Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Porovnanie návrhových vzorov Flux v jazyku Dart a Redux v ECMAScript® 2016 a ich vplyv na vývoj single-page aplikácie

Bakalárska práca

2017
Alena Poláchová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Porovnanie návrhových vzorov Flux v jazyku Dart a Redux v ECMAScript® 2016 a ich vplyv na vývoj single-page aplikácie

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Jakub Uhrík

Bratislava, 2017
Alena Poláchová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Alena Poláchová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Porovnanie návrhových vzorov Flux v jazyku Dart a Redux v ECMAScript® 2016 a ich vplyv na vývoj single-page aplikácie
Comparison of design patterns Flux in programming language Dart and Redux in ECMAScript® 2016 and their impact on the development of a single-page application

Cieľ: Porovnanie použitia návrhového vzoru Flux v programovacom jazyku Dart s použitím návrhového vzoru Redux v programovacom jazyku ECMAScript® 2016 pri vývoji single-page aplikácií. Zameranie pozornosti komparácie na jednoduchosť a udržateľnosť zdrojového kódu. Ilustrovanie vyššie spomenutých vzorov použitých v daných programovacích jazykoch na konkrétnom príklade migrácie aplikácie z návrhového vzoru Flux v programovacom jazyku Dart do návrhového vzoru Redux v programovacom jazyku ECMAScript® 2016.

Vedúci: Mgr. Jakub Uhrík
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 24.10.2016

Dátum schválenia: 24.10.2016 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Tu môžete poďakovať školiteľovi, prípadne ďalším osobám, ktoré vám s prácou nejako pomohli, poradili, poskytli dáta a podobne.

Abstrakt

Slovenský abstrakt v rozsahu 100-500 slov, jeden odstavec. Abstrakt stručne sumarizuje výsledky práce. Mal by byť pochopiteľný pre bežného informatika. Nemal by teda využívať skratky, termíny alebo označenie zavedené v práci, okrem tých, ktoré sú všeobecne známe.

Kľúčové slová: Dart, ECMAScript® 2016, Flux, Redux

Abstract

Abstract in the English language (translation of the abstract in the Slovak language).

Keywords:

Obsah

Úvod

Tu bude úvod do problematiky o mojej bakalárskej práci, načrtnutie problému a stručné popísanie obsahu jednotlivých kapitol.

Kapitola 1

Prostredie

V tejto kapitole si predstavíme vstupný popis kódu, ktorý sa budeme snažiť podľa zadaných podmienok zmeniť.

Pôvodná aplikácia je napísaná v jazyku Dart, ktorý bližšie popíšeme v podkapitole 2.1. Je to internetová aplikácia bežiaca v prehliadači založená na princípoch SPA-aplikácie.

1.1 Single-Page Application (SPA)

SPA, teda aplikácia fungujúca na jedno načítanie je model internetovej aplikácie. Ponúka rýchlosť desktopovej aplikácie a zároveň dostupnosť internetovej stránky.

Celá stránka je načítaná len raz, a to na začiatku. Logika aplikácie je potom kontrolovaná skriptom v prehliadači na strane klienta. Komunikácia so serverom prebieha len v malom množstve prípadov, ako je napríklad validácia údajov, autentifikácia alebo dostupnosť zdieľaných dát. Taktiež v čase, keď klient komunikuje so serverom, je možné zobraziť používateľovi vhodnú hlášku o spracovaní dát (na rozdiel od aplikácií, kde je stránka generovaná na serveri a zobrazí sa klientovi až po úplnom načítaní).

V takýchto aplikáciách sa často využíva práve jazyk JavaScript. Veľkou výhodou je multiplatformová dostupnosť vďaka internetovým prehliadačom a bez nutnosti inštalácie ďalších podporných programov. Podrobnejší popis sa dá nájsť v manuáli o SPA [8].

Kapitola 2

Programovacie jazyky Dart a ECMAScript® 2016

V tejto kapitole si povieme niečo o programovacích jazykoch Dart a ECMAScript® 2016, s ktorými budeme počas celej práce robiť.

2.1 Dart

V tejto sekcii si predstavíme hlavné črty programovacieho jazyka Dart. Dart je objektovo orientovaný programovací jazyk. Je založený na definovaní tried, kde trieda môže dediť od najviac jednej inej triedy. Jazyk Dart je voliteľne typovaný. (Túto vlastnosť si bližšie popíšeme v podkapitole 2.1.2). Informácie do tejto kapitoly boli čerpané najmä zo špecifikácie jazyka [1].

2.1.1 Triedy

Trieda (*class*) definuje formu a správanie určitej množine objektov. Tieto objekty nazývame inštancie (*instance*) danej triedy. Trieda môže byť definovaná deklaráciou samotnej triedy, alebo pomocou mixinov.

Trieda má konštruktory a členy (členy danej inštancie a statické členy). Členy sú metódy, premenné, gettery a settery.

Nadtrieda Každá trieda má práve jednu nadtriedu (*superclass*) okrem triedy *Object*, ktorá nemá. Táto nadtrieda sa uvádza za kľúčovým slovom *extends*, alebo je určená implicitne ako *Object*. Daná trieda dedí od nadtriedy všetky dostupné členy danej inštancie, ktoré neboli preťažené (*override*).

Rozhranie Trieda môže implementovať (*implements*) niekoľko rozhraní (*interface*). Rozhranie definuje, ako by sa malo pracovať s objektom. Má metódy, gettery, settery

a množinu „nadrozhraní“, ktoré rozširuje.

Mixin Mixin opisuje rozdiel medzi triedou a jej nadtriedou. Mixin je vždy odvodený od deklarácie existujúcej triedy. Mixin môžeme použiť pri definovaní novej triedy pomocou kľúčového slova *with M* (kde M je mixin). Mixin je užitočný v prípadoch, kedy viacerým zdanlivo nezávislým triedam chceme pridať rovnakú funkcionalitu.

Abstraktná trieda Trieda môže byť abstraktná, vtedy ju definujeme kľúčovým slovom *abstract*. Takáto trieda nemusí mať implementované všetky metódy. Mixiny sú abstraktné triedy.

Konštruktor Konštruktor triedy je špeciálna funkcia, ktorá vytvára inštanciu triedy. Volá sa rovnako ako trieda, ktorej prislúcha. Ak nie je špecifikovaná, volá sa v implicitnom konštruktore konštruktor nadtriedy.

2.1.2 Typy

Programovací jazyk Dart podporuje voliteľné typovanie založené na typoch rozhrania.

Statické typy Statické typy sú použité pri deklarovaní premenných, pri definícii návratových hodnôt funkcií a v ohraničení typu premenných. Tieto statické typy sú použité iba pri statickej kontrole a v kontrolovanom móde. Na produkčný mód nesmú mať žiaden vplyv. Medzi základné typy patria:

- konštanty (*constants*): čísla (*number*), booleovské premenné (*bool*), reťazce znakov (*string*), nulový objekt (*null*)
- kolekcie viacerých prvkov: zoznamy (objekt *List*), mapy (objekt *Map*)

Pri produkčnom móde sa všetky typy nahradia jednotným typom *dynamic*. Ten označuje neznámy typ.

Dynamic Typ *dynamic* má definovanú každú možnú operáciu s všetkými možnými počtami parametrov. Návratová hodnota týchto operácií je vždy *dynamic*. Chceme tak zabezpečiť, aby nám typ *dynamic* nikdy nevrátil chybovú hlášku o nesprávnom type. *Dynamic* je považovaný za typový objekt, aj keď to nie je trieda.

Void Typ *void* možno použiť len ako návratovú hodnotu funkcie. *Void* nie je považovaný za typový objekt. Môže oznamovať upozornenia v kontrolovanom móde, ak funkcia vracia inú hodnotu ako *null*.

Null Rezervované slovo *null* označuje null-ový objekt. Je to jediná inštancia triedy *Null*. Rozširovanie, implementovanie alebo použitie tejto triedy ako mixin spôsobí chybu pri kompilácii (*compile-time error*). Volanie akejkoľvek metódy na objekte *null* spôsobí chybu.

This Slovo *this* označuje aktuálne používanú inštanciu triedy, s ktorou pracujeme. Statický typ *this* je potom rozhranie tejto triedy.

2.1.3 Premenné

Premenné sú úložiská v pamäti. Neinicializovaná premenná má hodnotu *null*.

Static variable je taká premenná, ktorá nie je asociovaná s konkrétnou inštanciou, ale s celou knižnicou alebo triedou.

Final variable je taká premenná, ktorá je zviazaná s konkrétnym objektom od jej deklarácie. Spôsobuje *static warning* ak je inicializovaná aj v konštruktore. Spôsobuje *compile-time errror* ak nie je inicializovaná pri deklarácii. *Constant variable* je implicitne *final*.

Ak deklarácia nešpecifikuje typ premennej, tak je *dynamic*, čo predstavuje neznámy typ.

Gettery a settery Ku premenným pristupujeme pomocou prirodzených getterov a setterov. Getter je funkcia bez argumentov, ktorá v čase zavolania vyhodnotí výraz, ktorý definuje danú premennú a vráti výsledok. Setter je funkcia s jedným argumentom, ktorá danej premennej priradí hodnotu jej argumentu. *Final* premenné nemajú settery.

2.1.4 Funkcie

Funkcie predstavujú vykonateľné akcie. Funkcie pozostávajú z deklarácií, metód, getterov, setterov, konštruktorov. Každá funkcia má 2 časti: popis(signature) a telo(body). Popis funkcie obsahuje formálne parametre a môže obsahovať typ návratovej hodnoty. Telo funkcie môže mať 2 tvary:

- blok príkazov v zložených zátvorkách ({, }). Ak tento blok príkazov neobsahuje príkaz *return*, automaticky sa na koniec pridáva s návratovou hodnotou *null*.
- $=>$ *e*, čo je ekvivalentné *{return e;}*

Oba bloky príkazov môžu byť vykonané synchrónne(modifikátor *sync**) alebo asynchrónne(*async, async**).

Každá funkcia má zoznam parametrov, ktorý obsahuje zoznam povinných pozičných parametrov, potom zoznam voliteľných parametrov. Voliteľné parametre môžu byť pomenované alebo pozičné ale nie obe súčasne.

Ak sa neuvedie typ návratovej funkcie explicitne, jej typ je *dynamic*, alebo daná trieda, ak ide o konštruktor.

Externá funkcia(*external*) je funkcia, ktorá má deklaráciu a telo funkcie na rôznych miestach v kóde. Môžu to byť napríklad funkcie implementované externe v inom programovacom jazyku alebo také, ktoré sú dynamicky generované ale ich popis je statický a známy.

