UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

DATABÁZA ZNALOSTÍ

BAKALÁRSKA PRÁCA

2020 ZUZANA HALGAŠOVÁ

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

DATABÁZA ZNALOSTÍ

BAKALÁRSKA PRÁCA

Študijný program:	Aplikovaná informatika
Študijný odbor:	2511 Aplikovaná informatika
Školiace pracovisko:	Katedra Aplikovanej Informatiky
Školiteľ:	PaedDr. Roman Hrušecký PhD.
Konzultant:	Bc. Lukáš Chudý

Bratislava, 2020 Zuzana Halgašová

Obsah

1 Použité princípy a termíny

Táto kapitola vymedzuje termíny používané naprieč celou prácou.

Model–view–controller (MVC) Návrhový vzor, založený na oddelení modelu, pohľadu a ich riadenia.

Rozsiahlejšie termíny alebo princípy sú vymedzené v nasledujúcich sekciách kapitoly.

1.1 Reaktívne programovanie

Reaktívne programovanie je programovacia paradigma zaoberajúca sa tokom dát a propagáciou zmien. Táto paradigma hovorí o možnosti jednoduchého vyjadrenia statických alebo dynamických dátových tokov v programovacích jazykoch. Zodpovednosť za ich propagáciu však vykonáva exekučný model na pozadí.
Z definície vyplýva, že hodnota, ktorá je závislá od iných hodnôt, je automaticky zmenená na základe zmeny jej závislostí. Typickým príkladom je závislosť bunky od iných buniek (A1=B1+B2) v moderných tabuľkových procesoroch (napríklad *Microsoft Excel*). Kedykoľvek je hodnota B1 alebo B2 zmenená, je zmena exekučným modelom na pozadí propagovaná aj do bunky A1. Použitie v architektúre *MVC* indikuje automatickú synchronizáciu medzi model a view, alebo naopak.

1.2 Single-page application

Single-page application (SPA) je webová aplikácia, ktorá načítava všetky potrebné zdroje na navigáciu po stránke pri prvom načítaní. Ďalšie zdroje sú načítavané počas jej interakcie s užívateľom. Zväčša emuluje zmeny riadku URL, pričom k opätovnému načítaniu celej stránky nedochádza.

1.3 Redux

Redux je predvídateľný stavový kontajner pre JavaScript aplikácie, ktorý pomáha vytvárať konzistentné, škálovateľné a ľahko testovateľné aplikácie na rôznych prostrediach (klientske, serverové...).

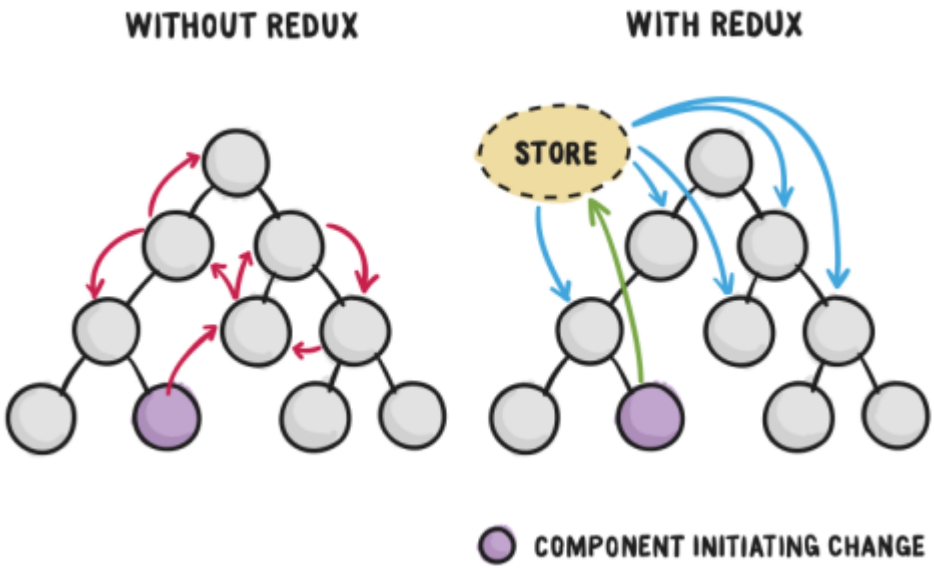
So stále narastajúcimi nárokmi na komplexnosť front-end aplikácií a využitím konceptu SPA sa stáva udržiavanie aplikačných stavov komplikované. Výslednú zložitosť udržiavania stavu komplikujú interakcie medzi jednotlivými komponentami, dátovými modelmi a ich vzájomné interakcie. Použitím kontajneru Redux sa stávajú všetky zmeny stavov predvídateľnými a udržateľnými (obr. 3.1) [9].

