze serveru, který je na jiné doméně než klient

**MVC** Model–View–Controller – architektura uživatelských rozhraní, převážně používaná webovými aplikacemi

**NPM** Node Package Manager – správce balíků pro Node.js

**REST** Representational state transfer – způsob poskytování interoperability mezi systémy na internetu

**SPA** Single-page application – webová aplikace fungující pouze v rámci jedné webové stránky

**TDD** Test-Driven Development – Programování řízené testy

**UUID** Universally Unique IDentifier – 128-bitové číslo, používané pro jednoznačnou identifikaci objektu

**URI** Uniform Resource Identifier – řetězec znaků identifikující zdroj

# Literatura


[1] *AngularJS AJAX* [online]. [cit. 2017/03/05]. Dostupné z: https://www.w3schools.com/angular/angular_http.asp.

[2] *AngularJS Data Binding* [online]. . [cit. 2017/03/05]. Dostupné z: https://docs.angularjs.org/guide/databinding.

[3] *To SPA or not to SPA* [online]. . [cit. 2017/03/05]. Dostupné z: http://blogs.msmvps.com/theproblemsolver/2013/10/23/to-spa-or-not-to-spa/.

[4] *Multi page web applications vs. single page web applications* [online]. . [cit. 2017/03/05]. Dostupné z: http://www.eikospartners.com/blog/multi-page-web-applications-vs.-single-page-web-applications.

[5] *AngularJS Module* [online]. . [cit. 2017/03/05]. Dostupné z: https://docs.angularjs.org/guide/module.

[6] *Single Page Application: A Practical Website Building Choice* [online]. . [cit. 2017/03/05]. Dostupné z: http://www.vocso.com/blog/single-page-application-a-practical-website-building-choice/.

[7] *AngularJS Reasons* [online]. . [cit. 2017/03/05]. Dostupné z: http://www.business2community.com/tech-gadgets/angularjs-web-apps-now-0932000.

[8] *AngularJS-Scope* [online]. . [cit. 2017/03/05]. Dostupné z: http://www.c-sharpcorner.com/UploadFile/2776f9/angularjs-scopes/.

[9] *Why do we need a single page application* [online]. . [cit. 2017/03/05]. Dostupné z: https://stackoverflow.com/questions/16642259/why-do-we-need-a-single-page-application.

[10] *AngularJS prvky* [online]. . [cit. 2017/03/05]. Dostupné z: https://docs.angularjs.org/guide/concepts.

[11] *AngularJS* [online]. . [cit. 2017/03/05]. Dostupné z: https://docs.angularjs.org/guide/introduction.

[12] *Assertion library Chai* [online]. [cit. 2017/03/05]. Dostupné z: http://chaijs.com/.

[13] *Elasticsearch document* [online]. [cit. 2017/03/05]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/guide/current/document.html#document.

[14] *Elasticsearch cluster workflow* [online]. [cit. 2017/03/05]. Dostupné z: https://help.kcura.com/9.3/Content/Relativity/Data_Grid/Relativity_Data_Grid.htm.

[15] *mvc* [online]. . [cit. 2017/03/05]. Dostupné z: https://www.upwork.com/hiring/development/express-js-a-server-side-javascript-framework/.

[16] *Express.js guide using-middleware* [online]. . [cit. 2017/03/05]. Dostupné z: http://expressjs.com/uz/guide/using-middleware.html.

[17] *Express.js guide basic-routing* [online]. . [cit. 2017/03/05]. Dostupné z: http://expressjs.com/en/starter/basic-routing.html.

[18] *CORS and JSONP* [online]. [cit. 2017/03/05]. Dostupné z: https://dev.socrata.com/docs/cors-and-jsonp.html.

[19] *Object schema validation* [online]. [cit. 2017/03/05]. Dostupné z: https://dzone.com/articles/hapijs-in-action-introducing-joi.

[20] *Utility library Lodash* [online]. [cit. 2017/19/04]. Dostupné z: https://lodash.com/.

[21] *JavaScript test framework Mocha* [online]. [cit. 2017/03/05]. Dostupné z: https://mochajs.org/.

[22] *Node Package Manager* [online]. [cit. 2017/03/05]. Dostupné z: https://docs.npmjs.com/getting-started/what-is-npm.

[23] *Node.js event-loop* [online]. . [cit. 2017/03/05]. Dostupné z: http://abdelraoof.com/blog/2015/10/28/understanding-nodejs-event-loop.

[24] *Modul node-schedule* [online]. . [cit. 2017/03/05]. Dostupné z: https://github.com/node-schedule/node-schedule.

[25] *The Difference Between TDD and BDD* [online]. . [cit. 2017/03/05]. Dostupné z: http://joshldavis.com/2013/05/27/difference-between-tdd-and-bdd/.