2.1.5 Upozornenia a chyby

Dart rozlišuje niekoľko druhov chýb. Napríklad:

- Kompilačné chyby (*compile-time errors*) teda chyby v čase kompilácie, ktoré bránia ďalšiemu behu programu. Tieto musia byť nahlásené kompilátorom pred spustením samotného chybného kódu.
- Statické upozornenia (*static warnings*) sú chyby zistené statickou kontrolou (teda nie za behu programu). Nemajú žiaden efekt v čase behu programu. Statická kontrola sa týka najmä konzistentnosti typov, ale neznemožňuje kompiláciu ani beh samotného programu. Statickú kontrolu by mali zabezpečovať vývojové prostredia a kompilátory.

Módy behu programu Programy môžu byť spustené v dvoch módoch:

- Checked mode (*kontrolovaný mód*) - v tomto móde fungujú *static warnings* aj *compile-time errors*. Je vhodný na písanie kódu a ladenie programu.
- Production mode (*produkčný mód*) - ako samotný názov napovedá, tento mód je určený na beh programu u klienta. V tomto móde sa *static warnings* nevyskytujú, sú tu len *compile-time errors* a chyby, ktoré sa vyskytli priamo pri behu aplikácie.

2.1.6 Súkromie

Dart podporuje dve úrovne súkromia, *private* (súkromný) a *public* (verejný). Objekt, ktorý je deklarovaný s kľúčovým slovom *private* je súkromný, inak je každý objekt verejný. Tiež môžeme definovať súkromný objekt tak, že jeho názov začína podčiarkovníkom („_“).

Programy v jazyku Dart sú organizované do knižníc. Objekt je dostupný v knižnici len ak je definovaný v danej knižnici, alebo ak je *public*.

2.1.7 Knižnice

Program v jayzku Dart pozostáva z jednej alebo viacerých knižníc. Môže byť vytvorený z viacerých kompilačných jednotiek (*compilation units*). Kompilačná jednotka môže byť

knižnica alebo *part*. Knižnice sú jednotkami súkromia. Kód definovaný vrámci knižnice ako súkromný je dostupný len vrámci danej knižnice.

Knižnica pozostáva z množiny importov, exportov a verejne deklarovaných objektov. Tieto môžu byť triedy, funkcie alebo premenné.

Import Kľúčové slovo *import* prepája knižnice. Určuje, ktoré knižnice môžu byť použité pri programovaní inej knižnice. *Import* umožňuje explicitne povedať, ktoré objekty z kategórie *public* chceme skryť, tie uvedieme za kľúčovým *hide*. Alebo vybrať podmnožinu objektov, ktoré chceme ponechať (a ostatné sa nám skryjú) pomocou kľúčového slova *show*. Ak by sa nám mohlo stať, že názvy funkcií alebo premenných sa vo viacerých knižniciach prekrývajú (čo je problém a snažíme sa tomu zabrániť), môžeme premenovať dané objekty pomocou kľúčového slova *as*. Podobne sa dá pomenovať aj samotná knižnica, kde potom voláme prvky knižnice nasledovne: *libraryName.objectName*.

Export Kľúčové slovo *export* definuje množinu objektov, ktoré sú prístupné po importovaní danej knižnice L, v ktorej sa *export* nachádza. Môžeme exportovať množinu objektov, alebo celú knižnicu. Tiež môžeme obmedziť export knižnice pomocou *show* a *hide* rovnako, ako pri importovaní.

Parts Ak máme veľkú knižnicu, môžeme ju rozdeliť do viacerých súborov pomocou *part* a *part of*. Všetky definície objektov, aj súkromné, sú medzi týmito časťami vzájomne viditeľné.

Hlavný súbor obsahuje kľúčové *part*, kde pomenuje cestu ku druhému súboru, ktorá reprezentuje časť knižnice. Tá pomenuje, ku ktorému hlavnému súboru prislúcha za kľúčovým *part of*. Importovanie ďalších knižníc potom stačí uviesť v hlavnom súbore.

Scripts *Skript* sa nazýva knižnica, ktorá obsahuje funkciu *main*. Takáto funkcia môže byť v jednom projekte práve jedna. Je to funkcia, ktorá sa spúšťa na začiatku programu.

Pub O to, aby boli všetky knižnice, ktoré sú importované, dostupné a aktualizované zabezpečuje špeciálny správca knižníc *pub* (*package manager*). Každá knižnica má zoznam závislostí - aké verzie cudzích knižníc používa. Pub vyžaduje špeciálny súbor (pubspec.yaml) obsahujúci zoznam knižníc s požadovanými verziami ku každej knižnici. Na základe tohoto súboru pracuje príkaz *pub get*, ktorý stiahne potrebné zmeny.

Okrem iného spravuje *pub* mená publikovaných knižníc, aby sa zabránilo kolíziám.

2.1.8 Súbežnosť

Kód v jazyku Dart je vždy jednovláknový. Ak chceme vykonávať viac súbežných činností, používame špeciálnu entitu *isolates*, ktorá má vlastnú pamäť a vlastnú kontrolu

vlákna. Tieto entity medzi sebou komunikujú pomocou posielania správ, nezdieľajú žiaden stav.

Dart podporuje asynchrónnosť vykonávania programu. Kľúčovým *await* odovzdáme kontrolu, pokým sa výraz za *await* vyhodnotí. Ak sa nevie vyhodnotiť v čase vykonávania tejto inštrukcie, namiesto hodnoty sa vytvorí inštancia triedy *Future*, za ktorú sa neskôr po dopočítaní dosadí daná hodnota výrazu.

2.1.9 Syntaktické pomôcky

Kaskádová notácia Kaskádová notácia má tvar *e..sufix*, kde *e* je nejaký výraz a *sufix* je postupnosť operátorov, metód geterov alebo seterov na danom výraze. Táto konštrukcia je ekvivalentná *(t){t.sufix; return t;}(e)*;

Zaujímavé je jej použitie, keď môžeme na jednom objekte zavolať viacero metód zasebou, bez toho, aby sme ju znova vypisovali. Pritom metódy, ktoré použijeme, nemusia mať návratovú hodnotu daný objekt. Príklad môžeme vidieť v ukážke 2.1 z internetovej stránky o jazyku dart[7].

```
querySelector('#button') // Get an object.
  ..text = 'Confirm'   // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

Listing 2.1: Kaskádová notácia

2.2 ECMAScript® 2016

V tejto podkapitole si predstavíme hlavné črty programovacieho jazyka ECMAScript® 2016.

JavaScript je objektovo orientovaný jazyk, ktorý upravuje internetové stránky v prehliadači a vykonáva výpočty v prehliadači. (Skriptovací jazyk je programovací jazyk zameraný na výpočty a manipuláciu s objektami už existujúceho systému.) Dnes je to plne vybavený všeobecne navrhnutý objektovo orientovaný programovací jazyk. Okolo tohoto programovacieho jazyka sa vytvorila veľká komunita, ktorá udržiava tento jazyk stále živý a veľmi používaný.

Organizácia ECMA International® definuje špecifikácie aj tohoto jazyka. V súčasnosti je najnovší ECMAScript® 2016, inak sa označuje aj ako 7. edícia. V tejto práci sa špecifikujeme na túto verziu jazyka, aj keď spomenieme niektoré vlastnosti, ktoré ešte nie sú v špecifikácii, ale sa pripravujú do ďalšej verzie. Presnú špecifikáciu možno nájsť v literatúre [2, ECMAScript® 2016]. Do tejto práce boli informácie o JavaScripte čerpané najmä zo série kníh [16, 11, 12, 15, 13, 14, You Don't Know JS]. Pomenovania JavaScript a ECMAScript sú pre tento jazyk ekvivalentné. V niektorých častiach práce

spomíname konkrétnu verziu jazyka ES6, ktorá obsahuje veľa nových štruktúr oproti predošlým verziám.

2.2.1 Triedy

JavaScript má niektoré syntaktické prvky, ktoré sa spájajú s triedami, ako napríklad *new*, *class* alebo *instanceof*. Avšak triedy ako také nemá. Na triedy sa môžeme pozrieť ako na návrhový vzor.

Objekty Všetko v JavaScripte sú vlastne objekty a teda aj trieda je objekt. Funkcie, ktoré sú volané s kľúčovým *new* sa bežne volajú konštruktory, aj keď v JavaScripte nevytvoria štandardnú triedu ako v iných triedovo orientovaných jazykoch. Ak hovoríme o inštancii triedy, myslíme tým kópiu daného objektu. Polymorfizmus na inštanciách triedy je opäť len výsledkom kopírovania vlastností. Preto aj odvodená trieda nemá odkaz na rodičovskú triedu, má od nej len nakopírované potrebné údaje.

Rozšírenia Mixin/extends pridáva špecifickú funkcionalitu z iného objektu. Pridáva ju kopírovaním. Avšak robí toto kopírovanie iba na prvej úrovni, teda ak hodnoty, ktoré sú ukladané pod kľúčmi nie sú primitívne, objekty, sú to objekty zdieľané referenciou. Čiastočne môžeme používať viacnásobné dedenie, ale nevyhneme sa kolíziám pri kopírovaní prvkov s rovnakým menom z viacerých zdrojov.

Prototype V JavaScripte existuje možnosť, ako previazať objekty medzi sebou. Každý objekt má vlastnosť prototype(predvolená hodnota je na Object.prototype). Táto vlastnosť nám umožňuje používať funkcie, ktoré sme objektu nešpecifikovali explicitne (napríklad funkcia toString).

Pomocou prototype môžeme robiť dedenie, ktoré sa veľmi podobá tomu z tried. Prototype nekopíruje, ale robí odkaz na objekt, "kam sa pozeráme, keď nevieme, aký objekt máme použiť". Na programovanie v JavaScripte sa však môžeme pozrieť aj ako na delegovanie správania medzi objektami, namiesto dedenia medzi triedami.

Nový objekt Volanie funkcie s kľúčovým *new* vytvorí nový objekt a vykoná telo funkcie. Funkcia ktorá je volaná ako konštruktor, nie je ničím iná od obyčajnej funkcie a každá môže vytvoriť objekt. Ak tejto funkcii pridáme prototype, budú ho mať všetky z nej odvodené objekty. Funkcia, ktorú nastavíme pre prototype sa nekopíruje medzi ostatné objekty, ale je zdieľaná. Ak chceme, aby program pridal linku na iný objekt (prototype) za nás, môžeme namiesto kľúčového slova *new* použiť funkciu *Object.create(. . .)*, ktorá vytvorí objekt za nás.

Triedy Hoci v JavaScripte triedy ako také nie sú, od verzie ES6 existuje pre JavaScript aj kľúčové slovo *class*, ktoré zaobaľuje prácu s prototypmi a snaží sa vytvoriť hierarchiu dedenia ako majú triedovo orientované (*class-oriented*) jazyky. Tiež podporuje *extends* a *super*, ako ich poznáme z iných jazykov.

O objektoch, ich prototypoch a rozšíreniach podrobnejšie možno nájsť informácie v knihe [12, this & Object Prototypes].

2.2.2 Typy

Jazyk ECMAScript® 2016 nie je typovaný jazyk. Napriek tomu rozlišuje niekoľko základných typov, na ktorých má definované konkrétne operácie. Typy jazyka ECMAScript sú *Undefined*, *Null*, *Boolean*, *String*, *Symbol*, *Number* a *Object*. Každá hodnota premennej v tomto jazyku je charakterizovaná jedným z uvedených typov. V JavaScripte nemajú typ premenné, ale hodnoty premenných, ktoré sú v nich uložené.

- *Primitívne typy* sú string, number, boolean, symbol, null, undefined a object
- *Objekty* sú ostatné typy. Každý primitívny typ má aj svoj objektový ekvivalent, na ktorom môžeme robiť operácie(napríklad zistiť dĺžku stringu). Funkcia je objekt, ktorý obsahuje aj vykonateľné príkazy. Ďalšie objekty sú napríklad Array, Date, RegExp alebo Error.

Objekty Objekty vieme definovať priamo, teda vymenovaním obsahu, alebo cez kľúčové *new*. Obe metódy vytvoria rovnaký objekt. Všeobecne sa preferuje definovanie vymenovaním prvkov. Obsah objektu môžeme plniť viacerými možnosťami, ako môžeme vidieť v ukážke 2.2. Vymenovaním (riadok 4), cez bodku (riadok 7 a 9) alebo cez hranaté zátvorky (riadok 8 a 10). Platí, že keď používame bodkovú konvenciu, môžeme mená kľúčov nazývať iba jednoslovnými názvami bez medzier a špeciálnych znakov. Do hranatých zátvoriek môžeme dať ľubovolný string vrátane medzier. Ak chceme použiť ako kľúč hodnotu premennej, musíme použiť hranaté zátvorky. Použitím rovnakého kľúča cez bodku aj v hranatej zátvorke sa dostaneme ku rovnakej hodnote.