Na dosiahnutie tohto cieľa využíva tri princípy:

1. Jediný zdroj pravdy (single source of truth) - Stav celej aplikácie je uložený v jednom objektovom strome, ktorý je súčasťou *store*.

2. Stav je určený len na čítanie (read-only) - Jediným spôsobom, ako zmeniť stav objektu, je vykonať akciu, ktorej parametrom je objekt popisujúci požadovanú zmenu. Táto akcia je spracovaná pomocou funkcie reducer, ktorej výstupom je nový stavový objekt a zmenené listy objektového stromu sú vždy nové objekty s inou referenciou. Týmto zabezpečíme, že všetky zmeny stavu sa odohrávajú v malom počte dedikovaných funkcií.

3. Zmeny sú vykonávané pomocou čistých (pure) funkcií - Tieto funkcie sa nazývajú reducer.



Obr. 1.1: Ilustrácia porovnávajúca interakciu komponentov bez použitia a s použitím knižnice *Redux*

1.3.1 Actions

Akcie (*actions*) sú balíky informácií, ktoré slúžia na odosielanie dát z aplikácie do store. Sú jediným spôsobom, ako meniť aplikačný stav a to pomocou metódy `store.dispatch()`. Objekt akcie obsahuje jej typ *(type)* a *payload*, objekt popisujúci požadovanú zmenu akcie daného typu.

1.3.2 Reducers

Akcie popisujú požadovanú zmenu, ale neurčujú, ako sa na jej základe má zmeniť stav aplikácie. To je úlohou *reducer* funkcií. *Reducer* je čistá (*pure*) funkcia, ktorej argumentmi sú predchádzajúci stav aplikácie a akcia (*action*) a jej výstupom je nový aplikačný stav.
Keďže je táto funkcia čistá, nevyvoláva žiadne vedľajšie efekty.

Štandardná implementácia *reducer* funkcie obsahuje `switch` *statement*, prepínač rozhodujúci sa na základe typu akcie.

1.3.3 Store

Store je objekt, ktorý má nasledujúce zodpovednosti:

- Udržiava stav aplikácie.
- Umožňuje prístup k stavu aplikácie.
- Umožňuje zmenu stavu aplikácie pomocou funkcie *dispatch*, ktorej argumentmi sú akcia (*action*) a dáta (*payload*). Argumenty tejto funkcie spracováva *reducer*.
- Umožňuje registrovať počúvania na zmeny stavu.
- Stará sa o odregistrovanie počúvaní na zmeny stavu.

Aplikácia obsahuje jediný objekt typu *Store*.

1.4 Responzívny webový design

Responzívny webový design (RWD) je metóda, pri ktorej dochádza k zobrazeniu rôzneho obsahu na rozličných zariadeniach, avšak pri použití rovnakých súborov HTML a CSS [11]. Jeho účelom je zjednodušiť a spríjemniť interakciu užívateľa s aplikáciou. Prostriedkom pre vytvorenie responzívneho designu sú *media queries*, funkcionalita CSS, ktorá umožňuje aplikovať štýly na základe typu zariadenia (tlačiareň alebo obrazovka), jeho charakteristík (šírka zobrazenej plochy) alebo prostredia (znížené svetelné podmienky)

2 Použité technológie

Kapitola predstavuje relevantné technologické riešenia, ktoré boli použité pri vývoji alebo sú priamou súčasťou aplikácie.