[26] *RIAK Clusters* [online]. [cit. 2017/03/05]. Dostupné z: http://docs.basho.com/riak/kv/2.2.0/learn/concepts/clusters/.

[27] *JavaScript file and module loader RequireJS* [online]. [cit. 2017/19/04]. Dostupné z: http://requirejs.org/.

[28] *Active and Passive Replication* [online]. [cit. 2017/03/05]. Dostupné z: https://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems/.

[29] *Swagger, What is Swagger* [online]. [cit. 2017/03/05]. Dostupné z: http://swagger.io/docs/specification/what-is-swagger/.

[30] *mvc* [online]. [cit. 2017/03/05]. Dostupné z: http://www.artima.com/articles/dci_vision.html.

[31] *elasticseasrch* [online]. [cit. 2017/03/05]. Dostupné z: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html.

[32] Clinton Gormley, Z. T. *Elasticsearch: The Definitive Guide*. O'Reilly Media, 2015. ISBN 978-1-4493-5849-5.

[33] Eric Redmond, J. D. *A Little Riak Book*. Basho, 2014.

[34] Freeman, A. *Pro AngularJS*. Apress, 2014. ISBN 978-1-4302-6449-1.

[35] Lerner, A. *Ng-book: the complete book on AngularJS*. Fullstack io; 1 edition (December 29, 2013), 2013. ISBN 978-0991344604.

[36] Mardan, A. *Express.js Guid: The Comprehensive Book on Express.js*. CreateSpace Independent Publishing Platform, 2014. ISBN 1494269279.

[37] Mardan, A. *Practical Node.js*. Apress, 2014. ISBN 978-1-4302-6596-2.

[38] Mclaughlin, B. *What Is Node?* O'Reilly Media, 2011. ISBN 978-1-4493-1005-9.

[39] Mikowski, M. S. – Powell, J. C. *Single Page Web Applications*. Manning, 2013. ISBN 978-1-6172-9075-6.

[40] Tutorialspoint, S. e. l. *Elasticsearch*. Tutorials point, 2016.

# A Dokument specifikace požadavků

## Modular Monitoring App

pro Palaxo Development s.r.o.

Verze **1.6**

Historie dokumentu

| **Datum** | **Verze** | **Popis** | **Autor** |
|---|---|---|---|
| 19. 10. 2016 | 1.0 | První verze | Jakub Šmaus |
| 27. 10. 2016 | 1.1 | Druhá verze | Jakub Šmaus |
| 22. 11. 2016 | 1.2 | Třetí verze | Jakub Šmaus |
| 29. 11. 2016 | 1.3 | Čtvrtá verze | Jakub Šmaus |
| 10. 12. 2016 | 1.4 | Pátá verze | Jakub Šmaus |
| 20. 2. 2017 | 1.5 | Šestá verze | Jakub Šmaus |
| 20. 4. 2017 | 1.6 | Sedmá verze | Jakub Šmaus |

# Úvod

## Předmět specifikace

Předmětem specifikace je modulární monitorovací aplikace pro firmu `PALAXO`. Tato webová aplikace bude monitorovat činnost produktu pro správu management dat výše zmiňované firmy, jenž nese název `Circularo`. Firma chce dosáhnout toho, že bude schopna při zavádění a provozu jednotlivých instancí Circularo jednoduše zjišťovat co možná nejrychleji jejich funkčnost a stav. Produkt Circularo je založen na následujících technologiích:
**AngularJS** - front-end.
**Node.js** - back-end.
**Elasticsearch** - NoSQL dokumentově orientovaná databáze.
**RIAK** - NoSQL key/value databáze.

Číslo verze specifikace - **1.6**.

## Cílové publikum, návod ke čtení

Specifikace je určena především vývojářům firmy PALAXO a vedoucímu práce.

## Rozsah projektu

Vytvoření webové modulární (možnost přidání další monitorovací komponenty) aplikace, která bude monitorovat běžící instance cloudových instancí Circularo, stav Elasticsearch (dále jako ES) klustrů, NGINX, RIAK a NodeJS servery. Aplikace bude periodicky zjišťovat stav všech komponent, poskytovat souhrnný náhled na jejich stav a informovat vývojáře mailem. Cílem je zajistit rychlejší možnost nalezení/opravení případných chyb v určitých komponentách a zajištění přehledu, jak jsou komponenty v daný moment funkční.