```
1  var objectString = new String("I am String");
2  var primitiveString = "I am primitive string";
3
4  var myObject = {
5    key1: 'value1';
6  }
7  myObject.key2 = primitiveString;
8  myObject["with space!"] = 'value2';
9  console.log(myObject.key1); // 'value1'
10 console.log(myObject["with space!"]); // 'value2'
```

Listing 2.2: tvorba objektu

Undefined typ má práve jednu hodnotu *undefined*. Každá premenná, ktorá nemá priradenú žiadnu hodnotu, ale je vytvorená, má práve túto hodnotu.

Null typ má práve jednu hodnotu, *null*. *Null* predstavuje prázdny objekt. Od *undefined* sa líši tým, že *null* môže byť priradený ako hodnota do premennej.

This Kľúčové slovo *this* má v JavaScripte svoje špeciálne miesto. V iných jazykoch je to obvykle objekt, v ktorom sa nachádzame, definoval ho autor pri písaní kódu (author-time binding). V JavaScripte je to objekt, ktorý volal náš objekt a je definovaný počas behu programu (runtime binding). *This* ukazuje väčšinou na miesto, odkial bola daná funkcia volaná, teda keď sa pozrieme do zásobníka volaní, bude to funkcia, ktorá je hneď pred našou.

Pri určovaní hodnoty premennej *this* sa používajú tieto štyri pravidlá:

- Štandardné viazanie premennej this (*default binding*) predstavuje miesto, odkial bola funkcia volaná. V striktnom móde štandardné nastavenie this nefunguje.
- Implicitné viazanie premennej this (*implicit binding*), ak voláme metódu na objekte v danom čase, this ukazuje na daný objekt
- Explicitné viazanie premennej this (*explicit binding*) vznikne použitím funkcie *call()* alebo *apply()* kde ako prvý parameter dáme odkaz na *this* ktorý chceme využiť. Podskupinou je zviazanie napevno (*hard binding*), kde spravíme wrapper okolo funkcie, ktorá nastaví hodnotu this na danú nemmennú hodnotu určenú argumentom. od verzie ES6 existuje metóda *bind()* pre funkcie ktorá robí *hard binding*
- Viazanie premennej vytvorením nového objektu pomocou kľúčového slova *new* (*new binding*), vytvorí sa nová inštancia objektu, ktorá má v čase vykonávania this nasmerované na seba

Špeciálne správanie majú *arrow functions*, kde sa *this* správa ako štandardné, ktoré poznáme z iných programovacích jazykov. V takejto funkcii this odkazuje na seba. O funkciách v ECMAScript® 2016 si povieme viac v podkapitole 2.2.4.

Podrobnejšie informácie o typoch možno nájsť v knihe [15, Types & Grammar], o *this* v knihe [12, this & Object Prototypes].

2.2.3 Premenné

Premenné v JavaScripte nemajú typ. Typ majú len hodnoty uložené v týchto premenných. Toto je veľký a často mätúci rozdiel oproti iným štandardným jazykom. Na typ hodnoty sa vieme pýtať príkazom *instanceof*.

Deklarovanie premenných v ECMAScript® 2016 ECMAScript® 2016 podporuje primárne tri typy deklarovania premenných. Sú to *var*, *const* a *let*.

V prípade *const* a *let* ide o premenné, ktoré sa nedajú deklarovať dvakrát s rovnakým menom a platia len vrámci daného bloku kódu. V štandardnej terminológii by sme ich mohli nazvať aj lokálne premenné.

Deklarovanie kľúčovým *let* nám zabezpečí, že pôsobnosť premennej je iba v aktuálnom najmenšom bloku ohraničenom zátvorkami *{}*. Užitočnosť *let* môžeme vidieť napríklad aj v cykloch *for(let i = 0, ...){}*.

Narozdiel od týchto, premennú *var* môžeme deklarovať aj viackrát a môžeme sa na ňu pýtať aj mimo bloku kde sme ju deklarovali. V ukážke 2.3 vidíme viacnásobné deklarovanie, volanie premennej mimo bloku, kde bola zavolaná aj volanie premennej predtým, ako bola vytvorená. Tieto premenné by sme mohli nazvať aj globálne premenné.

Ak sa vyskytne v kóde premenná, kompilátor hľadá, kde bola definovaná. Ak ju nenájde, vytvorí novú premennú s daným menom ako globálnu. Ak beží tento skript v prehliadači, môžeme sa na túto premennú pozrieť aj ako na premennú daného okna, a pristupovať ku nej nasledovne *window.meno_premennej*. (Takto v ukážke funguje premenná *k*.)

Deklarovať môžeme aj viacero premenných naraz (riadok 12), alebo „rozbaliť“ objekt do premenných v jednom kroku (riadok 14).

```
1  var i = 2;
2  if (i >= 0) {
3    console.log( i, j ); // i = 2 , j = undefined
4    var j = 5;
5    var i = 3;
6    k = 10;
7  } else {
8    var j = 4;
9  }
10 console.log( i, j, k ); // i = 3 , j = 5 , k = 10
11
12 var a = 2, b;
13 var obj = {name:"foo", id:1};
14 var {name, id} = obj;
```

Listing 2.3: JavaScript deklarovanie

2.2.4 Funkcie

Funkcia je špeciálny podtyp objektu, je to „spustiteľný“ objekt. Obsahuje kód, ktorý sa dá vykonať. Do verzie ES5 boli všetky funkcie definované štandardne *function foo(argumenty){ ... }*. Argumenty funkcie sú voliteľné a môžu byť pomenované. Tiež im môžeme pri definovaní funkcie nastaviť nejakú preddefinovanú hodnotu. Argumenty sú

premenné, ktoré sú definované len v tele funkcie. Funkciu môžeme uložiť do premennej a pracovať s ňou ako s objektom.

Funkcia môže a nemusí mať svoje meno. Funkcia bez mena sa volá anonymná funkcia. Jej nevýhodou je, že sa na ňu nevieme odkazovať z tela funkcie a pri pozeraní zásobníka volaní funkcií nevieme o akú funkciu sa jedná.

Špeciálny typ funkcie (od verzie ES6) je šípková funkcia (*arrow function*). Primárnym motívom jej vzniku bolo zabezpečiť správanie *this* premennej ako v iných programovacích jazykoch, teda aby smerovalo na túto funkciu. Je to vždy anonymná funkcia. Vždy vystupuje ako výraz (nemá svoju deklaráciu). Má jednoduchú a krátku syntax, pred šípkou argumenty (ak je len jeden, môže sa vyskytovať bez zátvorky) a za šípkou výraz, ktorý funkcia vracia (bez kľúčového slova *return*) alebo telo funkcie obalené kučeravými zátvorkami. Šípkové funkcie môžeme vidieť na malom príklade v ukážke 2.4.

