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

Diplomová práce

Vizualizace dat s využitím AngularJS

Plzeň, 2016 Petr Kukrál

Prohlášení

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

V Plzni dne 5. května 2016 Petr Kukrál

Poděkování

Rád bych poděkoval panu Ing. Martinu Dostalovi, Ph.D za odborné konzultace, přístup k vedení diplomové práce a za průběžné komentování textu celé práce. Také bych rád poděkoval své přítelkyni Veronice Švecové za pomoc s korekturami této práce.

Abstrakt

Tato práce se věnuje tvorbě webové aplikace pro vizualizaci dat s využitím JavaScriptu. Klientská část je implementována ve frameworku Angular, od společnosti Google, a serverová část je realizována v PHP. Práce porovnává existující možnosti vizualizace dat v oblasti webových technologií a zabývá se srovnáním nejpoužívanějších JavaScriptových frameworků a knihoven Angular, React a jQuery. Tyto technologie jsou porovnávány jak z hlediska náročnosti implementace, tak z pohledu výkonnosti. Práce mimo jiné obsahuje i návod na konverzi pluginu ze starší knihovny jQuery do novějšího Angularu.

Abstract

This thesis is focused on creating web applications for data visualization using JavaScript. The client side is implemented in the Angular framework, by Google. The server side is implemented in PHP. This thesis compares methods of data visualization on the Web and discuss the most popular JavaScript frameworks and libraries like Angular, React and jQuery. These technologies are compared with difficulty and performance. This thesis also includes the instructions for converting plugin from older library jQuery to newer framework Angular.

Obsah

1 Úvod

S rostoucím množstvím dat roste nutnost tato data správně vizualizovat. Díky vizualizaci si můžeme uvědomit souvislosti mezi daty a vizualizace nám slouží k rychlému přehledu o situaci, který bychom z objemných dat určili jen velmi obtížně. V této práci se budeme věnovat vizualizaci dat na webu s použitím moderního frameworku Angular.

Pro vizualizaci můžeme použít řadu zobrazovacích technologií. Vybrat správnou technologii pro vizualizaci je velice důležité. Správný výběr nám může ušetřit spoustu času a práce. Proto si jednotlivé technologie popíšeme v Kapitole 2.5 Vizualizace dat. V každé technologii vytvoříme jeden graf, abychom tyto technologie lépe pochopili a mohli je porovnat přímo na příkladech. Práce nás provede způsoby zobrazování dat:

- HTML canvas
- SVG
- HTML a CSS

Dále se v této práci budeme věnovat použití Angularu [1] pro vizualizaci dat. Angular je Javascriptový framework vyvíjený společností Google. V této práci si ukážeme, jak s jeho pomocí vizualizovat data. Porovnání Angularu s knihovnami React [2] a jQuery [3] je velmi diskutované téma. Proto je zde porovnáme již v konkrétních příkladech vizualizace dat. V porovnání si ukážeme klady a zápory jednotlivých technologií a popíšeme si, v jakých projektech je vhodné uvedené technologie použít.

V jQuery je vytvořeno velké množství pluginů pro zobrazování dat. Abychom tyto pluginy mohli využít i v novější technologii Angular, v Kapitole 2.4.7 Konverze jQuery pluginu do Angularu si ukážeme, jak co nejlépe přepsat jQuery plugin do Angularu.

Nad rámec zadání diplomové práce je možné implementovat plugin pro zobrazování dat v knihovně React, abychom si ji mohli porovnat s Angularem a jQuery. React je knihovna vyvíjená společností Facebook. Je spolu s jQuery často porovnávaná s Angularem. Také je možné ukázat si napojení Reactu do Angularu, které například potřebujeme, když již hotový plugin v Reactu chceme zapojit do naší aplikace.

Cílem práce je vytvořit sadu modulů pro Angular, které umožní základní vizualizaci dat formou grafů. Jednotlivé moduly budou označovány používaným pojmem widget a budou zapouzdřeny modulem "nástěnky", dále označovaném také jako dashboard. Dashboard slouží např. vedení podniku k rychlému přehledu o situaci v systému. Widgety jsou části dashboardu a sleduje se v nich určitá oblast systému. Příkladem widgetu může být denní počet tržeb v podniku.

2 Použité technologie

V této práci používáme velké množství různých druhů technologií. Technologie používáme k odlišným účelům, jako je:

- Správa verzí a balíčků
- Serverová aplikace
- Klientská aplikace a framework Angular
- Vizualizace dat

Tyto technologie zde postupně projdeme a popíšeme. V Angularu je implementována větší část celé práce, proto se mu budeme věnovat ve zvláštní kapitole.

2.1 Systémy pro správu verzí a balíčků

Na první pohled to může vypadat, že správa verzí a správa balíčků jsou podobná témata. Jedná se ale o dvě naprosto odlišné technologie. Správa verzí implementovaného kódu nám umožňuje tvořit stabilní části aplikace, do kterých se můžeme vracet. Správa balíčků slouží k vytvoření a správě závislostí naší aplikace na externích knihovnách.

Nejdříve si rozebereme a popíšeme správu verzí kódu. Celá aplikace je průběžně vytvářena pomocí verzovacího systému Git [4]. Pro hostování celého projektu je použita služba Github. Zdrojové kódy je tak možné online zobrazit na [5].

Git vytváří lokální repozitář (úložiště obsahující verze projektu), který pak můžeme synchronizovat s vnějším repozitářem. Velkou výhodou Gitu je [4], že většina operací se soubory je vytvářena lokálně. K většině operací tak nejsou potřeba informace z jiných počítačů v síti. Git umožňuje vytvářet jednotlivé verze systému pomocí commitů. Commit je stav projektu v daném čase s textovým komentářem. Podle [4] Git používá pro spravované soubory tři základní stavy:

- zapsáno (commited)
- změněno (modified)
- připraveno k zapsání (staged)

Zapsáním bezpečně uložíme data do našeho lokálního repozitáře. Změněno znamená, že v souboru došlo ke změnám, ale ty ještě nebyly zapsány do repozitáře. Tyto změny je možné buď potvrdit (připravit k zapsání v následujícím commitu) nebo zrušit (discard). Systém Git také umožňuje vyvíjení částí aplikace paralelně pomocí větvení [4]. To se nám v případě vývoje jedním vývojářem hodí hlavně v okamžiku, kdy vyvíjíme delší dobu novou funkcionalitu a aplikace může být po dobu vývoje nestabilní. Do hlavní (master) větve je přidáme až po dokončení.

Nyní si popíšeme balíčkovací systémy, které za nás řeší problematiku závislostí. Používání balíčkovacích systémů za nás zajišťuje, že všechny závislosti naší aplikace jsou stále aktuální. Novější verze balíčků mohou obsahovat opravy chyb a optimalizace výkonu, které jsou pro naší aplikaci zásadní. Pokud stahujeme závislosti ručně, nemáme možnost automatické kontroly existence novějších verzí balíčku. V balíčkovacích systémech si navíc můžeme nastavit, kterou verzi balíčku chceme stahovat. Budou se nám tak stahovat například opravy stávající verze.