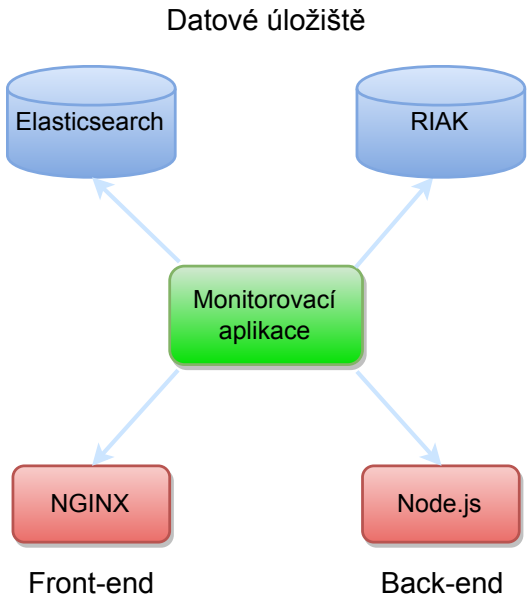
# Odkazy

**Monitoring app requirements**. [online]. 19.12.2016 [cit. 2016-03-21]. Dostupné z dokumentů Google Docs členům PALAXO teamu.

# Obecný popis

## Kontext systému

Jedná se o modulární monitorovací aplikaci, naprogramovanou v technologiích založených na programovacím jazyce JavaScript. AngularJS pro front-end (dále jako FE) a Node.js pro back-end (dále jako BE) aplikace. Jako datové úložiště pro monitorované instance je vybrána databáze ES. Aplikace bude sloužit ke komplexnímu monitorování aplikací firmy PALAXO. Kontextový diagram aplikace vypadá následovně.




Obrázek A.1: Kontext programu

## Třídy uživatelů

Naprogramovanou aplikaci budou používat vývojáři firmy PALAXO, kteří se budou i podílet dohledem na jejím vývoji. Aplikace je určena pro vývojáře firmy PALAXO.

# Provozní prostředí

Aplikace bude podporovat operační systémy Windows (verze 7 a výše) a Debian GNU/Linux. Je zapotřebí následující minimální hardware - počítač s CPU se 2 jádry, 8 GB RAM. Z hlediska software je zapotřebí zajištění běhového prostředí serveru využitím Node.js. Aplikace bude vyvíjena ve vývojovém prostředí IntelliJ IDEA, které je používané vývojáři firmy. Aplikaci bude možné spustit jak z tohoto vývojového prostředí, tak z příkazové řádky příkazem: `node app.js`.

# Omezení návrhu a implementace

Požadovaný programovací jazyk je JavaScript a jeho frameworky: AngularJS a Node.js.

# Funkce systému

## Vytvoření/smazání instance

### Popis a priorita

Ve webové aplikaci bude možné zakládat instance Circularo. Tyto instance bude možné rovněž editovat a mazat. Jejich monitorování bude možné kdykoliv pozastavit a zapnout.
Priorita: vysoká.

## Monitorování komponent instance

### Popis a priorita

Ve webové aplikaci bude možné přidat jednotlivé komponenty monitorované instance. Pro přidání jednotlivé části je nutné znát její adresu.
Priorita: vysoká.

### Funkční požadavky

**Monitorování - ES**: Monitoring komponenty pro NoSQL dokumentově orientovanou databázi ES.

**Monitorování - RIAK**: Monitoring komponenty NoSQL key/value RIAK.

**Monitorování - front-end serveru - NGINX**: Monitoring komponenty serveru NGINX.

**Monitorování - back-end serveru - Node.js**: Monitoring komponenty serveru Node.js.

## Konfigurace monitorování

### Nastavení adres

Ve webové aplikaci bude možné zadávat specifické adresy pro ES a RIAK.

### Nastavení Front-end a Back-end

Ve webové aplikaci bude možné zadávat FE a BE dané instanci. Dále bude možné měnit její adresy pro FE a BE a taktéž odstraňovat.

## Automatické posílání e-mailů

U každé instance bude možné zapnutí/vypnutí automatického zasílání e-mailů. Tyto e-maily budou posílány vývojářům firmy PALAXO a budou zobrazovat stav instance.

# Požadavky na vnější rozhraní

## Uživatelské rozhraní

Pro zobrazení aplikace se bude používat internetový prohlížeč. Aplikace bude schopna být zobrazena na běžně dostupných prohlížečích: Google Chrome, Mozilla Firefox, Internet Explorer a Microsoft Edge. Budou podporovány minimálně dvě nejnovější verze těchto prohlížečů.

## Komunikační rozhraní

Mezi FE a BE je použito rozhraní REST API, které komunikuje pomocí HTTP protokolu. Maily budou posílány pomocí SMTP serveru. Komunikace mezi BE a monitorovanými aplikacemi bude zajištěna opět přes REST API.

# Další parametrické (mimofunkční) požadavky

## Kvalitativní parametry

**Snadnost používání**: Důsledné a řádné komentování kódu pro snadnou čitelnost a srozumitelnost. Štábní kultura komentování bude vycházet dle zvyklostí firmy PALAXO.