```
1  var a = [1,2,3,4,5];
2  a = a.map( v => v * 2 ); // [2, 4, 6, 8, 10]
3
4  const sum = (x, y) => x + y;
5  sum(2, 5); // 7
6
7  var div = (x, y) => {
8    if (y != 0) {
9      return x/y;
10   } else {
11     return Infinity;
12   }
13 }
```

Listing 2.4: šípkové funkcie

2.2.5 Upozornenia a chyby

Striktný mód je cesta, ako sa priblížiť kontrolovanému módu v JavaScripte. Úmyselne má mierne inú sémantiku niektorých príkazov. Medzi najväčšie rozdiely medzi striktným a štandardným módom patria tieto vlastnosti:

- niektoré tiché chyby mení na chybové hlášky (pomocou kľúčového slova *throws*)
- zabraňuje niektorým chybám, aby stroj, ktorý kompiluje a vykonáva kód, mohol lepšie optimalizovať vykonávanie programu
- zakazuje niektoré syntaktické konštrukcie(napríklad definovať viackrát rovnakú premennú, alebo priradiť hodnotu neexistujúcej premennej)
- pridáva nové rezervované slová (implements, interface, let, package, private, protected, public, static, and yield)

Striktný mód vieme zapnúť pre celý skript, alebo len pre konkrétnu funkciu. Zapína sa príkazom *'use strict';* na začiatku funkcie/skriptu. Striktný mód je podporovaný rôzne rôznymi prehliadačmi. Kód, ktorý píšeme v striktnom móde by mal byť vykonateľný aj v štandardnom móde. Viac o striktnom móde sa možno dočítať na stránke *Mozilla Developer Network* [19].

Spracovanie chýb s premennými

Priradenie hodnoty neexistujúcej premennej:

- Ak je vykonávanie v štandardnom móde, premenná sa vytvorí ako globálna a priradí sa jej daná hodnota.
- Ak je v striktnom móde, globálnu premennú nevytvorí, ale vyhodí chybu typu *ReferenceError*

TypeError znamená, že sme na existujúcom objekte chceli vykonať nelegálnu alebo neuskutočniteľnú akciu.

2.2.6 Súkromie

ECMAScript® 2016 nemá rôzne úrovne súkromia. V JavaScripte sú dve úrovne platnosti premenných. Jednou je funkcia a druhou celý súbor. Verejné objekty z daného súboru sú len tie, ktoré sú označené kľúčovým slovom *export* alebo *export default*. Ostatné sú súkromné. Vo funkcii sú všetky objekty súkromné, ak ich nevrátime v objekte ako návratovú hodnotu z funkcie.

2.2.7 Knižnice

Do verzie ES5 bol kód delený do modulov. Od verzie ES6 JavaScript poskytuje možnosť importovať a exportovať súbor alebo knižnicu.

Zo súboru môžeme exportovať najviac jeden objekt predvolene (*export default* foo). Ostatné môžeme exportovať ako pomenované (*export* bar). Iba tie objekty, ktoré exportujeme, môžeme vidieť z iných súborov. Môžeme exportovať aj všetky objekty naraz (*export **), alebo exportovať objekty z inej predtým importovanej knižnice (*export * from 'baz';*).

Ak chceme používať objekt z iného súboru, musíme ho importovať. Ak importujeme predvolene exportovaný objekt, jeho meno nedávame do zátvoriek. Každý iný objekt ale musí byť v kučeravých zátvorkách (*import defaultObject, {obj2, obj3} from 'baz';*).

Môžeme importovať aj všetky objekty z danej knižnice, avšak potom musíme poskytnúť meno, pod ktorým budeme ku objektom pristupovať (*import * as FOO from 'foo';*).

Dostupnosť

Pre jazyk ECMAScript® 2016 je dostupné množstvo knižníc. Veľká časť z nich je dostupná cez správcu npm. Vkladanie knižnice do projektu je veľmi jednoduché volanie príkazu npm s vhodnou kombináciou prepínačov. (Volanie funkcie z priečinka projektu s prepínačom *–save* bolo u nás postačujúce.)

2.2.8 Súbežnosť

Pri programovaní webových aplikácií potrebujeme písať asynchrónny kód (napríklad dotazy na server). Nemôžeme si dovoliť čakať na odpoveď, ktorá trvá dlho (a ani nevieme, či príde). Potrebujeme reagovať na používateľa. JS beží v prehliadači, on rozhoduje o tom, čo je kedy vykonané, často je kód vykonaný sekvenčne (za sebou), nie paralelne.

Callback a Promise *Callback* (spätné volanie) je funkcia, ktorá sa zavolá, keď sa daný kód vykoná. *Callback* nie je úplne spoľahlivé riešenie. Preto potrebujeme nástroj na ktorý sa môžeme spoľahnúť. *Promise* (prísľub) vykonáva asynchrónny kód. *Promise* je typ objektu, ktorý nám sľubuje, že keď sa dokončí kus asynchrónneho kódu, hodnota *promise* sa doplní vypočítanou. Je *thenable*, čo znamená, že na ňom môžeme zavolať metódu *then*, ktorá sa vykoná vtedy, keď sa naplní *promise*. Poskytuje nám spôsob, ako spracovať neúspešný pokus o vykonanie kódu. Podobne ako *then* použijeme *catch* funkciu, ktorá odchytí chybovú hlášku, ak nejaká nastala. Malo by platiť, že vždy sa vykoná vetva *then* alebo vetva *catch*. Jednoduché použitie tejto štruktúry vidíme v ukážke 2.5. Alternatívne môžeme definovať *promise* s jednou funkciou, ktorá spracuváva aj úspešný aj neúspešný výsledok (*new Promise(function(resolve, reject){...})*). *Promise* sa nevyhýba callbackom, ale vhodne ich zaobaľuje, aby sa s nimi spoľahlivo pracovalo.

Promise vie čakať na viac hodnôt (*Promise.all([...])*) alebo na prvú z množiny (*Promise.race([...])*). Promises môžeme reťaziť za sebou. To nám umožňuje rozmýšľať nad asynchrónnymi operáciami sekvenčne.

```
1  const promise = fetch(serverUrl);
2
3  promise
4    .then(response => console.log(response))
5    .catch(error => console.log(error));
```

Listing 2.5: Promise

Generátory *Generátor* je funkcia, ktorá generuje niekoľko hodnôt a možno cez tieto hodnoty iterovať. Ku jednotlivým hodnotám sa dostaneme cez volanie funkcie *next()* na danom generátore.

Je to špeciálna funkcia, ktorá obsahuje niekoľko volaní *yield*, ktoré pozastavia vykonávanie funkcie a voliteľne vrátia hodnotu ako argument tohoto volania. Ak chceme pokračovať vo vykonávaní tejto funkcie, zavoláme na generátore funkciu *next()* s voliteľnými argumentami, ktoré sa potom do kódu doplnia namiesto kľúčového slova *yield*. V ukážke 2.6 vidíme volanie generátora, volanie s argumentom aj *yield* s návratovou hodnotou.

```
1  var it = function *foo(){
2    let a = 1;
3    a += yield a;
4    yield a;
5    return a*a;
6  }
7
8  it.next(); // spusti funkciu od zaciatku, 3. riadok vrati 1
9  it.next(2); // do riadku 3 sa namiesto yield doplni 2, 4. riadok vrati 3
10 it.next(); // return vrati poslednu hodnotu (9) a nastavi flag done=true
```

Listing 2.6: Generátor

Generátor môžeme využiť pri vykonávaní asynchrónneho kódu napríklad keď čakáme na odpoveď zo servera. Hlavná funkcia *yield*-ne request na server a keď je hotový, hlavná funkcia bude zavolaná a do nej doplnená odpoveď cez argument funkcie *next()* (hlavná funkcia sa správa ako generátor).

async a await Štruktúry, ktoré nepatria do špecifikácie ECMAScript® 2016 ale mali by byť v nasledujúcich verziách (pripravuje sa špecifikácia ôsmej edície) sú *await* a *async*. Tieto štruktúry schovávajú prácu s *promise*-mi a *generátor*-mi a uľahčujú logiku volania asynchrónnych funkcií. *Async* označuje funkciu, ktorá je vykonávaná asynchrónne, a môže obsahovať volanie *promise*, na ktorý treba čakať. Kľúčové slovo *await* označuje miesto volania asynchrónneho kódu. Pri použití *await* sa vykonávanie funkcie pozastaví a čaká sa na dokončenie *promise*, ku ktorému patrí. Podrobnejšie sú popísané na stránkach *Mozilla Developer Network* [17, 18].

Web Workers JavaScript je jednovláknový jazyk. Niektoré prehliadače však poskytujú nástroj (*Worker*), ako vykonávať úlohy vo viacerých vláknach paralelne (*task paralelism*). Tento nástroj sa v praxi používa na vykonávanie ťažkých matematických operácií, prácu a triedenie veľkých dát alebo spravovanie veľkého množstva komunikácie.

Viac o asynchrónnom programovaní sa možno dočítať v knihe [13, Async & Performance].

2.2.9 Syntaktické pomôcky

Rozširovací/zvyškový operátor ES6 pridáva nový operátor "...". Podľa kontextu, v ktorom sa nachádza, môže byť:

- zvyškový (*rest*) operátor - Pri definovaní funkcie nám umožňuje zadať ľubovolný počet argumentov v hlavičke funkcie. Definuje štandardne vymenované, plus tie zvyšné. Tie zvyšné potom umiestni do poľa. Sprehľadňuje definovanie funkcií a vyhýba sa riešeniu *arguments* z predchádzajúcich verzií JavaScriptu.
- rozširovací (*spread*) operátor - Umožňuje iterovateľný objekt rozbaliť a nakopírovať do jednotlivých položiek tohoto objektu. Nahrádza napríklad funkcie *apply* a *concat* na poli prvkov. Tiež umožňuje zavolať funkciu s argumentami z iterovateľného objektu jednoduchým spôsobom.

Názorné použitie tohoto operátora možno vidieť v ukážke 2.7. Informácie boli čerpané z knihy [10, Exploring ES6].