Pro správu balíčků, v naší práci, používáme dva nástroje npm [6] a Bower [7]. Použití dvou balíčkovacích nástrojů vychází z předpřipravené šablony pro vývoj aplikace v Angularu [8]. Npm [6] slouží, jako rozšíření příkazové řádky nodejs [9]. Pomocí npm stahujeme v naší aplikaci node.js balíčky, jako jsou:

- Karma [10] (viz. kapitola o testování)
- Protractor [11]
- Bower

Bower [7] je naším druhým balíčkovacím systémem. Pomocí Boweru zpravidla stahujeme již konkrétní Javascriptové knihovny a frameworky použité v naší klientské aplikaci. Jsou to například:

- Angular
- React
- jQuery
- Bootstrap [12]

2.2 Serverová aplikace

Pro serverovou část aplikace byl použit jazyk PHP. PHP je programovací interpretovaný jazyk, pracující na straně serveru. V novějších verzích PHP můžeme používat principy objektově orientované programování. Od verze PHP 5 jsou mimo jiné podporovány:

- konstruktory a destruktory
- klonování objektů
- abstraktní třídy
- rozhraní

Podle [13] se dá PHP použít jak na rozsáhlých projektech, tak i velmi minimalisticky. Můžeme napsat jediný řádek kódu, který funguje a to bez připojení externích knihoven, jako je tomu například u jazyka C. Například vypsání data na stránce vidíme v následující ukázce Kód 2.1.

```
// vypíše dnešní datum
<?php echo date("d. m. Y"); ?>
```

Kód 2.1: Ukázka minimalistického kódu jazyka Php

Verze PHP 5 a starší je slabě typový jazyk [13]. Od verze PHP 7 bude přidána typová kontrola proměnných (více v [14]), která nám velmi pomůže při vývoji, a která bude srovnatelná s významem TypeScriptu [15] pro JavaScript. PHP se může s využitím PDO (PHP Data Objects) napojit na většinu používaných SŘBD, mezi které patří i databáze jako jsou:

- Oracle
- PostgreSQL
- MySQL
- MS-SQL
- SQLite

V této práci používáme MySQL, což je relační SŘBD, který už od počátku svého vývoje kladl velký důraz na výkon a škálovatelnost [13] a je plně kompatibilní s MariaDB. Podle [13] má hlavní vliv na vysoký výkon MySQL právě cachování dotazů. MySQL si totiž uchovává výsledky z předchozích dotazů SELECT. Když se spouštějí dotazy nad databází, jsou nejdříve porovnány s dotazy v cache a pokud jsou shodné, vyhledávací proces se ukončí a vrátí se výsledky z cache.

MySQL mimo jiné obsahuje i podporu:

- Indexování s využitím B-stromu [16]
- Fulltextového indexování a vyhledávání [13]
- Triggerů a uložených procedur [13]

Indexování s využitím B-stromu je metoda pro ukládání a vyhledávání záznamů, kde složitost vyhledávání v N položkách je v nejhorším případě $\log_n(N)$. Fulltextové vyhledávání porovnává hledaná slova s texty uloženými v databázi. Nad rozsáhlými dokumenty bez indexace může být fulltextové vyhledávání pomalé, protože musíme procházet celé dokumenty. V indexovaných dokumentech procházíme pouze indexy a tím se vyhledávání může značně urychlit. V řadě databází potřebujeme po nastání nějaké události, například odstranění řádku, automaticky spustit sekvenci příkazů sloužící například ke kontrole integrity dat. K tomuto slouží trigger (česky spustit), který nám zajistí reakci na danou událost.

2.3 Klientská aplikace

Celá klientská aplikace byla napsána v Javascriptu, který se často používá pouze pro vývoj klientských aplikací. S příchodem nodejs [9] a jádra V8 [17] od Googlu, které zásadně zvýšilo výkon JavaScriptu, se ale v tomto jazyce začaly vytvářet i serverové aplikace. Nodejs [9] je asynchronní událostně orientované běhové prostředí pro Javascriptové serverové aplikace.

JavaScript nám umožňuje vyhledávat, měnit a kontrolovat HTML elementy na stránce. Umožňuje nám validovat formuláře před jejich odesláním, reagovat na kliknutí myší a mnoho dalšího. Vše probíhá bez opětovného načítání stránky. Webová aplikace se tak stává dynamickou, uživatelsky přívětivou single-page aplikací [18]. To znamená, že se uživatel pohybuje mezi částmi (pohledy nebo také View) aplikace bez nutnosti nového načítání stránky. Javascript je plnohodnotný programovací jazyk. Můžeme v něm používat principy objektově orientovaného programování, jako je například dědičnost, či polymorfismus [19]. Javascript je slabě typový jazyk. Typy proměnných můžeme ale přidat použitím TypeScriptu [15]. Ten je nadstavbou jazyka JavaScript, nabízející typovou kontrolu, refactoring kódu a mnoho dalšího. Nevýhodou aplikací v Javascriptu je různé chování na odlišných prohlížečích, a proto existuje řada knihoven a frameworků, které se toto chování snaží sjednotit. Příkladem může být knihovna jQuery [3].

JQuery je knihovna napsaná v jazyce Javascript. Umožňuje vyhledávat a měnit elementy na stránce a reagovat na události. JQuery knihovna nabízí sadu funkcí, které obalují již existující Javascriptové funkce [3]. Nepřináší nám tak žádnou novou funkcionalitu. Místo toho se snaží vytvořit jednotný přístup při psaní Javascriptových aplikací na různých prohlížečích. Díky knihovně jQuery se tak nemusíme starat o to, na jakém prohlížeči naše aplikace běží. JQuery sama rozpozná prohlížeč a přizpůsobí svou funkcionalitu tak, aby dostala stejný, nebo velmi podobný výsledek.

JQuery knihovna podle [3] obsahuje pouze nejpoužívanější funkce pro vývoj aplikací. Proto zůstává relativně malá. V minimalizované verzi má okolo 84 kB (platí pro verzi 2.2.3). Knihovnu ale můžeme rozšiřovat pomocí pluginů. Těch je celá řada. Jenom na oficiálních stránkách jQuery jich najdeme přes 2500. S příchodem pokročilejších knihoven a frameworků roste snaha přepisovat tyto pluginy do nových technologií. A právě tomu se budeme v této práci věnovat.

V Javascriptu se vytvářejí stále složitější aplikace, kdy je snahou převést výpočetní zátěž ze serverové aplikace na klientskou [20]. Proto v posledních letech vznikla i řada knihoven a frameworků, které umožňují vytvářet přehlednější a rychlejší Javascriptové aplikace. Příkladem může být Angular[1], nebo React [2].

React [2] je Javascriptová knihovna, která umožňuje vytvářet a měnit elementy na stránce. Oproti Angularu (viz další kapitola) nevytváří kompletní framework zajišťující nástroje pro vytvoření celé aplikace. Vytváří pouze zobrazovací vrstvu, tedy podle [2] vrstvu View v MVC architektuře.

React umožňuje vytvářet znovupoužitelné komponenty. Když tvoříme aplikaci v knihovně React, vytváříme sadu komponent, které tuto aplikaci tvoří. V návodech v [2] je knihovna React často používána s jQuery. JQuery zajišťuje například AJAXové volání, které knihovna React neobsahuje. Nicméně React můžeme kombinovat i s jinými knihovnami, které zpravidla zajišťují vrstvy s Modely a Controllery.

2.4 Angular

Angular [1] je velmi populární klientsky orientovaný Javascriptový framework. Snadno se používá a obsahuje propracovaný šablonovací systém. Šablonovací systém Angularu je zajímavý v těchto ohledech:

- Používá HTML jako šablonovací jazyk
- Nevyžaduje obnovu okna prohlížeče
- Umožňuje vytvářet nové HTML5 značky (direktivy), které se při běhu aplikace dále zpracovávají jako samostatné a oddělitelné moduly aplikace

Na Angularu nás na první pohled nejvíce zaujme šablonovací systém. Nicméně Angular toho nabízí mnohem více. Nabízí možnost vytvářet Controllery a Modely. K tomu ale nabízí i velké množství předpřipravených služeb a možností, jak naší aplikaci rozšiřovat. Kromě toho Angular klade velký důraz na DI (Dependency Injection).

Dependency Injection (česky vkládání závislostí) je způsob, jak vyvíjet aplikace. Když je naše aplikace závislá na nějakém zdroji, například na službě, můžeme si ji pomocí DI nechat předat. To potvrzuje i [21]. Například vytváříme aplikaci, která je závislá na službě stahující data ze serverové aplikace. Místo toho, abychom službu sami vytvořili, nebo si nechali předat celý soubor všech služeb, si jednoduše necháme předat pouze konkrétní službu. To uděláme vytvořením zápisu, že pro svůj běh potřebujeme tuto službu. Když je naše aplikace vytvořena a spuštěna, služba je jí předána v parametru. Díky tomu se při vývoji aplikace nemusíme starat o její vytváření a předávání napříč aplikací.