**Robustnost**: Aplikace bude obsahovat vlastní testy pro zvýšení robustnosti.

**Udržovatelnost**: Dodržování konvence při tvorbě standardní webové aplikace, v praxi používané označování názvů souborů a metod. Štábní kultura pro označování souborů a strukturu aplikace bude vycházet ze zvyklostí firmy PALAXO.

## Otestování aplikace

### Popis

Aplikace bude obsahovat testy. Jejich rozsah bude určen během vytváření aplikace. Důležitost je kladena na testy pokrývající BE aplikace. Budou testovány scénáře, které budou ověřovat základní funkčnost aplikace. Např. testování přidání nové instance Circularo, nebo pozměnění obsahu instance. Pro psaní testů budou použity frameworky Mocha, Chai.

# Ostatní požadavky

Kód aplikace a komentáře v kódu musí být lokalizovány v angličtině.

Zadavatel: Ing. Josef Neumann, Ph.D.

Podpis: .....

# B Předávací protokol

Prohlášení zadavatele:

Potvrzuji, že jsem převzal funkční a kvalitní softwarový produkt včetně zdrojových kódů a dalších součástí konzistentní konfigurace, příslušné dokumentace a instalačního balíčku, a to vše dle předem smluvených požadavků. Se softwarovou aplikací, její instalací a provozem jsem byl detailně seznámen.

V Plzni dne 2. 5. 2017

.....

Ing. Josef Neumann, Ph.D.

Prohlášení autora:

Autor předal zadavateli jím vytvořený sw produkt se všemi náležitostmi viz výše v dohodnutém rozsahu, kvalitě a termínu.

V Plzni dne 2. 5. 2017

.....

Jakub Šmaus

# C Model instance

Ukázka kódu C.1: JSON model instance

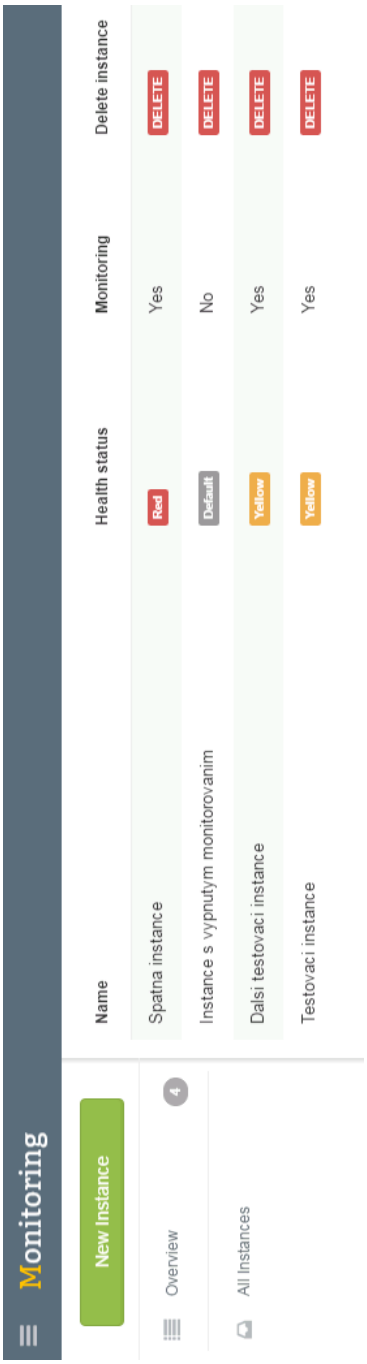
```json
{
  "instance": {
    "_meta": {
      "version": "1.1.1"
    },
    "dynamic": "strict",
    "properties": {
      "name": {"type": "string" },
      "monitored": {"type": "boolean" },
      "frontend": {
        "properties": {
          "name": {"type": "string" },
          "address": {"type": "string" }
        }
      },
      "backend": {
        "properties": {
          "name": {"type": "string" },
          "address": {"type": "string" }
        }
      },
      "elastic": {"type": "string" },
      "riak": {"type": "string" },
      "mail": {
        "properties": {
          "address": {"type": "string" },
          "notifications": {"type": "boolean" }
        }
      },
      "notificationSend": {"type": "boolean" }
    }
  }
}
```

# D REST-API

Tabulka D.1: Tabulka všech koncových bodů

| Koncový bod | Popis |
|---|---|
| GET /instances | Získá všechny instance |
| GET /instances/count | Získá počet instancí |
| GET /instances/:id | Získá jednu instanci podle ID |
| GET /instances/status/:id | Získá statusy instance podle ID |
| POST /instances | Aktualizuje informace o instanci |
| PUT /instances/:id | Vytvoří novou instanci |
| DELETE /instances/:id | Smaže vybranou instanci podle ID |
| POST /email | Pošle email |

# E SCREENSHOTY




Obrázek E.1: Overview aplikace

Obrázek E.2: Vytvoření nové instance

Obrázek E.3: Detail instance

Obrázek E.4: Swagger