```
1  function foo(x, y, ...z){ // rest
2    console.log(x, " ", y); // 1 2
3    console.log(z); // [3, 4, 5]
4  }
5  foo(1, 2, 3, 4, 5);
6
7  var arg1=[a1, a2], arg2=[a3, a4]; // spread
8  var con = [...arg1, ...arg2]; // [a1, a2, a3, a4]
9  foo([...arg1]); // a1 a2
```

Listing 2.7: rozširovací/zvyškový operátor

2.3 Porovnanie jazykov Dart a ECMAScript® 2016

Spoločné črty programovacích jazykov Dart a ECMAScript® 2016. **TODO**

Podpora jazykov na rôznych systémoch Dostupnosť jazykov Dart a ECMAScript® 2016 je rôzna. Zatiaľ čo jazyk JavaScript vie (takmer) každý prehliadač reprezentovať priamo, jazyk Dart je potrebné v produkcii prekladať do jazyka JavaScript (Pri vývoji aplikácie v Darte je možné použiť špeciálny prehliadač na tento jazyk).

Jazyk JavaScript (ECMAScript® 2016) vie bežať v prehliadači. Správanie prehliadačov sa však môže mierne líšiť. **TODO**gulp, pub serve

2.3.1 Tabuľka porovnania syntaxe

Vlastnosť	Dart	ECMAScript® 2016
definovanie a typy premenných	var, dynamic, (int, boolean, ...)	var, let, const
hodnoty premenných	- number, bool, string, null - objekt (List, Map, ...)	- number, boolean, string, null, undefined, symbol - objekt (Array, Date, RegExp, Error, ...)
funkcie	(){...} aj =>	(){...} aj =>
triedy	má všetky štandardné vlastnosti tried	nemá triedy, dedenie kopírovaním, previazanie viacerých cez *prototype*
objekty	objektovo orientovaný jazyk	objektovo orientovaný jazyk
asynchrónnosť	async, await, Future	Promises, Generátory, (async, await)
this	definované v čase písania kódu	definované v čase vykonávania kódu
knižnice	import, export, part	import, export
správa knižníc	pub	npm
počet voľne dostupných kniznic	450385[1](npm)	2575 (pub)
súbežnosť	jednovláknový (alebo možnosť použiť *isolates*)	jednovláknový (alebo *web workers*, poskytované prehliadačom)
prostredie, v ktorom aplikácia beží	prehliadač dartium, alebo skompilovaný do JS	väčšina bežných prehliadačov vie vykonávať JS priamo
vývojári	Google	Netscape Communications Corporation, Mozilla Foundation, Ecma International
vznik jazyka	2011	1995
podpora testovania	áno (knižnica test, ...)	áno (jest, ...)
podpora pre mobilné aplikácie	áno (flutter)	áno (react-native, ...)

[1]Podľa http://www.modulecounts.com z dňa 9.5.2017

Kapitola 3

Návrhové vzory Flux a Redux

V tejto kapitole si povieme niečo o návrhových vzoroch Flux a Redux, ktoré sú určené na spracovávanie udalostí v aplikácii.

3.1 Flux

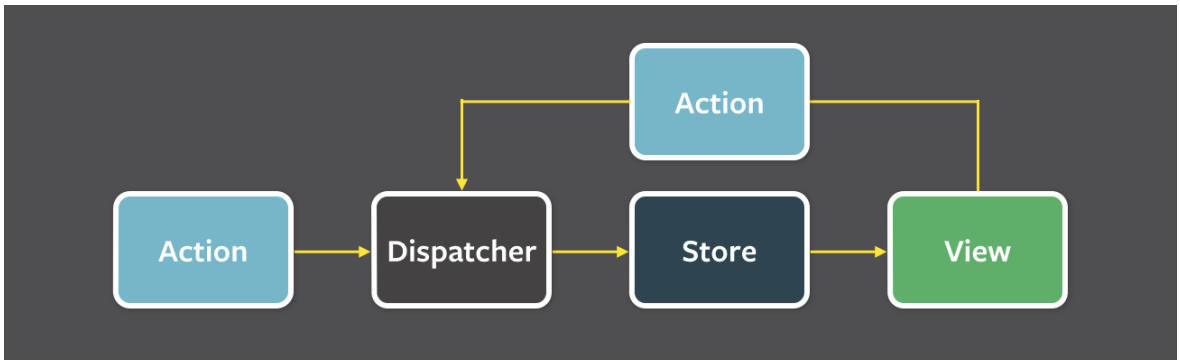
Hlavné črty návrhového vzoru Flux. [5, Overview] **TODO**

Flux je vzor pre spravovanie dát v aplikácii. Najdôležitejším konceptom je tok informácií jedným smerom. Obsahuje štyri základné časti (zobrazené schematicky na obrázku 3.1).

- *Akcie*, ktoré vytvára používateľ, prostredie, kde aplikácia beží alebo aj časti aplikácie.
- *Dispečer* spravuje všetky vytvorené *akcie*.
- *Store* reaguje na *akcie* a spravuje stav aplikácie.
- *View* ktorý vykresľuje stav aplikácie.

3.1.1 Tok dát

1. daný úvodný stav
2. vykreslenie *view* komponentov
3. vznik *akcie*, oznámenie *akcie* dispečeru (funkcia *dispatch*)
4. dispečer upozorní všetky *story*
5. každý *store* spracuje *akciu*, prípadne zmení stav
6. zmena v stave sa vykreslí do komponentov (2. bod)



Obr. 3.1: Flux architektúra [6]

3.1.2 Dispečer (dispatcher)

Dispečer spravuje všetky akcie vykonané v aplikácii. V celej aplikácii by mal byť len jeden. Dispečer obsahuje spätné volanie (callback) na každý store v aplikácii. Keď sa vykoná nová akcia, dispečer pošle túto akciu všetkým storom. Sám nemusí obsahovať akúkoľvek vyššiu logiku, slúži len na distribúciu.

3.1.3 Store

Story obsahujú stav a logiku aplikácie. Store reaguje na akciu, na základe ktorej môže zmeniť stav, ktorý spravuje. Storov môže byť viac a každý upravuje nejakú podčasť dát. Každý store poskytuje dispečerovi na seba callback. Keď sa udeje nejaká akcia, bude o tom upozornený. Na základe typu akcie sa rozhodne, či a ako bude meniť stav aplikácie. (Napríklad ak máme dva story Images a Texts, tak pri vytvorení akcie EditText sa pravdepodobne store Image rozhodne nič nerobiť.) Po zmene stavu vytvorí udalosť, ktorou upozorní view časť, že treba prekresliť údaje.

3.1.4 Akcie

Akcie definujú internú API aplikácie. Zachytávajú možnosti interakcie s aplikáciou. Sú to jednoduché objekty s kľúčom *typ* a voliteľnými pridanými informáciami. Typ akcie by nemal obsahovať žiadne implementačné detaily.

Akcie vytvára view časť (napríklad keď reagujeme na stlačenie tlačidla), server (napríklad chybová hláška počas komunikácie) alebo aj store (keď odstránime používateľa, chceme odstrániť aj všetky jeho príspevky).