Další výhodou Angularu je, že již od začátku nám nabízí možnost testování. Když se s Angularem začneme učit, oficiální tutoriál Angularu nás vede ke psaní testů ihned po implementaci nové funkcionality. Dalo by se tedy říci, že Angular nás od začátku vede k dobrým návykům vytváření aplikace. Testovací nástroje jsou přitom připravené přímo pro Angular. Odpadá tak nutnost hledat a připojovat vlastní testovací nástroj.

2.4.1 Šablonovací systém

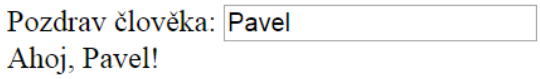
Angular ve svých šablonách používá standardní HTML. Navíc ale HTML rozšiřuje o další značky. Jak můžeme rozšíření HTML využít si vysvětlíme na příkladu. Když se uživatel poprvé přihlásí do naší aplikace, chceme mu zobrazit uvítací hlášku. Do proměnné *firstSignIn* uložíme pravdivostní hodnotu, zda je uživatel přihlášen poprvé. Celá hláška je uložena ve značkách <p>, jak ukazuje Kód 2.2.

```
<p ng-show="firstSignIn">
   Vítejte. Doufáme, že se vám naše aplikace bude líbit.
</p>
```

Kód 2.2: Ukázka direktivy ng-show

Angular HTML rozšířil o atribut *ng-show*. Tento atribut vyhodnocuje výraz v uvozovkách. Pokud je výraz pravda, zobrazí daný element. Pokud není, element nezobrazí. Takto nám Angular do HTML přidává velké množství značek a atributů. Můžeme v něm také vyhodnocovat a zobrazovat samotné proměnné. Opět si uvedeme jednoduchý příklad. Chceme vytvořit *Hello world*. Místo toho, aby náš program pozdravil celý svět, mu můžeme do formulářového pole napsat jméno, koho má pozdravit. Celý příklad vidíme v ukázce Obrázek 2.1.

```
<div ng-init="person = 'Pavel'">
  Pozdrav člověka: <input type="text" ng-model="person">
  <h1>Ahoj, {{person}}!</h1>
</div>
```



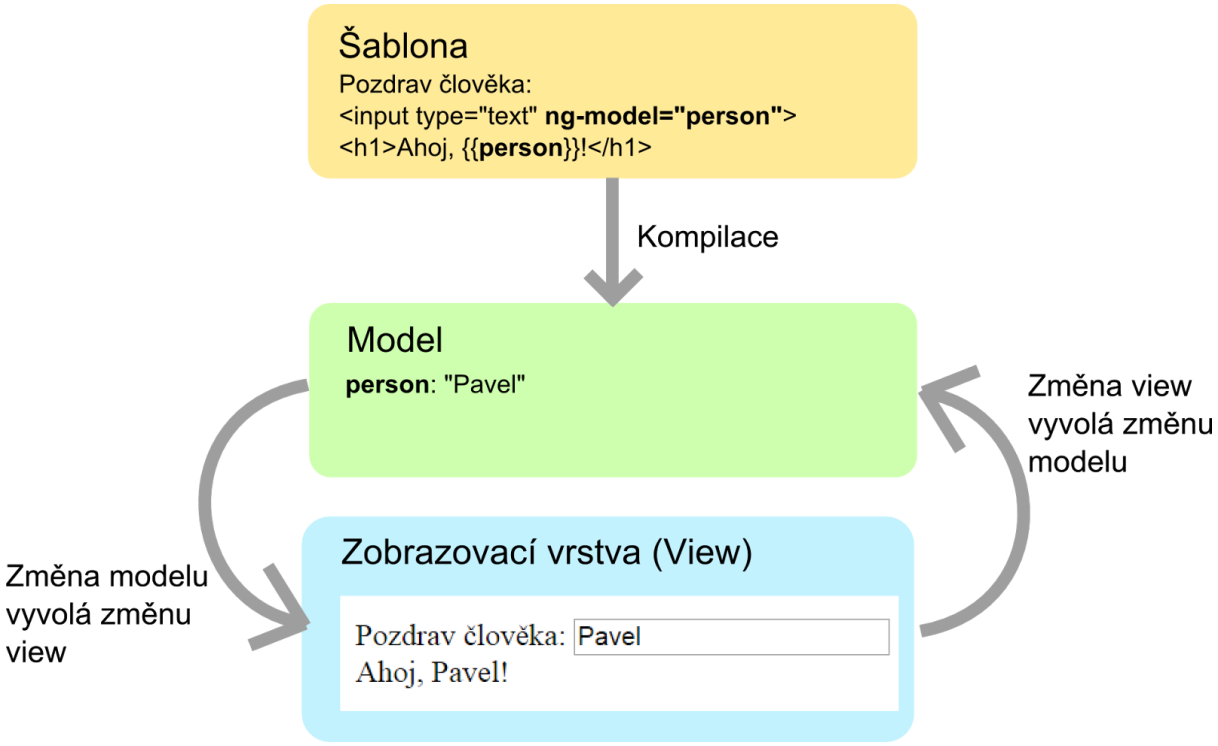
Obrázek 2.1: Hello World s proměnným jménem v prohlížeči

Pomocí atributu *ng-model*, jsme do naší aplikace zavedli novou proměnnou *person*, která se zadává do formulářového pole. Tu potom zobrazíme ve výsledném pozdravu. Ve formulářovém poli můžeme proměnnou měnit a bude se přímo upravovat i náš pozdrav. Vše probíhá v klientské aplikaci, tedy nedochází k obnově stránky. Celé toto propojení nám zajišťuje obousměrný data binding (Two-way data binding), který si vysvětlíme ve stejnojmenné podkapitole. Povšimněme si také, že k vytvoření této mini aplikace nebylo potřeba jediné řádky Javascriptu. To je další výhodou Angularu. Angular se snaží přenášet některé části logiky aplikace přímo do šablonovacího systému. Naše Javascriptové kódy jsou tak kratší a v šablonách přímo vidíme, jaké události a modely jsou s našimi elementy propojené.

2.4.2 Obousměrný data binding

Abychom si mohli vysvětlit, co to je obousměrný data binding (two-way data binding), vraťme se ještě k příkladu z předchozí kapitoly. V ní jsme si ukázali, jak bez použití jediného řádku Javascriptu můžeme vytvořit pozdrav, u kterého dokážeme měnit jméno. Jméno jsme měnili ve formulářovém poli *<input>* a okamžitě se nám měnilo i v našem pozdravu. Toto je možné právě pomocí obousměrného data bindingu.

Podle [1] většina dnešních šablonovacích systémů používá jednosměrný data binding. To znamená, že Model (v našem případě proměnná person) je propojen s šablonou. Jakmile se model změní, změní se i výpis proměnné v šabloně. Angular je ale odlišný. Kontroluje změny zobrazovací vrstvy (View) a pokud ke změně dojde, mění i Model. Změnu Modelu pak okamžitě propaguje do šablon. Celý systém ukazuje Obrázek 2.2 s použitím předchozího příkladu.



Obrázek 2.2: Znázornění obousměrného data bindingu na příkladu

Jak z obrázku vidíme, změna jména ve formulářovém poli s atributem *ng-model* vyvolá změnu i samotného modelu, konkrétně proměnné person. Změna proměnné person, vyvolá změnu zobrazovací vrstvy, tedy výpisu proměnné person. Toto propojení se nazývá Obousměrný data binding.

2.4.3 MVC architektura

MVC architektura je návrhový vzor, jak správně dekomponovat aplikaci do několika vrstev Model, View a Controller. Model reprezentuje data a business logiku, View vytváří zobrazovací vrstvu a Controller se stará o aplikační logiku. Tyto celky by měly být na sobě co nejméně závislé [22]. Změna jednoho by neměla příliš ovlivnit ostatní.

Podobných architektur, jako je MVC, existuje celá řada. Příkladem může být architektura MVP (Model, View, Presenter). Angular podle [1] využívá architekturu MVW (Model, View, Whatever - česky cokoliv). Naznačuje tak, že nepoužívá při svém návrhu pouze Controllery. Místo Controlleru můžeme použít například direktivu, kterou si popíšeme v následující kapitole.