2.1 Balíčkový manažér npm

npm je manažér balíčkov pre JavaScript a Node.js aplikácie. Bol vytvorený v roku 2009 pre potreby jednoduchého zdieľania prepoužíteľných modulov. Modul, alebo balíček je adresár, ktorý obsahuje viacero súborov a jeho súčasťou je aj súbor package.json. Tento súbor obsahuje špecifikáciu balíčka, vrátane jeho názvu, verzie a jeho závislostí od iných balíčkov. Súčasťou dnešnej bežnej webovej aplikácie sú stovky balíčkov.
Ekosystém *npm* obsahuje tri časti:

1. Webová stránka *npmjs.com* – Je užívateľsky dostupným zoznamom všetkých balíčkov, ktoré sa nachádzajú v registri.

2. Register – Implementácia repozitára všetkých balíčkov.

3. CLI (command-line interface) – Program spustiteľný v konzole. Je prostredím pre interakciu s registrom. Umožňuje inštalovať existujúce balíčky, vytvárať nové a ich následnú publikáciu do registra. CLI inštaluje balíčky do adresára *node modules*, ktorý je neoddeliteľnou súčasťou každého dnešného webového projektu.

2.2 TypeScript

TypeScript je open-source programovací jazyk, ktorý vznikol v roku 2012 v dielňach spoločnosti Microsoft. Nie je samostatne spustiteľný v prehliadači, ale je nadmnožinou jazyka JavaScript, do ktorého sa kompiluje. Platný program v JavaScripte je aj platným programom TypeScriptu. Zachováva syntax JavaScriptu, pričom je silne ovplyvnený jazykom C (autor TypeScriptu, Anders Hejlsberg, je aj architektom jazyka C). O kompiláciu sa stará prekladač napísaný s použitím prostredia *Node.js*. Je súčasťou balíčkového systému *npm*. Ako názov napovedá, hlavným cieľom je priniesť do jazyka statické typovanie. Do jazyka pridáva niekoľko vlastností, ktoré predurčujú jazyk k využitiu v komplexných aplikáciách:

Statické typovanie: Jazyk podporuje dobrovoľné statické typovanie premenných, metód, funkcií, objektov, atď. Zároveň poskytuje existujúce typy pre objekty, ktoré nie sú vytvorené používateľom (napríklad typy objektu window, ktorý je súčasťou každého prehliadača v rámci Web APIs). Typy vieme priradiť pomocou balíčka DefinitelyTyped2 aj pri existujúcej knižnici, aj keď nie sú jej vnútornou súčasťou. Kontrola typov je iba súčasťou kompilácie, vo výslednom balíčku sa typy nenachádzajú.

Access Modifiers: Pridáva možnosť použitia príznakov `public`, `private` a `protected` pre atribúty tried.

Parametre súčasťou konštruktora triedy: Parametre konštruktora triedy, ktoré sú označené jedným z príznakov, sa automaticky stávajú aj jej atribútmi.

2.2.1 Statická analýza kódu pomocou TSLint

TSLint je derivátom balíčka ESLint, vyvíjaný pod záštitou spoločnosti *Palantir*. Je rozšírením statickej analýzy TypeScript zdrojového kódu. Kontroluje kód pre jeho čitateľnosť, udržiavateľnosť a pomáha predchádzať chybám funkcionality. Pri správnej konfigurácii sa projektové pravidlá nachádzajú v súbore *tslint.json*. TSLint obsahuje natívne viac ako 100 pravidiel, ktoré je možné rozšíriť dodatočnými knižnicami (napríklad *codelyzer*, knižnica obsahujúca rozširujúce pravidlá pre statickú analýzu *Angular* projektov).

Príklady pravidiel sú:

- `semicolon` – Pravidlo vynucuje používanie bodkočiarky na konci každého tvrdenia.
- `prefer-const` – Pravidlo vynucuje používanie kľúčového slova pre konštantu (*const*), namiesto kľúčových slov pre premennú (*var*) a blokovú premennú (*let*) vždy, keď je to možné.
- `no-eval` – Pravidlo zakazuje použitie potencionálne nebezpečnej evaluačnej funkcie *eval*.

2.3 Webový framework Angular

Informácie v tejto podkapitole mam zo zdroja [1].