```
1  {
2    type: 'delete-user',
3    userId: '1'
4  }
```

Listing 3.1: Akcia vo Flux architektúre

3.1.5 Views

View je časť návrhu, ktorá vykresľuje stav zo storu. Aby bola táto časť vždy aktuálna, musí daný view komponent počúvať na všetky udalosti od storu, ktoré hovoria o zmene relevantných dát. Ak sa zmení stav, store vytvorí udalosť a view sa prekreslí. Architektúra flux neurčuje, ako má byť tento stav vykreslený.

3.1.6 side effects

TODO

3.2 Redux

Hlavné črty návrhového vzoru Redux.

Redux je popis spracovania udalostí v aplikácii. Dáva do popredia lieárne spracovanie udalosti. Pozostáva zo štyroch hlavných častí (zobrazené na obrázku 3.2):

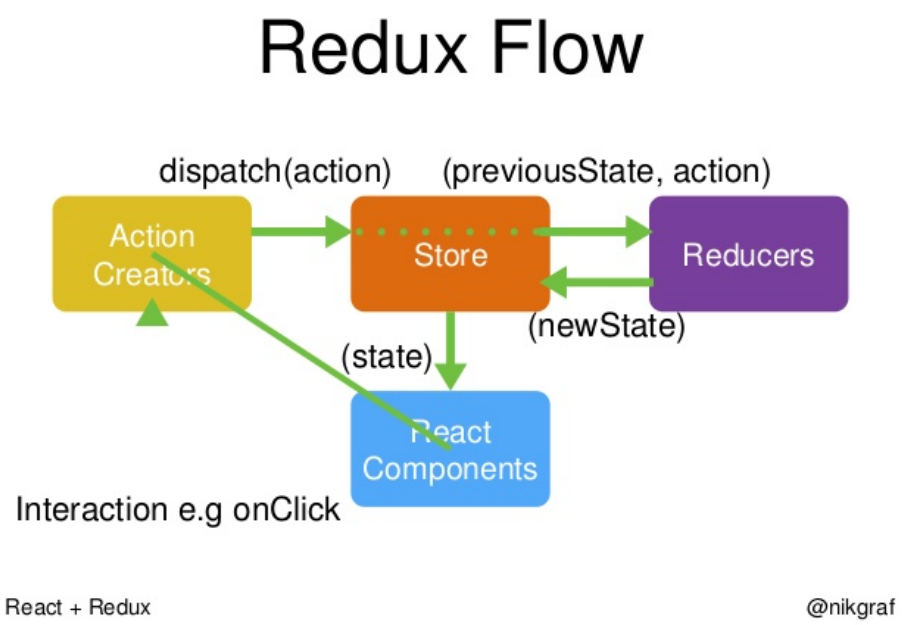
- jeden *store*, ktorý spravuje celý stav aplikácie
- *reducer* - čistá funkcia, ktorá jediná mení stav aplikácie
- *komponenty* vykresľujú aktuálny stav aplikácie
- *akcie*, ktoré definujú vnútorné rozhranie aplikácie

3.2.1 Tok dát

1. daný úvodný stav
2. vykreslenie *komponentov*
3. vznik *akcie*, dispečnutie akcie pre *store*
4. *reducer* na *akcii* a aktuálnom stave, ktorý vráti nový stav
5. zmena v stave sa vykreslí do *komponentov* (2. bod)

3.2.2 Store

Store, správca stavu, vystupuje ako jediný zdroj pravdy v aplikácii. Všetky dáta, ktoré sú vykreslené, pochádzajú zo stavu. Teda ak poznáme tento stav, veľmi jednoducho vieme zostrojiť prostredie, v ktorom celá aplikácia beží. Rovnako v prípade chýb vieme oveľa jednoduchšie zistiť, kde chyba nastala. Tento stav je nemenný(immutable). Ak ho chceme zmeniť, musíme vytvoriť novú inštanciu, v ktorej urobíme potrebné zmeny.



Obr. 3.2: Redux architektúra [9]

Store má svoju funkciu *dispatch()* ktorá slúži na vytváranie akcií. Všetky akcie by mali byť spracované cez túto funkciu. Na tieto akcie môže store reagovať zmenou stavu.

3.2.3 Komponenty

Komponenty slúžia na vykreslenie stavu aplikácie pre používateľa. Ponúkajú rozhranie pre používateľa na komunikáciu s programom. Komponenty vieme rozdeliť na stavové (*Container Components*) a prezentačné (*Presentational Components*).

Prezentačné komponenty iba vykresľujú data, ktoré dostanú. Starajú sa o výzor aplikácie. Tieto možno používať na viacerých miestach (nadpis, tabuľka).

Stavové komponenty počúvajú na zmenu stavu. Majú na starosti, ako by mala stránka fungovať. V reduxovej aplikácii poskytujú rozhranie pre vytváranie akcií a majú prístup ku funkcii dispatch.

3.2.4 Akcie

Akcia je akákoľvek udalosť, ktorá sa môže v aplikácii vyskytnúť, od stlačenia tlačidla používateľom, až po chybové hlášky alebo stiahnutie dát zo servera. Na vytvorenie akcie používame funkciu dispatch. Každá akcia musí obsahovať typ a môže voliteľne obsahovať aj nejaké prídavné data. Na základe tejto akcie potom reducer vypočíta nový stav.

3.2.5 Reducer

Takmer všetká logika aplikácie sa deje v reduceroch. Reducery sú jediný objekt, ktorý môže meniť stav aplikácie.

Reducer je čistá funkcia. Má dva argumenty, stav aplikácie a akciu, ktorá sa uskutočnila. Výstupom je nový stav. Vďaka vlastnosti, že nemá žiadne vedľajšie efekty, ju môžeme veľmi ľahko testovať.

Reducer môžeme vyskladať z viacerých menších čistých funkcií, kde každá z nich sa stará len o určitú malú časť stavu. Vďaka tomu zostáva kód prehľadný a jednoduchý. (**TODO** hodia sa sem nejaké kecy o funkcionálnom programovaní?) Pri písaní reduceru nesmieme zabúdať na to, že nový stav, ktorý vrátime, nesmie byť „starý prerobený“ ale musíme ho prekopírovať a dáta zmeniť až v novej inštancii.

Given the same arguments, it should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.(**TODO**, aj Date.now() má side effects!)

3.2.6 Immutable štruktúry

Na to, aby sme neporušili myšlienku reducera, teda že to má byť čistá funkcia, tak nesmieme zmeniť vstupné parametre. Preto je vhodné použiť nemenné (*immutable*) štruktúry. Pre väčšinu jazykov existuje natívna podpora alebo knižnica pre takéto štruktúry. Ďalšou alternatívou je striktne dodržiavať zásadu nemeniť existujúce dáta a pri zmene vrátiť novú štruktúru s aktuálnymi zmenami. Pri primitívnych typoch (string, number, boolean) toto platí, pri vyšších objektoch si to však musíme skontrolovať sami.

3.2.7 Middlewares

Niekedy treba robiť aj akcie, ktoré nevieme robiť lineárne, nemôžeme robiť lineárne, alebo na ne len nechceme čakať. Príkladom je dopyt na server, kedy čas príchodu odpovede nezávisí úplne od nášho programu. Vtedy môžeme použiť...**TODO**redux-observable vs. redux-saga

3.3 Porovnanie vzorov Flux a Redux

Popis spracovania udalostí v oboch vzoroch a ich vzájomné porovnanie. Vymenovanie a zhrnutie spoločných znakov a rozdielnych.

Historicky prvý bol Flux od Facebooku. Vznikol ako náhrada modelu MVC, kde vo Fluxe je snaha lineárne spracovávať všetky zmeny stavu, čím sa stáva aplikácia omnoho prehľadnejšou. Po vykonaní akcie sa všetky story dozvedia o akcii a príslušné z nich na ňu reagujú.

Redux vznikol modifikáciou fluxu, teda mohli by sme povedať, že je to špeciálny typ fluxu. Kým flux hovorí o spracovaní akcie ako takej, Redux prichádza s myšlienkou, ako meniť stav a to použitím reduceru - čistej funkcie. Použitie reducera spôsobuje, že zmena predchádzajúceho stavu na nasledujúci je plne kontrolovaná dvoma vstupnými parametrami reducera (pôvodný stav a akcia) a pri rovnakom vstupe je výstup vždy rovnaký. Vďaka tomu sa ešte viac uľahčuje testovanie prechodov medzi stavmi.

View View časť je v oboch návrhoch rovnaká. V oboch je to sada komponentov, ktoré zobrazujú statické dáta a je im poskytnutá schopnosť vytvárať nové akcie. Počúvajú na zmenu dát a pri zmene sa prekreslia.

TODO virtual DOM

Akcie Akcie sú v oboch návrhoch rovnaké. Každá akcia je objektom, ktorý obsahuje typ a voliteľne prídavné informácie.

Dispatcher Vo Fluxe je objekt dispatcher, ktorý je jediný a cez neho idú všetky akcie. V Reduxe môže byť tento objekt vynechaný, pretože akcie sa pridávajú storu, ktorý je jediný a ten prevezme zodpovednosť za lineárne spracovanie akcií.

Store Vo Fluxe máme jeden alebo viacero storov, v ktorých sú každý zodpovedný za nejakú logickú podčasť stavu. Každý store je upozornený o každej akcii, ktorá nastane. Na rozdiel od toho Redux obsahuje práve jeden store, ktorý je zodpovedný za celý stav.

Reducer V Reduxe tvorí jednu z hlavných častí reducer. Je zodpovedný za zmenu dát. Vo Fluxe by sme ekvivalent našli v storoch, ktoré menia dáta na základe akcií. Rozdiel je, že od reducera vyžadujeme, aby bol čistá funkcia, teda aby nezávisel na žiadnych hodnotách iných ako sú vstupné parametre funkcie a nemal vedľajšie efekty. Pri storoch sme toto nevyžadovali, keďže story potrebujú napríklad robiť dotazy na server. V Reduxe na side effects slúžia Middlewares.

Reducer sa pri väčších aplikáciách zvykne rozdeliť na viacero menších reducerov, kde každý z nich spravuje celé dáta, alebo nejakú podčasť dát (napríklad ak máme dáta uložené v stromovej štruktúre, reducer môže pôsobiť na podstrome z týchto dát).

3.3.1 Simulovanie behu programu

V redux aplikácii celý stav závisí iba od úvodného stavu a všetkých akcií, ktoré sa vytvorili v danej aplikácii. Teda ak máme poradie akcií, vieme celý postup zrekonštruovať. Toto je veľmi príjemná vlastnosť pri odlaďovaní aplikácie ako aj pri hľadaní chýb.

3.4 Postrehy

TODO V reduxe je funkcia reducer čistá funkcia. Preto vždy pri písaní funkcií sme sa sústredili na písanie takýchto funkcií, ktoré sa neskôr lepšie skladajú. Heslo jedna funkcia má robiť jednu vec sme preferovali pre lepšiu prehľadnosť a čitateľnosť kódu.

Pri mixinoch sme si spomínali, že tieto objekty sme premenili na čisté funkcie. Toto nám veľmi vyhovuje v prípade reduxového návrhu.

3.5 Knižnice open source

TODO dat do vlastnej kapitoly?

Este Celú aplikáciu sme začali vyvýjať v prostredí [4, este]. Snaha držať sa agilného prístupu vývoja aplikácie nás nasmerovala na využitie čo najviac už existujúceho kódu.

React Na vykreslenie komponentov sme pri vývoji použili knižnicu React. Jej veľkou výhodou je, že je rozšírená medzi programátormi a existuje pre ňu mnoho ďalších kompatibilných knižníc. Tiež veľmi pekne spolupracuje s našim návrhovým vzorom *Redux*, keďže React-ové komponenty majú úlohu iba dáta vykresliť.

Komponenty material dizajnu **TODO**

react-toolbox

material-ui - onTouchTap - vhodné pre natívne aplikácie a pre mobilné aplikácie - my vyvíjame aplikáciu aj pre prehliadač

Router O niečo zložitejšie je routovanie a správa url v aplikácii. Existuje viacero možností, ako riešiť routovanie.

Na túto funkciu sme využili knižnicu react-router. Jej výhodou je, že routovanie z komponentov je veľmi jednoduché a prirodzené. Čo mne osobne chýbalo, bola málo popísaná možnosť meniť adresu mimo komponentov. Túto vlastnosť by sme veľmi ocenili najmä kôli ideológii Redux-u, keďže tu by mal byť jediným zdojom pravdy práve stav aplikácie v stave. Po použití tejto knižnice máme zdroje pravdy aspoň dva, jeden pre dáta aplikácie a druhý pre adresu url.

Trošku „krajšie“ v zmysle redux-ovej logiky by bolo riešenie s použitím knižnice router-5, ktorá rieši celý routing na základe dát v stave, kam si ukladá informácie o aktuálnej adrese (aj predchádzajúcich).

[3, Redux]

TODO ktoré z nasledujúcich ešte spomenúť?

gulp, intl, material-ui, normalizr, react, react-native, react-router, redux, webpack

Kapitola 4

Aplikácia

Téma na túto prácu vznikla z praxe.

TODO v práci som sa nesústredila na prihlasovanie do systému...

4.1 Prečo

Zmena programovacieho jazyka Existujúca aplikácia vyvíjaná v jazyku Dart už má svoje zabehnuté prostredie v ktorom beží. Dart je výborný jazyk pre začínajúci tím (má voliteľnú kontrolu typov, pomerne presnú štruktúru.) Už však nie je taký živý a knižnice ku nemu vznikajú pomaly a v obmedzenom množstve. Taktiež podpora vývojových prostredí je menšia a slabšia. (V čase, keď bežal jazyk Dart naplno, existovalo preň vývojové prostredie, dnes existuje viacero programov s podporou Dart-u.) Ďalším dôvodom bol fakt, že JavaScript beží v prehliadači priamo, zatiaľ čo Dart bolo potrebné v produkcii kompilovať do JavaScript-u, keďže nie je podporovaný všetkými prehliadačmi.

Taktiež bolo potrebné písať si vlastné knižnice na zobrazovanie komponentov a ich štýlovanie.

Preto bola snaha presunúť už existujúci kód do jazyka JavaScript, ktorý je v súčastnosti veľmi živý jazyk podporovaný mnohými komunitami. Pri vývoji aplikácie v JavaScripte sme sa zamerali na lean vývoj (/fail fast) preto sme si ako základ aplikácie zvolili balík knižníc [4, este].

Napriek tomu tieto jazyky sú navzájom veľmi podobné a bolo jednoduché prispôsobiť sa jazyku JavaScript po skúsenostiach s Dartom. Rozdiel bol viac v použitých knižniciach ako v samotných jazykoch.

Pri práci s knižnicou *react* sme zaznamenali nevýhodu, keď nebolo možné používať syntax pre jeden komponent na pole viacerých komponentov, čo sme vyriešili obalením poľa pomocným komponentom. Tiež by sa tento problém dal riešiť použitím funkcie namiesto syntaxe pre komponent.