Nyní se ale zaměříme na MVW architekturu s použitím Controlleru. Na jednoduchém příkladu si představíme, jak Angular funguje. Objekt *$scope* zatím chápejme pouze jako objekt, do kterého vložíme data (např. řetězec), která jsou okamžitě dostupná v šabloně. Předchozí příklad upravíme tak, abychom k proměnné *person* mohli přistoupit z Controlleru:

```
// šablona
<div ng-controller="PersonController">
  Pozdrav člověka: <input type="text" ng-model="person">
  <h1>Ahoj, {{person}}!</h1>
</div>

//Controller v Javascriptu
app.controller('PersonController', function($scope) {
 $scope.person = 'Honza';
});
```

Kód 2.3: Příklad jednoduchého použití Controlleru

V ukázce Kód 2.3 přibyla do šablony nová značka *ng-controller*. Ta nám označuje, že o veškerý kód vložený do této značky se stará daný Controller, v našem případě *PersonController*. V samotném Controlleru z ukázky Kód 2.3 pak nastavujeme, že po načtení aplikace obsahuje proměnná *person* jméno *Honza*. Dosáhneme tak stejného výsledku, jako u příkladu výše, kde jsme počáteční inicializaci provedli pomocí atributu ng-init. Nyní máme ale proměnnou dostupnou z Controlleru, a tak s ní můžeme dále pracovat.

V příkladu použití Controlleru z ukázky Kód 2.3 vidíme, jak v praxi probíhá DI. K běhu našeho Controlleru potřebujeme objekt *$scope*. Proto si o jeho vytvoření a předání zažádáme. V Angularu si o něj zažádáme tak, že ho napíšeme do parametru funkce, která představuje Controller. Angular pak objekt rozpozná podle jeho názvu a správně nám ho vytvoří a předá.

Další věcí, které si můžeme na příkladu Kód 2.3 všimnout, je v tuto chvíli magická proměnná *app*. Tu vytváříme v implementaci jako první a představuje nám celou aplikaci. Do této aplikace pak připojujeme další různé části, jako jsou například Controllery, nebo služby. Vytvoření proměnné *app* vidíme v ukázce Kód 2.4.

```
var app = angular.module("simpleApp", []));
```

Kód 2.4: Ukázka vytvoření jednoduché aplikace

Aplikace vzniká tím, že vytvoříme modul v Angularu. Modul si v Angularu podle [1] můžeme představit, jako jmenný prostor, kam připojujeme Controllery, služby a další závislosti. V prvním parametru vytvoříme unikátní název modulu. V našem případě to bude název *simpleApp*. V druhém argumentu si pak můžeme nechat předat závislosti, které naše aplikace ke svému běhu vyžaduje.

2.4.4 Scope

O objektu *$scope* by se dalo napsat mnoho kapitol. My si tu pouze shrneme některé jeho důležité vlastnosti. Jak jsme si již napsali, *$scope* nám slouží, jako přepravka dat mezi Controllerem a šablonou. Můžeme do něj uložit řetězec a v šabloně tento řetězec najdeme ve stejnojmenné proměnné. To ale není jediná vlastnost objektu *$scope*. Podle [1] *$scope* obsahuje i velkou řadu užitečných funkcí. Postupně si tu představíme:

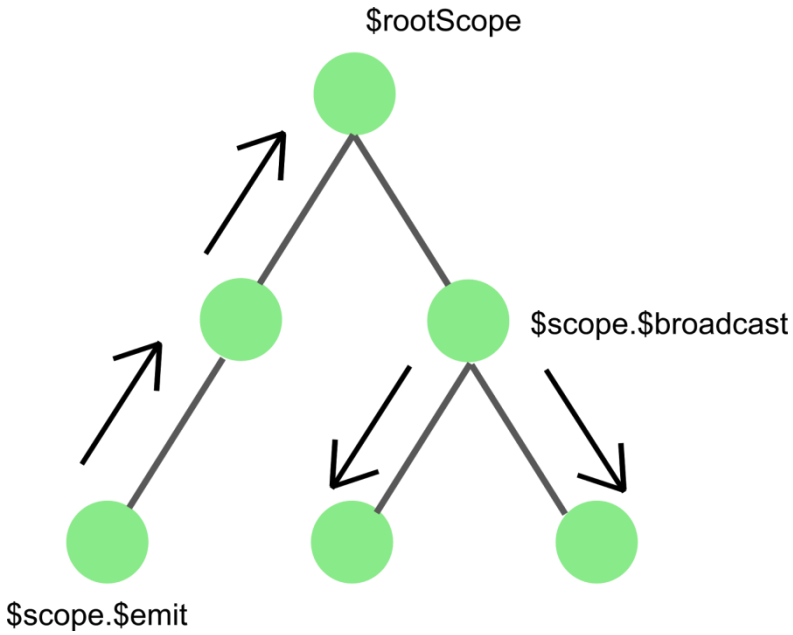
- $watch
- $new
- $emit a $broadcast

Funkce *$watch* hlídá změnu dané proměnné. Jednoduše napíšeme funkci, která se spustí v momentě, kdy dojde ke změně. Kdybychom podobný kód chtěli implementovat pouze v Javascriptu, nebylo by to tak snadné. Museli bychom zjistit, jaký typ proměnné chceme hlídat. Podle toho bychom na ni buď navázali událost (například událost *onchange*), nebo ji periodicky kontrolovali zda se nezměnila. Tohle vše Angular řeší za nás a my se tak můžeme soustředit na psaní výkonného kódu. Příklad použití funkce *$watch* vidíme v ukázce Kód 2.5:

```
// oldValue - hodnota proměnné před změnou, newValue - hodnota po změně
$scope.$watch('nazevHlidanePromenne', function(newValue, oldValue) {
   //implementace jednoduchého počítadla změn
   $scope.counter = $scope.counter + 1;
});
```

Kód 2.5: Použití funkce *$watch*

Funkce *$scope.$new* podle [1] slouží k vytvoření nového *$scope*. Tuto funkci zpravidla používáme, aniž o tom víme. Každý Controller dostane při vytvoření nové *$scope*. Vytváření nového *$scope* je důležité, protože tak nedochází k prolínání dat. Kdybychom měli jen jeden *$scope* pro celý projekt, tak by docházelo k tomu, že by se střetávali názvy proměnných. Jeden Controller by nastavil například proměnnou *person na* hodnotu *"Honza"* a druhý by ji přepsal na hodnotu *5*. Každý by totiž používal tuto proměnou jinak bez toho, aby o sobě věděli. Při vytváření nového *$scope*, podle [1] dochází k nastavení rodičovského *$scope*. Při vytváření aplikace tak vytváříme i strom *$scope*, který má jeden společný kořen *$rootScope*. V tomto stromě si pak můžeme předávat různé události.



Obrázek 2.3: Scope strom a předávání událostí

Jelikož v implementaci jednotlivých pluginů často narážíme na předávání událostí, nastíníme si jejich princip. Pomocí funkce *$scope.$on* můžeme reagovat na určitou událost. Příklad takové události může být například událost *$destroy*, která se nám pošle před odstraněním daného uzlu. Na takovou událost reagujeme například tím, že odpojíme všechny funkce, které mají nastavené periodické spuštění nad daným uzlem a mohli by tak v budoucnu působit chyby.

Události se delegují napříč celým stromem *$scope*. K delegování událostí můžeme použít jednu ze dvou základních funkcí. Funkce *$scope.$emit* deleguje událost směrem ke kořenu stromu. Funkce *$scope.$broadcast* deleguje událost směrem dolů, viz. Obrázek 2.3.Výše zmíněná událost *$destroy* patří mezi základní události v Angularu. Můžeme si ale vytvořit i vlastní události, které můžeme propagovat napříč celým stromem. Podle [1] ale musíme být velmi opatrní při vytváření nových událostí. Vytváření vlastních událostí může mít totiž špatný vliv na výkon celé aplikace.

