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: jedno, druhé, tretie (prípadne štvrté, piate)

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 [7].

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. Statická verzia objektu null je $\perp$.

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 m 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.

2.2 ECMAScript® 2016a jeho porovnanie s jazykom Dart

Hlavné črty programovacieho jazyka ECMAScript® 2016.

ECMAScript bol pôvodne navrhnutý ako webový skriptovací jazyk, ktorý upravuje internetové stránky v prehliadači a vykonáva výpočty v prehliadači. Dnes je to plne vybavený všeobecne navrhnutý objektovo orientovaný programovací jazyk. Skriptovací jazyk je programovací jazyk zameraný na výpočty a manipuláciu s objektami už existujúceho systému. **TODO**

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.

A class is instantiated into object form by a copy operation.

Don't let polymorphism confuse you into thinking a child class is linked to its parent class. A child class instead gets a copy of what it needs from the parent class. Class inheritance implies copies.

Functions called with new are often called "constructors", despite the fact that they are not actually instantiating a class as constructors do in traditional class-oriented languages.

Mixin/extends pridáva(kopíruje) špecifickú funkcionalitu z inej "triedy". you can partially emulate the behavior of "multiple inheritance", but there's no direct way to handle collisions if the same method or property is being copied from more than one source **TODO**

Postrehy pri písaní *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.

2.2.2 Typy

Jazyk ECMAScript® 2016nie 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.

- *Primitívne typy* sú string, number, boolean, null a undefined
- *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 vieme definovať priamo, teda vymenovaním obsahu, alebo cez kľúčové *new*. Obsah objektu môžeme plniť viacerými možnosťami. 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 2.1: tvorba objektu

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

Null typ má práve jednu hodnotu, *null*.

This this is not an author-time binding but a runtime binding. this ukazuje na miesto, odkial bola daná funkcia volaná, teda keď sa pozrieme do zásobníka volaní, bude to funkcia, ktorá je hneď pred(pod, nad?) našou

- *default binding* this predstavuje miesto, odkial bola funkcia volaná. V striktnom móde defaultné nastavenie this nefunguje.
- *implicit binding* ak voláme metódu na objekte v danom čase, this ukazuje na daný objekt
- *explicit binding* funkcie call(), apply() kde ako prvý parameter dáme odkaz na *this* ktorý chceme využiť. Podskupinou je *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*
- *new* binding - keď funkciu vytvoríme kľúčovým *new*, vytvorí sa nová inštancia objektu, ktorá má v čase vykonávania this nasmerované na seba

arrow functions sa správajú ako štandardné this ktoré poznáme z iných jazykov. V takejto funkcii this odkazuje na samú seba.

Podrobnejšie informácie možno nájsť v literatúre ...

2.2.3 Premenné

ak je v normálnom, a má danej premennej priradiť hodnotu, vytvorí ju ako globálnu ak je v striktnom móde behu programu, globálnu premennú nevytvorí ale vyhodi ReferenceError TypeError vráti, ak napríklad objekt zavoláme ako funkciu, on nájde daný objekt ale danú operáciu s ním nevie vykonať. ReferenceError vyhodí, ak premennú nenájde a je v striktnom behu programu.

ReferenceError is Scope resolution-failure related, whereas TypeError implies that Scope resolution was successful, but that there was an illegal/impossible action attempted against the result.

Deklarovanie premenných v ECMAScript® 2016 ECMAScript® 2016pod-poruje 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. Narozdiel od týchto, premennú *var* môžeme deklarovať aj viackrát a môžeme sa na ňu pýtať aj mimo bloku kde sme ju deklarovali. V ukážke 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.

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, ...){}*.

```
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
```

Listing 2.2: JavaScript deklarovanie

2.2.4 Upozornenia a chyby

TODOStrict mode - podporovaný rôzne rôznymi prehliadačmi - mení niektoré tiché chyby, tým že spraví throws ...mistakes -> errors- zabraňuje niektorým chybám, aby engine ktorý kompiluje a vykonáva kód, mohol lepšie optimalizovať - 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)

- má dosah na funkciu alebo na celý skript - zapína sa príkazom *'use strict';* na začiatku funkcie/skriptu

2.2.5 Súkromie

?? čo k tomu?? **TODO**

2.2.6 Knižnice

TODOimport, export, npm

vs. pub

Dostupnosť

Dostupnosť jazykov Dart a ECMAScript® 2016je 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).

Pre jazyk ECMAScript® 2016je 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.)

2.2.7 Súbežnosť

TODOasynchrónnosť, niečo ako await?