Príchod nových frameworkov a knižníc vo front-endovom vývoji reaguje na nedostatok oddelenia zodpovednosti (*separation of concerns*) predošlých riešení. Podľa analýzy služby *builtwith.com* až 88,5% webových stránok (stav k decembru 2017) využíva knižnicu *jQuery*, ktorej použitie vedie k miešaniu zodpovedností získavania, transformácie, samotného zobrazovania dát a užívateľskej interakcie. V roku 2016 vydáva spoločnosť *Google* framework

s názvom *Angular*, ktorý je založený na návrhovom vzore *MVC*. Je nástupcom jeho predchodcu *AngularJS*, pričom sa však jedná o iný, nie spätne kompatibilný, nástroj. V komunite dochádza často k ich zámene, čo je jeden z dôvodov, prečo sa *Angular* často označuje aj ako *Angular 2* alebo Angular 2+. Ako vývojový jazyk používa primárne TypeScript, v ktorom je aj samotný framework napísaný.

Komponenty (Components): Komponent je základným stavebným blokom každej *Angular* aplikácie. Je definovaný ako: „výrez obrazovky, ktorý môžeme nazvať pohľadom a ktorý deklaruje prepoužíteľné UI stavebné bloky pre aplikáciu“. Obsahuje šablónu a k nej prislúchajúce štýly. Každý komponent má svoj životný cyklus, na ktorého udalosti vieme vykonávať vlastné akcie. V architektúre *MVC* predstavuje *view* (pohľad).

Direktívy (Directives): Na rozdiel od komponentu, direktíva neobsahuje šablónu. Taktiež má svoj životný cyklus. Jej úlohou je modifikovať správanie komponentov alebo HTML elementov.

Services: Úlohou komponentov je zobrazovanie dát, nie ich získavanie alebo ukladanie. Túto úlohu plnia *services*, ktoré predstavujú v *MVC* architektúre *controller* (radič). Do komponentov sa vkladajú pomocou vkladania závislostí (*dependency injection*), a teda sú označené dekorátorom `@Injectable`.

Modul: Každý použitý komponent a direktíva musia byť, uvedením v časti `declarations`, členmi modulu. Moduly sú logické celky aplikácie. Medzi modulmi môžu vznikať vzájomné závislosti. Všetky použité moduly musia byť súčasťou aplikačného modulu `AppModule`, alebo súčasťou modulov, ktoré sú jeho potomkami.

2.3.1 Detekcia zmien

Funkcionalita, odlišujúca *Angular* od jeho predchodcu *Angular*, ktorým je nová verzia silno ovplyvnená, od iných frameworkov a knižníc používaných pri vývoji front-endu, je zabudovaná automatická detekcia zmien.

Detekcia zmien (*change detection*) je proces, ktorý zabezpečuje synchronizáciu medzi pohľadmi (views) a modelmi jednotlivých komponentov. Detekcia v *Angular* je tiež, ako v jeho predchodcovi, obojsmerná, s rozdielom, že tok informácií je jednosmerný a obojsmernosť detekcie je syntaktickým cukrom nad dvoma jednosmernými tokmi. *AngularJS* implementoval detekciu zmien pomocou veľkého množstvá *watchers* (každý atribút, ktorý bol prepojený so šablónou mal vlastný watcher). A preto bol každý atribút skontrolovaný pri každom štarte životného cyklu. Tento systém sa nazýva *dirty checking* a bol jediným dostupným mechanizmom na detekciu zmien. Detektor zmien prechádza každý list stromu komponentov iba raz, vždy začínajúc v koreni. Z toho vyplýva, že rodičovský komponent je skontrolovaný vždy skôr ako jeho potomok.

V novej verzii frameworku máme dostupné stratégie dve.
Nastavujú sa na úrovni komponentov:

Default: *Angular* po každej asynchrónnej akcii označí komponenty, ktoré by mohla akcia ovplyvniť, na kontrolu. Zmena atribútu v *JavaScript* objekte nevytvára objekt nový, ale modifikuje jeho inštanciu. Referencia tohto objektu sa nemení, a preto porovnanie objektov neodhalí zmenu v ich vnútre. Pri kontrole preto porovnáva predošlé a súčasné hodnoty všetkých atribútov použitých v šablóne. V zložitej aplikácii prebieha mnoho asynchrónnych operácií, preto sú komponenty neustále kontrolované. Problémy vo výkone často vidíme pri použití v komponentoch zobrazujúcich dlhé zoznamy, obsahujúcich veľké množstvo referencií.