```
1   get buttons => [
2     proforma,
3     packed,
4     cancelOrder,
5     paid,
6   ];
```

Listing 4.1: Pole komponentov v Darte

```
1  const Buttons = order => (
2    <Box>
3      <Proforma props={{ order }} />
4      <Packed props={{ order }} />
5      <CancelOrder props={{ order }} />
6      <Paid props={{ order }} />
7    </Box>
8  );
```

Listing 4.2: Pole komponentov v JavaScripte s použitím knižnice Redux

Jazykové mutácie Veľký prínos sme zaznamenali v spravovaní jazykových mutácií aplikácie. Knižnica *react-intl* vie vyexportovať nepreložené a nové frázy, aj keď nie sú uložené v jednom súbore, čo sprehľadňuje správu týchto fráz.

Zmena návrhového vzoru Flux je pomerne jednoduchý návrhový vzor. Výhodou Reduxu oproti fluxu je však ešte viac prehladná aplikácia. Pri Reduxe je jasné, čo a kedy sa zmenilo. Redux vďaka čistým funkciám a zoznamu vykonaných akcií poskytuje možnosť simulovať celý beh v ľubovolnom čase, zopakovať niektoré kroky s rovnakým stavom aplikácie alebo sa vrátiť v čase dozadu.

V pôvodnej aplikácii sme robili dotazy na server z jednotlivých storov. V novej aplikácii sme toto nemohli nechať na reducer, pretože by to porušilo princípy Redux aplikácie. Preto tieto dotazy robí middleware. Pri písaní novej aplikácie ma tento vzor naučil zamýšľať sa nad tým, ktorá akcia je príčinou takéhoto dotazu na server, pretože bolo potrebné počúvať práve na ňu.

Entity V aplikácii sme spravili jednu zásadnú zmenu so štruktúrou vnútorného stavu aplikácie. Doteraz boli entity (objednávky, protokoly) priamo v stave pod daným kľúčom. V Reduxovej aplikácii sme ich presunuli do samostatnej vetvy, kde sú jednotlivé entity uložené pod typom a identifikačným reťazcom. Potom v stave namiesto zoznamu celých entít existuje len zoznam ID. Túto zmenu sme spravili preto, aby bolo jednoduchšie upravovať jednotlivé entity. Teda aj keď sa na danú entitu odkazujeme na viacerých miestach, jej dáta upravujeme len na jednom. Túto funkcionalitu nám pomáha udržiavať knižnica *normalizr*. Tiež pri veľkom množstve entít to uľahčilo vyhľadávanie danej entity podľa ID (netreba iterovať cez pole entít, stačí sa pozrieť do mapy pod správnym kľúčom).

4.2 Ako preložiť všetky časti kódu

TODO

4.2.1 Komponenty

Komponenty v Darte s Flux-om V Dartovej applikácií sú všetky komponenty triedami.

- Každý základný komponent má nejaké dáta (props), ktoré vykresľuje. Prekreslí sa, keď sa tieto dáta zmenia.
- Komponent rozširuje LocalizedComponent, ktorý spravuje jazyk aplikácie a umožňuje pridávať texty na stránku v správnej jazykovej mutácii.
- Nad ním je InjectComponent, ktorý navyše obsahuje manipulátor a vie, na akej route sa nachádza. Vďaka tomu vie zmeniť routu (cez funkciu url()). Manipulátor vie, či beží v prehliadači, alebo na mobile.
- Najvyššie v hierarchii všeobecných komponentov je FullComponent. Tento komponent má objekt dispatcher. Na tomto objekte môže priamo vytvoriť akciu (funkciou *dispatch*, *dispatchAsync*, alebo *dispatchError*). Táto funkcionalita je zabezpečená pomocou mixinu DispatcherUtils. Tiež má logger s vlastným menom. (?)

Komponenty v JavaScripte s Redux-om V JavaScriptovej aplikácii máme dva typy komponentov: stavové a prezenčné.

Prezenčné sú jednoduchšie, priamočiarejšie. Všetky parametre, ktoré potrebujú, by mali dostať od volaného komponentu (vrátane funkcií na vytváranie akcií). Nemajú prístup ku celému stavu aplikácie. Mali by to byť hlavne znovu použiteľné komponenty (napríklad riadok tabuľky).

Stavové komponenty sú o niečo zložitejšie. Ako samotný názov napovedá, majú prístup ku celému stavu aplikácie, z ktorého si vyberajú len potrebné časti. Na to nám pomáha funkcia *connect* z knižnice *react-redux*. Stavový komponent vyzerá podobne ako prezenčný, tiež všetko vykresľuje len zo svojich parametrov. Funkcia *connect* tvorí medzivrstvu medzi týmto komponentom a jeho exportovaným variantom. Do tohoto komponentu doplní potrebné dáta zo stavu. Tieto komponenty sú však menej univerzálne, keďže sú viazané na konkrétne dáta zo stavu.

Connect Funkcia *connect* akceptuje dva argumenty. Jedným z nich je funkcia, ktorá namapuje stav na argumenty komponentu. Druhý je funkcia, ktorá namapuje dispatcher na akcie. (*Connect* môžeme použiť aj na zmenu vlastných parametrov pre

prezenčné komponenty, avšak to väčšinou nie je žiadúce, pretože to komplikuje logiku kódu. Do prezenčných komponentov by sme mali posúvať už len hotové dáta na vykreslenie. Predídeme tak aj zbytočnému prekresľovaniu kódu kôli zmeneným vlastným dátam.)

Zmeny Stále platí, že komponenty nesmú meniť stav, jediný spôsob, ako by mali mať možnosť toto uskutočniť je vytvorenie akcie.

Komponent v JavaScripte v Reduxe (Reacte) definujeme ako funkciu s voliteľnými pomenovanými parametrami (v kučeravých zátvorkách). Komponent môže vrátit len jeden objekt, nie pole objektov. Tento komponent však môže obsahovať viacero komponentov na jednej úrovni. Ak definujeme tomuto objektu okrem parametrov aj nejaký obsah, je dostupný cez parameter children.

V ukážke 4.5 je komponent v jazyku Dart a v ukážke 4.6 je ten istý komponent v jazyku ECMAScript® 2016.

- Komponent: Namiesto vytvorenia inštancie triedy vytvoríme reactový komponent.
- LocalizedComponent: Jazykové mutácie v JavaScriptovej aplikácii vyriešime použitím vhodnej knižnice, v našom prípade *react-intl*.

Tieto dva typy komponentov sú väčšinou prezenčné komponenty.

- InjectComponent: Zmenu routy riešime pomocou knižnice *react-router*. Idea tejto knižnice je založená na komponente *Link*, ktorý vytvorí akciu na zmenu routy. Úlohou tohoto komponentu je aj zistiť, na akej route sa nachádza. Pri vykreslení aplikácie komponent *Route* rozhoduje, ktorý komponent bude vykreslený. V čase, keď ho zavolá, mu do propsov dá aj routu na ktorej sa nachádza. Z nej potom možno funkciou *connect* zistiť potrebné informácie o aktuálnej route. Tento komponent môže byť tiež prezenčný, ale už tu niekedy môžeme potrebovať funkciu *connect* najmä ak tento komponent vykresľuje Route komponent.
- FullComponent: Tento komponent má možnosť priamo vytvárať akcie. Ku nemu je analogický stavový komponent. V druhom argumente funkcie *connect* namapujeme dispatcher na akcie, ktoré je potrebné v tomto komponente vytvárať.

Gettery V Dartovom kóde sme častokrát využívali gettery 2.1.3 (getter v čase zavolania vyhodnotí výraz, ktorý reprezentuje). Môžeme ku getterom pristúpiť dvoma spôsobmi: buď ich definujeme tieto objekty ako funkcie mimo komponentu, alebo priamo v komponente vyberieme z atribútov potrebné data - spravíme z nich premenné/výrazy. V ukážke 4.6 z aplikácie je použitý prvý prístup. Tento drží funkcie malé a zachováva pôvodnú štruktúru kódu.

Mixiny V Dartovom kóde sme časti kódu, ktoré sa opakovali, dali do mixinov 2.1.1. V JavaScripte s Reduxom však chceme preferovať vykreslenie čistými funkciami, čo podporuje aj ideológia knižnice *react*. Preto sme z mixinov spravili malé pomocné knižnice s čistými funkciami, ktoré len jednoducho importujeme. Často tieto mixiny využívali práve dáta danej triedy (komponentu) pomocou getterov. Toto ale pri knižniciach nie je možné, keďže komponent nezdieľa svoje dáta s knižnicami, ktoré používa.

Postrehy pri písaní TODO *Mixiny* som nahradila čistými funkciami, ktoré som dala do samostatného súboru a vytvorila tak malú knižnicu funkcií. Nevýhodou pri používaní funkcií namiesto mixinov je nemožnosť využiť gettery. Všetky hodnoty, ktoré chceme použiť vrámci jednej funkcie musíme mať ako vstupné parametre funkcie.

Perzistentné štruktúry Perzistentné štruktúry sme v jazyku Dart používali s pomocnou knižnicou *vacuum_persistent*. V jazyku ECMAScript® 2016 sú viaceré možnosti, jednou je používať knižnicu na takéto štruktúry, ďalšou je kopírovať existujúce štruktúry pri zmene. Tá druhá možnosť je náročnejšia na kontrolu, ale programátorsky príjemná s rozširovacím operátorom 2.2.9 (od verzie ES6). Pre jednoduchosť kódu sme zvolili túto možnosť cez rozširovací operátor.

TODO DOM komponenty - rieši react-DOM, predtým sme museli mať na to (vlastnú ?) knižnicu

organizácia priečinkov: všeobecné veci (+ natívne) a také ktoré manipulujú stav v common, tie, ktoré bežia iba v prehliadači, tie sú v browser. Máme snahu robiť aj natívnu aplikáciu preto chceme čo najmenej kódu v browser a čo najviac v common.

4.2.2 Story

Story v pôvodnej aplikácii spracúvajú všetky akcie. Každý store má zoznam akcií, na ktoré počúva. Okrem toho počúva aj na zmenu routy.

Úlohy store vo Fluxe:

- uchováva a mení vnútorný stav aplikácie(insert, insertInOrCreate, ...) => REDUCER
- komunikuje so serverom (dispatchAsync) => MIDDLEWARE
- mení routu (*dispatchRoute*, dispatch(GOROUTE)). => KOMPONENT
- vytvára nové akcie !!! PROblÉM !!! => poradie volania reducerov v *configureReducer.js*
- Loguje zmeny. => asi knižnica nejaká :) (napr redux-logger)

počúvanie na zmenu routy

4.2.3 Akcie

Každá akcia má typ a nejaké prídavné dáta. Typ akcie je string, dáta: predtým niekoľko polí, teraz jeden objekt *payload*. Ľahká kompozícia aj dekompozícia dát.

action creators

Na akcie počúvajú story aj middlewares.

4.2.4 ďalšie:

...asi do kapitoly kniznice???

- routovanie - rozhodovanie, aku stranku vykreslim (routeConfig.js) - získavanie aktuálnej routy napr: ownProps.params.id

- jazykové mutácie: <FormattedMessage ...messages.ForPackaging />

4.2.5 Ukážky kódu

```
1  //zmena v stave - store
2    listen({
3      ...
4      "$ORDER-$PACKED": _markOrderAs(ORDERPACKED),
5      ...
6    });
7
8    ...
9
10   _markOrderAs(String state) => (event) {
11     insert([ORDER, STATUS], state);
12
13     dispatch("$RETAILER-$ORDER-$EDIT-$STATUS",
14         data: getInOr(event, DATA, per({}))
15     );
16   };
17
18 //zmena poslana na server - store
19   listen({
20     ...
21     "$RETAILER-$ORDER-$EDIT-$STATUS": _updateOrderStatus,
22     ...
23   });
24
25   ...
26
27   _updateOrderStatus(event) {
28
29     var eventData = getMap(event, DATA);
30
31     eventData = insertInOrCreate(eventData, STATUS, getInOr(data, [ORDER, STATUS]));
32
33     String id = getInOr(data, [ORDER, ID]);
34
35     DiscoveryMap aditionalData =  new DiscoveryMap.fromJson(unpersist(eventData));
36
```