2.2.8 TODO

[6, introduction] [2, ECMAScript® 2016]

2.3 Spoločné znaky jazykov Dart a ECMAScript® 2016

Spoločné črty programovacích jazykov Dart a ECMAScript® 2016. Pravdepodobne odkaz na tabuľku s podrobnejším popisom vybraných spoločných a rozdielných príkazov.

2.4 Rozdiely a ich prepojenie

Rozdielne časti daných jazykov a návrh riešenia, ako by sa dali nahradiť. Využitie zaujímavých čŕt jazyka ECMAScript® 2016.

npm vs. pub

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]

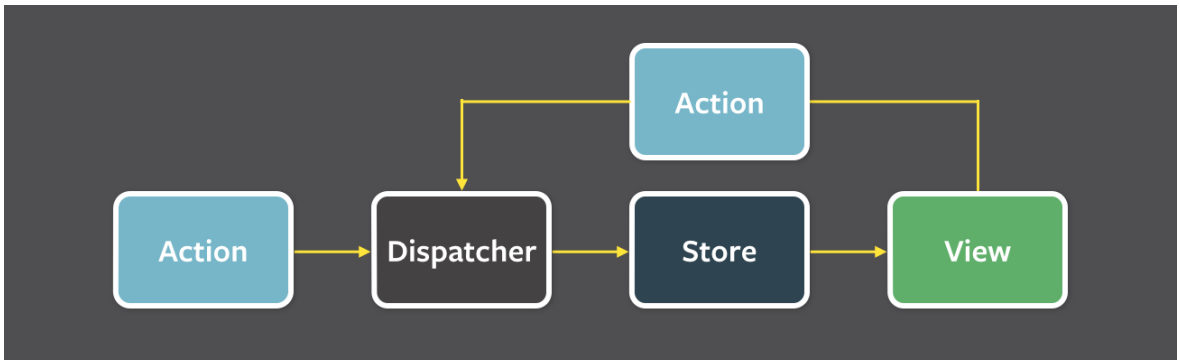
Flux je vzor pre spravovanie dát v aplikácii. Najdôležitejším konceptom je tok informácií jedným smerom. Obsahuje štyri základné časti. *Akcie*, ktoré vytvára používateľ, prostredie, kde aplikácia beží alebo aj časti aplikácie. *Dispečer* spravuje všetky vytvorené akcie. *Store*, ktorý reaguje na akcie a mení data aplikácie a *View*, ktorý data aplikácie vykresľuje.

3.1.1 Tok dát

- daný úvodný stav
- vykreslenie komponentov
- vznik akcie, dispatchnutie akcie pre dispatcher
- dispatcher upozorní všetky story
- každý store spracuje akciu, prípadne zmení data
- zmena v dátach sa vykreslí do komponentov

3.1.2 Dispečer

Dispečer spravuje všetky akcie vykonané v aplikácii. V celej aplikácii by mal byť len jeden. Dispečer obsahuje callback na každý Store v aplikácii. Keď sa vykoná nová



Obr. 3.1: Flux architektúra

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 data, ktoré spravuje. Storov môže byť viac a každý upravuje nejakú podčasť dát. Každý store poskytuje Dispečerovi callback na seba, aby keď sa udeje nejaká akcia, bol o tom upozornený. Na základe typu akcie sa rozhodne, či a ako bude meniť data aplikácie. (Napríklad ak máme dva story Images a Texts, tak pri vytvorení akcie EditText sa pravdepodobne store Image rozhodne nič nerobiť.) Po zmene dát 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 časť je tá, ktorá vykresľuje dáta 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 zmenia dáta, store vytvorí udalosť a view sa prekreslí. Architektúra flux neurčuje, ako majú byť tieto data vykreslené.

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, jeden *store*, v ktorom sú všetky dáta aplikácie uložené, *reducer* - čistú funkciu, ktorá jediná mení akokoľvek stav aplikácie, komponenty, ktoré dané data zo stavu vykresľujú a akcie, ktoré definujú vnútorný interface aplikácie.