2.4.5 Direktivy

Direktivy tvoří větší část celé naší aplikace. Proto si je zde rozebereme. Direktivy nám v Angularu umožňují vytvářet nové značky a atributy. Zpravidla, když jsme si v kapitolách výše psali o novém atributu, či značce v Angularu, mluvili jsme vlastně o direktivě. Direktivy, které poskytuje přímo Angular, mají předponu *ng*. Příklady takovýchto direktiv jsou:

- ng-controller
- ng-repeat
- ng-click

Direktivu *ng-controller* jsme si ukazovali v příkladu výše. Slouží k tomu, že specifikuje zobrazovací vrstvu (View) konkrétního Controlleru. Když například budeme mít v elementu *div* direktivu *ng-controller=”PersonController”* víme, že vše, co je v tomto elementu, obsluhuje tento Controller.

Direktiva *ng-repeat* slouží k procházení pole prvků. Používá se například v momentě, kdy chceme vytvořit z pole HTML seznam prvků. Příklad, který ukazuje Kód 2.6, nám vygeneruje seznam všech ročních období.

```
// v Controllerru
$scope.seasons = ['jaro', 'léto', 'podzim', 'zima']
// v šabloně
<ul>
  <li ng-repeat="season in seasons" >{{ season }}</li>
</ul>
```

Kód 2.6: Ukázka použití direktivy ng-repeat

Posledním příkladem direktivy je *ng-click*. Tato direktiva slouží k tomu, že spustí danou funkci po kliknutí na element. Do funkce můžeme předat jako parametr proměnnou používanou v šabloně. Pro ukázku si rozšíříme předchozí příklad. Když klikneme na měsíc, vytvoří se vyskakovací okno se jménem ročního období, jak vidíme v ukázce Kód 2.7.

```
// v šabloně
<li ng-repeat="season in seasons" ng-click="showSeason(season)">
{{ season }}</li>
// v Controlleru
$scope.showSeason = function(seasonName) {
  alert("Klikli jste na " + seasonName);
}
```

Kód 2.7: Ukázka použití direktivy ng-click

Nemusíme se ale omezit jen na používání předpřipravených direktiv. Můžeme si vytvořit i vlastní direktivu podle našich představ. Vytváření direktiv je velmi mocný nástroj Angularu. Můžeme si díky tomu vytvářet nové HTML značky, atributy či třídy. Vše si opět ukážeme na příkladu. V našem projektu chceme použít framework Bootstrap [12]. V projektu máme mnoho tabulek, a aby bylo možné použít styly z Bootstrapu, musí taková tabulka mít třídu *table*. Navíc některé tabulky je potřeba ohraničit čárou (třída *table-bordered*), nebo pro přehlednost zvýraznit každý druhý řádek tabulky (třída *table-striped*). Abychom tohoto dosáhli, vytvoříme v ukázce Kód 2.8 direktivu *table*.

```
//šablona v Angularu
<table type='striped'>
 … obsah tabulky ...
</table>

//výsledný HTML kód
<table class='table table-striped'>
 … obsah tabulky ...
</table>

//direktiva
app.directive('table', function() {
  return {
     restrict: 'E',
     compile: function(element, attributes) {
        element.addClass('table'); //přidá každé tabulce třídu table
        if ( attributes.type ) { //přidá např. table-striped
           element.addClass('table-' + attributes.type);
        }
     }
};});
```

Kód 2.8: Ukázková direktiva table

Další výhodou je také to, že při přechodu na novou verzi Bootstrapu nemusíme měnit značky v celém našem projektu. Pokud by nový Bootstrap používal místo třídy *table* zkrácený zápis *tbl*, stačí nám zápis změnit na jednom místě. Výsledek se pak projeví v celé aplikaci.

Jak vidíme v ukázce Kód 2.8, direktivu vytváříme pomocí funkce directive. Té je předán název a funkce, která vrací její nastavení. V nastavení máme nyní dva parametry:

- restrict
- compile

Parametr *restrict E* (element) udává, že direktiva bude vytvořena jako HTML značka. Direktivu ale můžeme k elementu vázat i jako atribut, jako je tomu například u direktivy ng-click. V tomto případě by pak byl parametr *restrict* nastaven na *A* (attribute).

Direktivy můžeme tvořit ještě dalšími dvěma způsoby. Třídou označenou písmenem *C* (class) a komentářem *M* (comment).

Druhým parametrem je funkce *compile*. Ta nám tvoří výkonný kód dané direktivy. Je jí předán již konkrétní element a jeho parametry. Funkce nám pak element upraví přidáním potřebných CSS tříd.

2.4.6 Direktivy s vlastními šablonami

Direktiva nemusí pouze měnit stávající HTML elementy. Může vytvářet i nové elementy nebo vkládat HTML šablony. Vkládání šablon se nám hodí hlavně u složitějších aplikací. Pokud chceme vytvořit znovupoužitelnou, ale nezávislou funkcionalitu, vytvoříme jí pravděpodobně pomocí direktivy. A pokud má tato direktiva vytvářet větší množství HTML kódu, použijeme k tomu HTML šablonu. Uveďme si příklad. Chceme vytvořit profilovou fotku uživatele s jeho jménem. Abychom nemuseli kopírovat HTML kód s profilovou fotkou, vytvoříme si na to direktivu *ProfilePhoto*. Chceme, aby bylo použití direktivy a výsledný kód stejný, jako je v ukázce Kód 2.9.

```
//použití v naší aplikaci
<profile-photo imgPath='honza.jpg' userName='Honza Novák' />

//výsledný HTML kód
<div>
  <img src='/photos/honza.jpg' alt='Honza Novák'>
  <span class='userName'>Honza Novák</span>
</div>
```

Kód 2.9: Ukázka použití direktivy s vlastní šablonou

Vytvoříme direktivu *ProfilePhoto*, která má dva atributy *imgPath* a *userName*. V prvním atributu uvedeme relativní cestu k profilové fotce a v druhém jméno uživatele. Direktiva by nám měla vygenerovat kód, který vidíme v druhé části ukázky Kód 2.9. Kdybychom takovouto direktivu chtěli napsat v Angularu bez použití HTML šablony, stálo by nás to nemalé úsilí v Javascriptovém kódu. Museli bychom přes metody z jqLite [21] vytvořit všechny elementy a správně je do sebe napojit. Výsledný kód by byl nepřehledný, špatně udržovatelný a nápadně podobný jQuery pluginu.

K vytvoření následného HTML použijeme šablonu, aby náš kód zůstal přehledný. Šablona je blok HTML kódu, který daná direktiva spravuje. Standardně se šablona direktivy odděluje do zvláštního HTML souboru, ale může být i ve stejném souboru, jako direktiva. V šabloně si vytvoříme veškerý HTML kód, který pak chceme uživateli zobrazit. Implementaci výše zmíněného příkladu vidíme v ukázce Kód 2.10.

```
//direktiva
app.directive('profilePhoto', function() {
  return {
     restrict:'E',
     replace: true,
     templateUrl:'template/profilePhoto.html',
     link: function($scope, element, attrs) {
       $scope.imgPath = '/photos/' + attrs.imgPath;
       $scope.userName = attrs.userName;
     })
  }
});
//šablona direktivy - profilePhoto.html
<div>
  <img src='{{ imgPath}}' alt='{{ userName }}'>
  <span class='userName'>{{ userName }}</span>
</div>
```

Kód 2.10: Ukázka implementace direktivy s vlastní šablonou

V direktivě *ProfilePhoto* nám přibyly parametry, které ještě neznáme. Postupně si je projdeme. Jedná se o parametry:

- replace
- templateUrl
- link

Parametr replace true znamená, že se má nahradit aktuální HTML kód direktivy šablonou. Odstraní se tedy celý element *<profile-photo>* a nahradí se elementem * *, který tvoří kořenový element této šablony. Dalším důležitým parametrem je *templateUrl*. Ten nám udává cestu k šabloně dané direktivy. Posledním novým parametrem je funkce *link*. Tato funkce je obdoba funkce *compile*. Připravuje nám proměnné, které budou používány v dané šabloně.