OnPush: Použitím stratégie *OnPush* povieme komponentu, že k jeho objektom pristupujeme s využitím princípu *immutability*, pri ktorom nemodifikujeme objekt priamo, ale vždy pri zmene vytvárame objekt s novou referenciou. Detekcia zmien sa navyše spúšťa iba pri zmene atribútov typu `@Input`, ktoré sú do komponentov vkladané ich rodičovskými komponentami v HTML šablóne. Týmto princípom signifikantne znížime počet kontrol, čím dokážeme rapídne zvýšiť výkon aplikácie.

2.3.2 Dependency injection

Spoločným znakom takmer všetkých návrhových vzorov typu továreň (*factory*) je, že zodpovednosťou *high-level* tried je získať konkrétne inštancie svojich typov závislostí, ktoré potrebuje. Adaptujú *pull* model, z ktorého vyplýva, že jednou z ich závislostí je aj trieda zodpovedná za vytváranie alebo nájdenie objektov, ktoré chce *high-level* trieda používať. Vkladanie závislostí, alebo *dependency injection (DI)*, je návrhový vzor, ktorý na rozdiel od iných *factory* návrhových vzorov využíva *push* model, pomocou princípu *Inversion of Control*. Použitie tohto princípu prenáša zodpovednosť za priradenie závislostí triede, ktorá *high-level* triedu vytvára.

V kontexte *Angular* aplikácie je najčastejším použitím *DI* vloženie *service* do komponentu. Triedy, ktoré sú použiteľné na vloženie, sú označené TypeScript dekorátorom `@Injectable`. Pod textom sa nachádza minimalistická reprodukcia kódu, ktorá znázorňuje tento prípad použitia.

```
// Module
import { NgModule } from '@angular/core';

@NgModule({
    declarations: [],
    imports: [],
    exports: [],
    providers: [FeatureService],
})
export class FeatureModule {}

// Service
import { Injectable } from '@angular/core';

@Injectable()
export class FeatureService {
    getFeature() {
        // ...
    }
}

// Component
@Component({
    selector: 'app-feature',
})
export class FeatureComponent  {
    feature;
    constructor(private featureService: FeatureService) {
        this.feature = featureService.getFeature();
    }
}
```

2.3.3 Angular CLI

Jedným z faktorov, ktoré bránia technológiám pri dostatočnej penetrácii na trh, je jednoduchosť ich použitia. Donedávna bolo možné vytvoriť projekt v *Angular* napísaním vlastného štartovacieho skriptu, alebo využiť jeden z mnohých *starter* projektov, ako *angular-starter* alebo *angular-seed*. Dnes je súčasťou *Angular* repozitáru aj projekt s názvom *Angular CLI*. Program spúšťaný v príkazovom riadku nám umožňuje vygenerovanie nového projektu, automatické vytváranie súčastí aplikácie (komponentov, direktív, modulov...), obsluhuje kompiláciu kódu a spúšťanie vývojového servera. Každá z týchto funkcionalít je, na rozdiel od použitia *starter* balíčkov, vysoko konfigurovateľná pomocou vstupných parametrov CLI.

Webpack: Webpack, štandard moderného front-endu, je aj súčasťou *Angular CLI*. Ide o balíkovač *JavaScript* modulov. Jeho vstupom môžu byť *ES Modules*, *Common JS* a AMD

moduly, čím sa stáva vhodným nástrojom na zjednotenie veľkého množstva kódu z rozličných zdrojov. Kompiláciou vytvára jednotlivé alebo viacpočetné súbory (*bundles*), ktoré sú pri štarte aplikácie načítavané asynchrónne, čím skracujú čas načítania.

2.4 Event-based reaktívna knižnica RxJS

RxJS je knižnica, ktorá je implementáciou v skupine knižníc *ReactiveX* pre *JavaScript* a využíva koncepty *event-based* reaktívneho programovania na skladanie asynchrónnych tokov dát pomocou návrhového vzoru *Observer*. Funkčnosť moderných webových aplikácií je závislá od množstva asynchrónnych udalostí (typicky požiadavka na webovú službu, udalosti myši alebo klávesnice, atď.). Vrstvenie týchto udalostí je náročné na udržateľnosť kvality kódu. Motiváciou pre vytvorenie a použitie tohto vzoru v *JavaScript* je snaha o vyhnutie sa tzv. *callback hell*, veľkého množstva zanorených volaní, ktoré tvoria nečitateľný kód. Udalosti sa pri použití RxJS reťazia, v protiklade k ich zanoreniu.