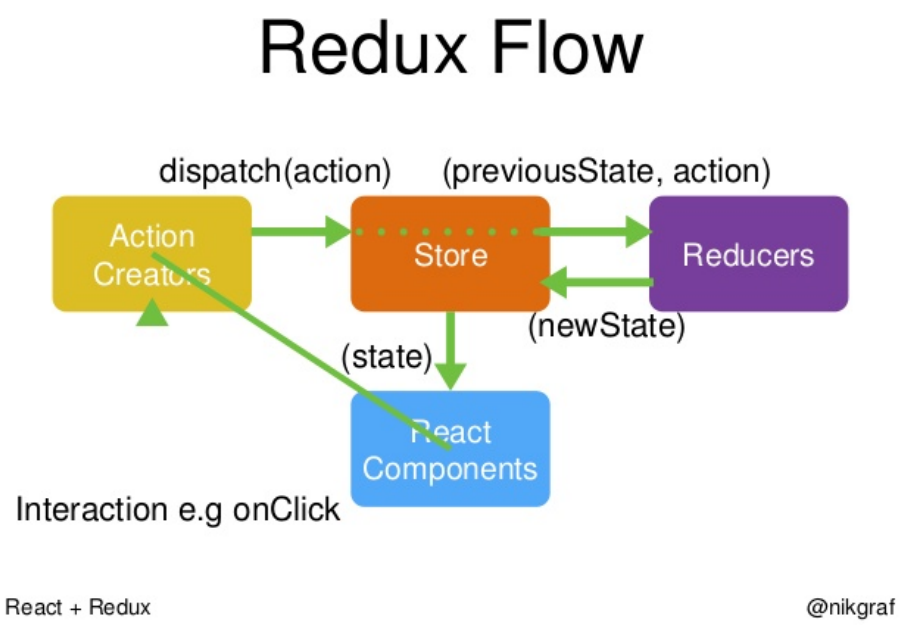
3.2.1 Tok dát

- daný úvodný stav
- vykreslenie komponentov
- vznik akcie, dispatchnutie akcie pre store
- reducer na akcii a aktuálnom stave
- zmena v dátach sa vykreslí do komponentov

3.2.2 Store

Store, správca stavu, vystupuje ako jediný zdroj pravdy v aplikácii. Všetky data, ktoré sú vykreslené, pochádzajú zo stavu. Teda ak poznáme tento stav, veľmi jednoducho vieme potom nasimulovať 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 teda zmeniť, musíme vytvoriť novú inštanciu, v ktorej urobíme potrebné zmeny. Toto je dôležité, aby sme vedeli sledovať beh aplikácie.

Store má svoju funkciu *dispatch()* ktorá slúži na prijímanie podnetov zvonka, aby store zmenil svoj stav.



Obr. 3.2: Redux architektúra

3.2.3 Komponenty

Komponenty slúžia na vykreslenie stavu aplikácie pre užívateľa. Ponúkajú rozhranie pre používateľa na komunikáciu s programom a v prípade reduxovej aplikácie je to práve pomocou vytvárania akcií. Komponenty vieme rozdeliť na "múdreä "hlúpe".Hlúpe komponenty iba vykresľujú data, ktoré dostanú. Tieto sú potom veľmi ľahko znovupoužiteľné. Tie múdre komunikujú spätne s dátami. 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 užívateľom, 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

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 vykonala a výstupom je nový stav. Vďaka tejto vlastnosti, že nemá žiadne "side effects"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.Pri písaní reduceru nesmieme zabúdať na to, že nový stav, ktorý vrátime, nesmie byť "starý prerobenýäle musíme ho prekopírovať a dáta zmeniť až v novej inštancii.

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 väčšínu jazykov existuje podpora alebo knižnica pre takéto štruktúry. Daľšou alternatívou je striktne dodržiavať zásadu nemeniť existujúce dáta a pri zmene vrátiť novú štruktúru s aktuálnymi zmenami. Pri (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**

3.3 Knižnice open source

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

[4, Este starter kit]

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 react-toolbox

material-ui - onTouchTap - vhodné pre natívne aplikácie a pre mobilné aplikácie - my vyvíjame aplikáciu 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 pravdyäspoň 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 aj ukladá informácie o aktuálnej adrese (aj predchádzajúcich).

[3, gettting started]

3.4 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 - kým v MVC toto zabezpečovalo veľa kontrolorov, vo fluxe idú všetky akcie iba cez jeden dispatcher.

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.

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 dát. Každý store je upozornený o každej akcii, ktorá nastane. Na rozdiel od toho Redux obsahuje práve jeden store, ktorý je zodpovedný za celé data.

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 side 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 nejakú syntaktickú podčasť dát(napríklad ak máme dáta uložené v stromovej štruktúre, reducer môže pôsobiť na podstrome z týchto dát).

3.4.1 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.

3.5 Postrehy

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.

3.6 Návrhy migrácie Flux do Redux

Viaceré možné návrhy migrácie s referenciami na existujúce návrhy na internete.

3.7 Riešenie

Jedno vybrané riešenie z vyššie uvedených. Zdôvodnenie a návrh implementácie mnou zvoleného riešenia.

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] David Flanagan. *JavaScript: the definitive guide.* O'Reilly Media, Inc., 2006.

[7] 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`.