Použití direktiv se šablonami nám tvoří přehledný a znovupoužitelný kód. Chceme-li vytvořit funkcionalitu, kterou můžeme používat v různých projektech, použijeme právě direktivy s šablonami. V naší práci používáme direktivy například pro vytváření grafů. Díky direktivám se grafy dají použít i v jiných aplikacích, nebo vícekrát na jedné stránce.

2.4.7 Konverze jQuery pluginu do Angularu

V následujícím textu budeme často využívat pojmy jQuery plugin a Angular direktiva. Proto si je jednoduše popíšeme. JQuery plugin je funkcionalita oddělená od ostatního kódu. Plugin má vlastní nastavení a může být v aplikaci použit vícekrát. Příkladem takového pluginu může být například plugin, který vytvoří z existujících dat spojnicový graf. Angular direktiva je jQuery pluginu velmi podobná. Také se jedná o oddělenou

funkcionalitu nezávislou na zbytku aplikace. Oproti pluginu má ale přesně danou strukturu popsanou v dokumentaci. Dokumentace nám pomáhá s lepší orientací ve větších direktivách. Díky dokumentaci se také s direktivami lépe pracuje.

V této práci jsme se z části věnovali přepisování existujících pluginů ze starší technologie jQuery do novější technologie Angular. Máme hned několik možností, jak postupovat. Všechny tyto varianty jsou odlišné a ovlivňují výslednou náročnost a kvalitu přepsaného řešení. Proto by mohlo být zajímavé nastínit si postup, jak tento přechod udělat:

- postupně přepisovat plugin do Angularu
- nejdříve si vytvořit direktivu v Angularu a až pak přidávat funkcionalitu
- z jQuery pluginu vzít pouze matematické vzorce a zbytek implementovat znovu

Asi nejsnazší variantou je postupné přepisování pluginu z jQuery do Angularu. Musíme si totiž uvědomit, že Angular i jQuery mají společné jádro. Toto jádro se v Angularu nazývá podle [21] jqLite. Jádro zajišťuje především práci s HTML DOM. Díky tomu, že je jádro společné, velká část funkcí pro práci s elementy má stejnou funkcionalitu [20] a často i stejný název [10]. Díky tomu víme, že nemusíme přepisovat veškerou implementaci, ale že můžeme některé části jQuery pluginu použít. Také můžeme použít implementované matematické funkce nebo funkce pro práci se souřadnicemi. Nakonec ale stejně větší část pluginu úplně přepíšeme. Musíme totiž přepsat mimo jiné celé vykreslování HTML z jQuery do Angular šablon.

Druhou možností je vytvořit si nejdříve direktivu v Angularu a poté přidávat postupně funkcionalitu. Pokud přepisujeme spojnicový graf, tak se nejdříve snažíme vykreslit samotné body. Když máme funkční počítání souřadnic a vykreslování bodů, proložíme je křivkami. Nakonec můžeme např. přidat událost, že pokud uživatel najede kurzorem myši nad bod, tak se bod zvýrazní a ukáže se popisek. Toto řešení je oproti předchozímu náročnější. Přepisování pluginu po částech nás vede k lepší dekompozici kódu a zpravidla nejsme tak závislí na knihovně jQuery.

Poslední možností je plugin nepřepisovat. Vytvoříme úplně novou implementaci a z jQuery pluginu použijeme pouze matematické vzorce. Tento postup je asi nejčistší. Výsledkem je přehledný kód, který vychází přímo ze zásad vývoje v Angularu. Místo toho, abychom se Angular snažili přizpůsobit pluginu, vytváříme kód, který přizpůsobujeme Angularu. U tohoto postupu je nejpravděpodobnější, že výsledný kód nebude nijak závislý na jQuery. To přinese nejen odstranění závislosti na této knihovně, ale i větší rychlost, protože jQuery vůbec nemusíme do projektu připojit. Přehled a porovnání jmenovaných možností ukazuje Tabulka 2.1.

	Výhody	Nevýhody
Postupné přepisování	V každé fázi je program spustitelný. Nemusíme předem znát veškeré funkce Angularu	Výsledný kód je zpravidla více závislý na jQuery, než by musel.
Přidávání funkcionality do předem vytvořené direktivy	V každé fázi je program spustitelný. Nejsme tak závislí na knihovně jQuery.	Jsou kladeny větší nároky na komplexní znalosti Angularu.
Z jQuery převzít pouze matematické vzorce	Nemusíme být vůbec závislí na knihovně jQuery. Toto řešení má největší potenciál využít nejlepší praktiky vytváření kódu (best practices) z Angularu	Vysoká náročnost. Komplexní znalost Angularu a postupů na vytváření direktiv.

Tabulka 2.1: Porovnání jednotlivých postupů přepisování pluginu z jQuery do Angularu

Nyní se můžeme pustit do samotného přepisování. Jak jsme si již napsali, některé části z existujícího pluginu můžeme využít. Nyní si uvedeme části, které se musí přepsat v každé ze tří uvedených variant:

- Pluginy přepisujeme do direktiv
- Vytváření HTML
- Navázání událostí
- AJAXové volání

Přepsání funkcí pro vytváření HTML znamená, že HTML, které nám vygeneroval jQuery plugin, by měla vygenerovat i direktiva v Angularu. Když vygenerujeme podobné HTML, výsledná direktiva v Angularu může být pro uživatele od jQuery pluginu k nerozeznání. Uživatel tedy nezaznamená změnu, ale my můžeme využívat výhod, které nám Angular nabízí. Tato část práce je asi zároveň nejnáročnější a některé další body z ní vychází.

Při vytváření HTML se nevyplatí dívat do kódu pluginu jQuery a přemýšlet, jaké HTML plugin vygeneruje. Vytváření elementů v jQuery není přehledné a málokdy jde na první pohled zjistit, jak bude výsledné HTML vypadat. Proto je asi nejlepší variantou plugin spustit a podívat se přímo na výsledné HTML. Toto HTML si pak můžeme překopírovat a přepsat v HTML šabloně, kterou Angular používá. V ní pak pomocí předpřipravených direktiv z Angularu vytvoříme dynamické vykreslování HTML a napojíme obsluhu událostí.

Jelikož se jedná o nejtěžší část práce, celý problém si vysvětlíme na příkladu. Chceme vytvořit velmi jednoduchou direktivu, která zobrazí kontakt na uživatele. Pro zjednodušení zobrazí pouze jméno, příjmení a email uživatele. Toto zadání již řeší plugin v jQuery, který chceme přepsat do Angularu. Javascriptový objekt se posílá pluginu ve formátu, který vidíme v ukázce Kód 2.11.

```
{firstName: "Petr", lastName: "Kukrál", email: "kukral@students.zcu.cz"}
```

Kód 2.11: Objekt s daty v ukázkové aplikaci

Tento objekt se předá jQuery pluginu, který vykreslí HTML z ukázky Kód 2.12.

```
<div class='contact'>
  <span>Petr</span>
  <span class='lastName'>Kukrál</span>
  <a href="mailto:kukral@students.zcu.cz">kukral@students.zcu.cz</a>
</div>
```

Kód 2.12: Výsledné HTML v ukázkové aplikaci

Naším cílem je vykreslit to samé HTML. Naším důvodem může být, že grafický návrh aplikace je již dávno hotový a odzkoušený a jsou implementované CSS. Pokud výsledné HTML změníme, musíme zasáhnout i do těchto souborů. Pokud nahlédneme do jQuery pluginu, pravděpodobně nalezneme mnoho řádek Javascriptového kódu, vytvářejícího HTML elementy. Ty jsou pak vkládány do elementu *contact*. Takový kód je dlouhý, nepřehledný a nejde nijak využít. Můžeme ale využít vygenerovaný HTML kód z ukázky Kód 2.13. Stačí ho pouze trochu upravit:

```
<div class='contact'>
  <span>{{contact.firstName}}</span>
  <span class='lastName'>{{contact.lastName}}</span>
  <a href="mailto:{{contact.email}}">{{contact.email}}</a>
</div>
```

Kód 2.13: Upravené HTML pro Angular v ukázkové aplikaci

Výhodou tohoto řešení je, že Angular šablona je jasná a přehledná. Také se nám značně zkrátí Javascriptový kód. V něm totiž vůbec nepracujeme s HTML elementy. Jediné, co uděláme je to, že předáme šabloně data, která má vykreslit. Jak vidíme v ukázce Kód 2.14, celý Javascritový kód se nám tak zkrátí na jedinou řádku:

```
$scope.contact = contact;
```

Kód 2.14: Výsledný Javascript pro nastavení šablony v ukázkové aplikaci

Pokud přepisujeme složitější plugin, pravděpodobně musíme do implementace pluginu stejně nahlédnout, plugin si projít a udělat si představu o tom, jak funguje. U jednoduchých pluginů to nemusí být vůbec potřeba.