```
37      insert([ORDER], into(getInOr(data,[ORDER]), eventData));
38
39      dispatchAsync("$RETAILER-$ORDER-$UPDATED", resource.patch(aditionalData, id));
40
41    }
```

Listing 4.3: Dotaz na server vo Flux-e v store

```
1  //zmena v stave - reducer
2    switch (action.type) {
3      ...
4      case "CHANGE_ENTITY_FIELD": {
5        var {entity, id, field, value} = action.payload;
6
7        var schema = schemas(entity);
8
9        var newField = {}
10       newField[field] = value;
11
12       var oldEntity = denormalize(id, schema, state);
13       var newEntity = {...oldEntity, ...newField};
14       var normalized = normalize(newEntity, schema);
15
16       var normalizedEntities = {...state[entity], [id]: normalized.entities[entity][id
             ]};
17
18       return {...state, [entity]: normalizedEntities};
19     }
20
21     default:
22       return state;
23
24   }
25
26 //zmena poslana na server - middleware
27 const entityChangedEpic = (action$: any, { firebase }: Deps) =>
28   action$
29     .filter((action: Action) => action.type === "CHANGE_ENTITY_FIELD")
30     .mergeMap(action => {
31       var { entity, id, field, value } = action.payload;
32
33       var newField = {};
34       newField[field] = value;
35       var source = `${pl(entity)}/${id}.json`
36       const promise = fetch(serverUrl + source, {
37         method: "PATCH",
38         headers: new Headers({
39           "X-Requested-With": "eusahub-redux-app",
40           "X-USER-TOKEN": token,
41           "Content-Type": "application/json"
42         }),
43         body: JSON.stringify(newField)
44       });
45
46       return Observable.from(promise.then(response => response.json()))
47         .mergeMap(response =>
48           Observable.from(entityPatched(entity, response)))
49         .catch(error => Observable.of(appError(error)));
50     });
```

Listing 4.4: Dotaz na server v Redux-e cez middleware

```
library eusahub.components.retailer.pack_order;

import 'package:flux/component.dart';
import 'package:tiles/tiles.dart' as tiles;
import 'package:tiles/src/dom/dom_attributes.dart' as tiles;
import 'package:EusahubBrowser/constants.dart';
import 'package:EusahubBrowser/utils.dart';

import 'package:EusahubBrowser/remark.dart';
import 'package:EusahubBrowser/src/utils/interfaces/full_component.dart';

import 'order.dart';

class PackOrder extends FullComponent
    with Status, Transports, Tab {
  PackOrder(Props props) : super(props);

  render() =>
      row([
        col(attributes, md: 9),
        col(actions, md: 3),
      ]);

  get attributes =>
      panel(
          content,
          currentUrl: currentUrl,
          panelHeader: l("Order {{slug}}", placeholders: {SLUG: getInOr(data, SLUG, ""
              )}));

  get content {
    if (showParcels) {
      return parcels;
    } else if (showPickup) {
      return pickup;
    } else if (showCustom) {
      return custom;
    } else {
      return null;
    }
  }

  get actions => orderActions(props: cp(data, name: ACTIONS));

  get parcels => orderParcels(props: cp(data, name: PARCELS));

  get pickup => alert(l("This is pickup order"));

  get custom => alert(l("This is order with a custom transport"));

  bool get showParcels =>
      showParcelsStatuses.contains(status) && transportsContainsDPD;

  bool get showPickup =>
      showParcelsStatuses.contains(status) && transportsContainsPickup;
```

```
55
56    bool get showCustom =>
57      showParcelsStatuses.contains(status) && transportsContainsCustom;
58
59    final List showParcelsStatuses = [ORDERPACKED, ORDERAWAITING_FULFILLMENT];
60
61  }
62
63  tiles.ComponentDescriptionFactory packOrder = tiles.registerComponent((
64      {props, children}) => new PackOrder(props));
```

Listing 4.5: Komponent v Darte s Flux-om

```
1   import React from "react";
2   import { compose } from "ramda";
3   import { connect } from "react-redux";
4   import { FormattedMessage, defineMessages } from "react-intl";
5   import { denormalize } from "normalizr";
6
7   import linksMessages from "../../common/app/linksMessages";
8   import { Box, PageHeader, Text } from "../../common/components";
9   import { Title } from "../components";
10  import * as c from "../../common/app/constants";
11  import { order as orderSchema } from "../../common/app/schemas";
12  import { changeEntityField, appShowDialog } from "../../common/app/actions";
13
14  import {
15    transportsContainsDPD,
16    transportsContainsPickup,
17    transportsContainsCustom
18  } from "./helpers/transports";
19  import OrderParcels from "./parcels/Parcels";
20  import Actions from "./actions/Actions";
21
22  const messages = defineMessages({
23    pickupOrder: {
24      defaultMessage: "This is pickup order.",
25      id: "app.content.retailer.order.This is pickup order"
26    },
27    customTransportOrder: {
28      defaultMessage: "This is order with a custom transport.",
29      id: "app.content.retailer.order.This is order with a custom transport"
30    }
31  });
32
33  const OrderPage = (
34    {
35      orderId = false,
36      order = false,
37      path,
38      changeEntityField,
39      appShowDialog
40    }
41  ) => (
42    <Box>
43      <Title message={linksMessages.for_packaging} />
44      <Attributes order={order} path={path} />
45      <Actions
46        order={order}
```

```
47        changeEntityField={changeEntityField}
48        appShowDialog={appShowDialog}
49      />
50    </Box>
51  );
52
53  const Attributes = ({ order, path }) => (
54    <Box>
55      <PageHeader
56        heading={
57          <FormattedMessage
58            id="Order {slug}"
59            defaultMessage={`Order {slug}`}
60            values={{ slug: order.slug }}
61          />
62        }
63        description={`home${path}`}
64      />
65      <Content order={order} />
66    </Box>
67  );
68
69  const Content = ({ order }) => {
70    if (showParcels(order)) {
71      return <Parcels order={order} />;
72    } else if (showPickup(order)) {
73      return <Pickup />;
74    } else if (showCustom(order)) {
75      return <Custom />;
76    } else {
77      return null;
78    }
79  };
80
81  const Parcels = ({ order }) => <OrderParcels order={order} />;
82
83  const Pickup = () => (
84    <Text><FormattedMessage {...messages.pickupOrder} /></Text>
85  );
86  const Custom = () => (
87    <Text><FormattedMessage {...messages.customTransportOrder} /></Text>
88  );
89
90  const showParcels = order =>
91    showParcelsStatuses.includes(order.status) &&
92    transportsContainsDPD(order.transports);
93
94  const showPickup = order =>
95    showParcelsStatuses.includes(order.status) &&
96    transportsContainsPickup(order.transports);
97
98  const showCustom = order =>
99    showParcelsStatuses.includes(order.status) &&
100   transportsContainsCustom(order.transports);
101
102 const showParcelsStatuses = [c.ORDERPACKED, c.ORDERAWAITING_FULFILLMENT];
103
104 function denormalizeOrder(state, orderId, ownProps) {
105   return denormalize(orderId, orderSchema, state.entities);
```

```
106 }
107
108 export default compose(
109   connect(
110     (state, ownProps) => ({
111       orderId: ownProps.params.id,
112       order: denormalizeOrder(state, ownProps.params.id),
113       path: ownProps.location.pathname
114     }),
115     {
116       changeEntityField,
117       appShowDialog
118     }
119   )
120 )(OrderPage);
```

Listing 4.6: Komponent v ECMAScript® 2016s Redux-om

Záver

V závere mojej bakalárskej práce zhrniem, aké bolo zadanie, ako som postupovala a ako by sa dalo na moju prácu nadviazať.

Literatúra

[1] Ecma International 2015. *Dart Programming Language Specification, Version 1.11.* 4th edition, 2015. [Citované 2016-12-4] Dostupné z `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-408.pdf`.

[2] 2016 Ecma International. ECMAScript 2016 language specification, 2016. [Citované 2016-12-5] Dostupné z `http://www.ecma-international.org/ecma-262/7.0/#`.

[3] Dan Abramov. Getting started with Redux, 2016. [Citované 2016-12-4] Dostupné z `https://egghead.io/courses/getting-started-with-redux`.

[4] Daniel Steigerwald and the community. Starter kit Este for universal full–fledged react apps., 2016. [Citované 2016-12-4] Dostupné z `https://github.com/este/este`.

[5] Facebook Inc. Overview of Flux, 2015. [Citované 2016-12-4] Dostupné z `https://facebook.github.io/flux/docs/overview.html#content`.

[6] Facebook Inc. Structure and data flow (flux), 2015. [Citované 2017-5-10] Dostupné z `https://facebook.github.io/flux/docs/in-depth-overview.html#content`.

[7] Google. A tour of the dart language, 2017. [Citované 2017-5-10] Dostupné z `https://www.dartlang.org/guides/language/language-tour`.

[8] Michael S Mikowski and Josh C Powell. Single page web applications. *B and W*, 2013. [Citované 2017-01-23] Dostupné z `http://deals.manningpublications.com/spa.pdf`.

[9] Nikolaus Graf. React + redux introduction, 2015. [Citované 2017-5-10] Dostupné z `https://www.slideshare.net/nikgraf/react-redux-introduction`.

[10] Axel Rauschmayer. Exploring es6, 2015 - 2017. [Citované 2017-5-10] Dostupné z `http://exploringjs.com/es6.html`.

[11] Kyle Simpson. *You Don't Know JS: Scope & Closures.* O'Reilly Media, Inc., 2014.

[12] Kyle Simpson. *You Don't Know JS: this & Object Prototypes*. O'Reilly Media, Inc., 2014.

[13] Kyle Simpson. *You Don't Know JS: Async & Performance*. O'Reilly Media, Inc., 2015.

[14] Kyle Simpson. *You Don't Know JS: ES6 & Beyond*. O'Reilly Media, Inc., 2015.

[15] Kyle Simpson. *You Don't Know JS: Types & Grammar*. O'Reilly Media, Inc., 2015.

[16] Kyle Simpson. *You Don't Know JS: Up & Going*. O'Reilly Media, Inc., 2015.

[17] © 2005-2017 Mozilla Developer Network and individual contributors. async function, 2017. [Citované 2017-5-10] Dostupné z `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function`.

[18] © 2005-2017 Mozilla Developer Network and individual contributors. await, 2017. [Citované 2017-5-10] Dostupné z `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await`.

[19] © 2005-2017 Mozilla Developer Network and individual contributors. Strict mode, 2017. [Citované 2017-5-10] Dostupné z `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode`.