Triedy, ktoré nám RxJS poskytuje sú:

Observable: Trieda reprezentujúca tok dát. Je pozorovateľná (observable), takže pozorovateľ (observer) ju môže pozorovať použitím operátora `subscribe`.

Subject: Je rozšírením triedy typu Observable, pričom do toku môžeme vkladať hodnoty pomocou operátora `next`.

Subscription: Je inštanciou vzťahu medzi triedami *Observable* alebo *Subject* a ich pozorovateľom.

Kľúčovou vlastnosťou knižnice nie je len vytváranie tokov a ich pozorovanie, ale aj možnosť ich modifikácie, spájania, rozdeľovania, k čomu nám slúžia desiatky operátorov rozširujúcich triedu *Observable*.

2.5 Súbor reaktívnych knižníc @ngrx/platform

@ngrx/platform je monorepozitár, repozitár obsahujúci viacero projektov, ktorého súčasťou sú knižnice pre reaktívne programovanie v *Angular*.

2.5.1 Redux-like implementácia @ngrx/store

@ngrx/store je implementáciou *Redux* knižnice pre *Angular*framework. Knižnica implementuje princípy z knižnice *Redux*, ale neobmedzuje sa výhradne na ne:

Použitie RxJS: Objekt typu *Store* využíva výhody knižnice RxJS, t. j. možnosť sledovať zmeny v stavovom kontajneri pomocou operátora subscribe. Trieda *Store* je implementovaná ako rozšírenie triedy *Observable*.

Vkladanie závislostí (dependency injection): Knižnicu integrujeme do projektu vložením `StoreModule` do sekcie `imports` v našom aplikačnom module. Vykonanie tohto kroku nám umožňuje vkladať *Store* na potrebné miesta.

Využitie stratégie detekcie zmien OnPush: Použitie stavového kontajnera nám neumožní zvýšiť len čitateľnosť a udržateľnosť aplikácie, ale aj jej výkon. Tvorcovia knižnice uvádzajú koncept takzvaných *smart* a *dumb* komponentov.
Zodpovednosťou *smart* komponentov je delegovať stav na svojich potomkov pomocou input atribútov v HTML šablóne, jedná sa o *top level* komponenty zaobaľujúce kľúčový výsek pohľadu.
Použitím stratégie *OnPush* závisia *dumb* komponenty na zmene ich atribútov z rodičovského komponentu, čím znížime nutnosť spúšťania detekcie zmien na nevyhnutné minimum.

2.5.2 Model vedľajších efektov @ngrx/effects

@ngrx/effects je nadstavba nad *@ngrx/store* knižnicou, ktorá umožňuje počúvať na akcie (*actions*) zo store a na ich základe vyvolávať iné akcie.

Typickým príkladom je inicializácia dát:

1. Komponent požaduje načítanie dát odoslaním (*dispatch*) akcie (*action*) `DATA_LOAD`.
2. *Reducer* akcie (*action*) je prázdny, takže nevykoná žiadnu zmenu stavu, a teda vráti starý stav.
3. Definície efektov obsahujú efekt naviazaný na akciu `DATA_LOAD`, tento efekt volá požiadavku na webovú službu typu *REST*.
4. Služba vracia odpoveď:
   (a) Odpoveď je správna:
      i. Efekt vyvolá akciu (*action*) typu `DATA_LOAD_SUCCESS`, ktorej *payload* sú požadované dáta.
      ii. *Reducer* akcie (*action*) uloží dáta zo služby do objektu typu *store*.
      iii. Dáta zobrazíme užívateľovi.
   (b) Odpoveď obsahuje chybu:
      i. Efekt vyvolá akciu (*action*) typu `DATA_LOAD_ERROR`, ktorej *payload* je popis chyby, alebo chybová hláška.
      ii. *Reducer* akcie (*action*) uloží chybu zo služby do objektu typu *store*.
      iii. Užívateľovi zobrazíme chybovú hlášku.

Literatúra

[1] Dokumentácia k webovému frameworku Angular,
`https://angular.io/docs`