Už jsme si vysvětlili, jak přepsat vykreslování HTML. Nyní si napíšeme něco o reakcích na události. Pokud chceme reagovat na nějakou událost v jQuery, máme na to jednoduché předpřipravené funkce. Takový příkladem je funkce click [1], do které vložíme kód, který se má vykonat. Tento postup je velmi podobný i v Angularu. Také určíme funkci, která se má vykonat po kliknutí na element. Rozdíl je ale v tom, jakým způsobem na stránce vyhledáváme element, ke kterému chceme událost připojit. V jQuery budeme muset použít vyhledávání nad celou stránkou, nebo v lepším případě vyhledávání jen v části stránky. Kvůli tomu budeme muset často přidávat nové třídy, abychom rozeznali jednotlivé prvky na stránce. Angular celý tento problém řeší za nás. Přímo v HTML si do daného elementu pouze přidáme jednoduchou ng-direktivu, do které napíšeme, jaká funkce se má při této události zavolat. Vše si opět ukážeme na jednoduchém příkladu.

Máme kód pro zobrazení kontaktu z příkladu výše. Po kliknutí na email se zobrazí velmi jednoduchá zpráva: *"Opravdu chcete uživateli Petr Kukrál napsat email?"* Zpráva se zobrazíme pomocí metody confirm [23], která zobrazí vyskakovací okno s toutu zprávou a možnostmi OK a CANCEL. HTML, které generuje jQuery plugin, vidíme v ukázce Kód 2.15.

```
<div class='contact'>
  ... vypsání jména a příjmení ...
  <a href="mailto:kukral@students.zcu.cz" class='email' data-name='Petr
    Kukrál'>kukral@students.zcu.cz</a>
</div>

... obsluha události v jQuery pluginu ...
$('.contact .email').click(function(e){
  var name = $(this).data('name');
  confirm('Opravdu chcete uživateli ' + name + ' napsat email?');
});
```

Kód 2.15: Ukázka přepisování událostí z jQuery do Angularu

Jelikož kód přepisujeme do Angularu, nemusíme používat vyhledávání elementu email pomocí zápisu *$('.contact .email')*. Ani nemusíme předávat data přes atribut *data*. Naopak přidáme do HTML direktivu *ng-click*, která nám celé toto propojení pohodlně zajistí. Data o uživateli si jednoduše předáme přímo přes toto volání. Výslednou Angular šablonu tedy pouze upravíme tím, že přidáme *ng-click*, tedy reakci na kliknutí na daný element. Jak vidíme z ukázky Kód 2.16, výsledný kód se nám o něco zpřehlední:

```
<div class='contact'>
  ... vypsání jména a příjmení ...
  <a href="mailto:{{contact.email}}" ng-click='clickFn(contact)'>
    {{contact.email}}
  </a>
</div>

... obsluha události v Angular direktivě ...
$scope.clickFn = function(c) {
  confirm('Opravdu chcete uživateli '+ c.firstName +' '+ c.lastName
+'napsat email?');
}
```

Kód 2.16: Ukázka upravené reakce na událost v Angularu

Obě ukázky kódu jsou na první pohled podobné. Mějme ale na paměti, že u ukázky v jQuery zobrazujeme až vygenerovaný HTML kód. Kdybychom zobrazovali kód, který HTML vytváří, byl by výrazně delší.

Dalším bodem je přepsání všech AJAXových dotazů. AJAXový dotaz je http požadavek klientské aplikace, který nezpůsobí obnovení stránky. Můžeme si tak například aktualizovat data v grafu bez nutnosti obnovit stránku. JQuery k takovým dotazům zpravidla používá podle [3] funkci *jQuery.ajax*. Toto volání musíme při přepisování pluginu nahradit. Podle [1] můžeme použít buď nízkoúrovňovou službu *$http*, nebo službu *$resource*. Služba *$http* je více podobná funkci *jQuery.ajax*. Dovoluje nám pokládat specifické http požadavky (např. GET, POST atd.) a zpracovávat přímo odpověď serveru, podobně jako *jQuery.ajax*. S jejím využitím se nám bude plugin snáze přepisovat. V naší práci ale při přepisování používáme službu *$resource*. Oproti službě *$http* už požadavky nepokládáme přímo. Místo toho si o potřebná data pouze zažádáme. Služba *$resource* pak podle naší konfigurace vytvoří konkrétní požadavek a výsledek převede do námi určeného formátu. Používání služby *$resource* nám tedy ušetří skládání dotazu a převádění vrácené odpovědi například na pole či objekt.

V návodu si uvádíme řadu postupů a technologií, které je dobré při přepisování pluginu do Angularu znát. Výhodou Angularu je, že má velmi podrobnou dokumentaci, ve které se dobře orientuje. Nejde jen o pouhý manuál, ale dokumentace obsahuje i řadu ukázek a příkladů, na kterých problém rychleji pochopíme. Kolem Angularu je velmi aktivní komunita, kterou vidíme například při pohledu na Github [24]. Angular zde má okolo 50 tisíc hvězd (kladných ohodnocení od uživatelů) a 20 tisíc kopií repozitáře (forků). Když Angular porovnáme s jQuery, které je o 4 roky starší, má jQuery o 10 tisíc hvězd méně a kopií má pouze polovinu. Proto když narazíme na problém, je velmi pravděpodobné, že ho neřešíme první, a že k němu existuje zpracované řešení.

2.5 Vizualizace dat

Než se pustíme do samotné implementace, musíme vybrat správnou technologii pro zobrazování dat. V práci používáme různé druhy vizualizace dat na webu. Zde si ukážeme, k čemu jsou vhodné, a jaká mají omezení. Konkrétně se podíváme na technologie:

- HTML5 canvas
- SVG
- HTML elementy

2.5.1 HTML5 canvas

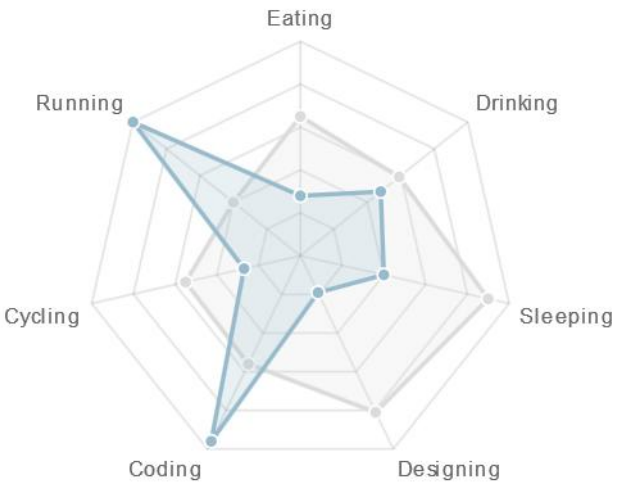
HTML5 canvas [25] můžeme chápat, jako kreslící plátno, do kterého kreslíme přímky a křivky. Jedná se o HTML element, který nabízí kreslení pomocí Javascriptu. Od elementu canvas si můžeme nechat vrátit 2D context, který nám již nabízí samotné funkce pro kreslení. Příkladem takové funkce je *lineTo*, která nakreslí přímku z aktuálních do zadaných souřadnic, nebo funkce *moveTo*, která posune kreslící bod na zadané souřadnice. Pro lepší představu kreslení v canvasu je zde Obrázek 2.4, pomocí něhož si ukážeme, jak nakreslit jednoduchý červený čtverec.

```
//vyhledáme element canvas na stránce
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d"); //vrátíme si context

ctx.fillStyle = "rgb(200,0,0)"; //nyní budeme kreslit červenou barvou
ctx.fillRect (10, 10, 55, 50); //nakreslíme jednoduchý čtverec
```

Obrázek 2.4: Nakreslení jednoduchého čtverce v canvasu

Výhodou kreslení pomocí canvasu je možnost vizualizovat data mnoha možnými způsoby. Samozřejmostí je vizualizace dat pomocí sloupcových, koláčových, nebo spojnicových grafů. Jak ale ukazuje knihovna chartjs [26], využívající ke svému běhu právě canvas, můžeme vizualizovat data i pomocí paprskového grafu, jak ukazuje Obrázek 2.5.



Obrázek 2.5: Ukázka paprskového grafu v canvasu knihovny chartjs [26]

Canvas má ale i několik zásadních nevýhod. Jelikož kreslíme přímo do plátna, nevytváříme žádné HTML elementy. Na HTML elementy se v Javascriptu dobře vážou události. Například když najedeme nad sloupec v grafu, mohl by se nám zobrazit dodatečný popisek sloupce. Kdyby byl sloupec HTML element, snadno k němu navážeme takovou událost. Jednoduše pomocí událostí [27] napíšeme, že pokud je kurzor myši nad tímto sloupcem, má se zobrazit jeho popisek. Nestaráme se tak o to, na jakých souřadnicích je daný element, a na jakých je kurzor. Pokud ale kreslíme pomocí canvasu, nemůžeme navázat událost přímo k vykreslenému sloupci. Naopak musíme složitě přepočítávat souřadnice objektů a kurzoru myši. To nám přináší nutnost implementovat další kód, který nepřináší žádnou další funkcionalitu a představuje velké riziko chyby.

Další nevýhodou je responsibilita. Když je graf responzivní, dokáže se jeho velikost přizpůsobit velikosti stránky. To nám ale canvas nenabízí. Pokud chceme graf po zmenšení stránky překreslit, musíme si ručně přepočítat souřadnice a pravděpodobně i celý graf smazat a nakreslit ho znovu.

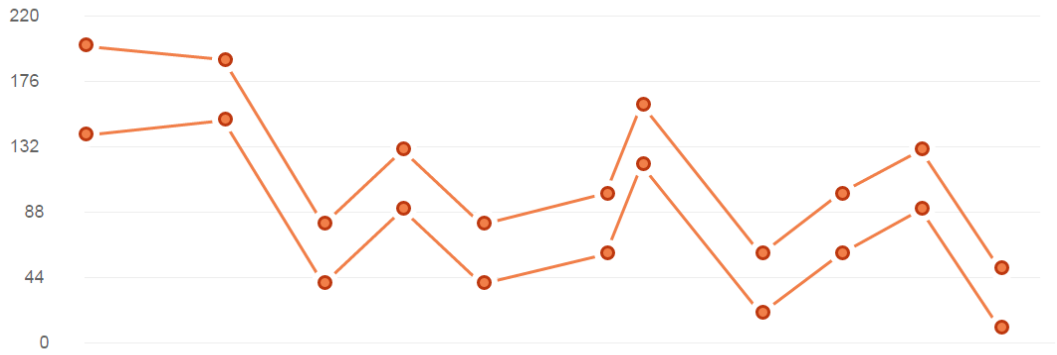
2.5.2 SVG

SVG je při zobrazování dat zajímavá varianta. S tímto velmi známým formátem se pracuje ve vektorovém kreslení. Jelikož jde o známý formát, existuje o něm velké množství knih, návodů a článků. Můžeme si i zdarma stáhnout ikonky a další obrázky v SVG, které pak můžeme v naší práci využít. Kreslení v SVG probíhá tak, že do HTML stránky vložíme element SVG. Do tohoto elementu pak již můžeme vkládat přímo SVG kód. Obrázek 2.6 zobrazuje malbu jednoduchého čtverce v SVG. Přímo tento kód můžeme vložit do HTML.

```
<svg width="100" height="100">
  <rect width="100" height="100" style="fill:rgb(200,0,0)" />
</svg>
```

Obrázek 2.6: Nakreslení jednoduchého čtverce v SVG

Mezi výhody SVG patří, že můžeme kreslit různé typy grafů. Určitě můžeme vykreslit koláčový, sloupcový či spojnicový graf. Můžeme také společně s CSS vytvářet různé animace, jako používá [28]. Příklad grafu kresleného v SVG ukazuje Obrázek 2.7.

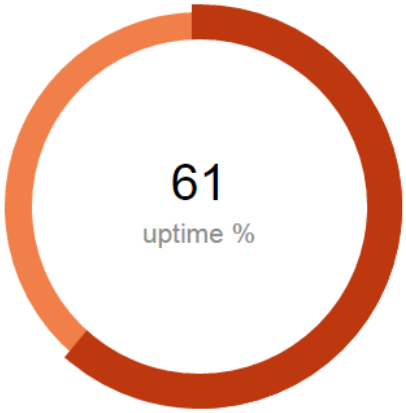


Obrázek 2.7: Spojnicový graf s použitím SVG a HTML v knihovně [29]

Pokud se rozhodneme použít SVG, tak v něm budeme schopni nakreslit všechny druhy grafů, nebo alespoň velkou většinu. Další výhodou SVG je, že ho můžeme vkládat přímo do HTML. Pomocí Javascriptu ho za běhu aplikace můžeme měnit. Můžeme měnit barvu čar, zvětšit velikost bodu při najetí kurzorem, nebo překreslit graf úplně jiným grafem. Nemusíme se při tom starat o počítání souřadnic, na kterých se element nachází, protože události můžeme napojovat přímo na HTML elementy. Toto je veliká výhoda hlavně oproti canvasu. Také se nám dobře vytvářejí responzivní grafy. Mezi nevýhody ještě donedávna patřila podpora. Nicméně nyní již většina používaných verzí prohlížečů dokáže se SVG pracovat. Podporu SVG v jednotlivých prohlížečích si ukážeme v konečném shrnutí.

2.5.3 HTML elementy

Poslední možností, jak kreslit grafy, je pomocí HTML elementů. Pomocí různých elementů (*div, ul, li* ...) a CSS jsme schopni vykreslit některé typy grafů. V HTML můžeme vykreslit například sloupcový graf. Sloupce jsou tvořeny jednotlivými blokovými elementy.



Obrázek 2.8: Ukázka donut grafu v HTML a CSS3 z knihovny [29]

Když použijeme CSS3, jsme schopni vykreslit i donut graf, který zobrazuje Obrázek 2.8. Abychom měli porovnání úplné, i zde si ukážeme, jak nakreslit jednoduchý čtverec v pomocí HTML, jak ukazuje Obrázek 2.9.

```
// HTML
<div class="square"></div>

//CSS
.square {width:100px; height:100px; background-color: rgb(200,0,0)}
```

Obrázek 2.9: Nakreslení jednoduchého čtverce v HTML

Hlavní výhodou používání HTML pro vizualizaci je, že do HTML stránky vkládáme grafy ve stejné technologii. S HTML umí prohlížeče skvěle pracovat. Výborně ho podporuje Javascript, a pokud nevyužíváme CSS3, máme podporu napříč všemi prohlížeči. Velmi dobře se nám vytvářejí responzivní grafy. Stačí zadat, kolik procent mají být elementy velké a prohlížeč si už sám přepočítá potřebné výšky a šířky všech elementů. Další nesmírnou výhodou je použití CSS3 pro animace. CSS3 animace jsou implementované přímo na straně prohlížeče. Ten oproti JavaScriptovým animacím ví, že vykonává animaci a může její běh přenést do vlastního vlákna [30]. Podle [31] CSS3 animace vždy upřednostňujeme před JavaScriptovými animacemi.

Hlavní nevýhodou je, že nejsme schopni kreslit všechny typy grafů. V HTML nenakreslíme například spojnicové nebo koláčové grafy. Do příchodu CSS3 jsme ale nebyli schopni nakreslit ani donut graf. Je tedy otázkou, jak dlouho tento problém bude trvat.

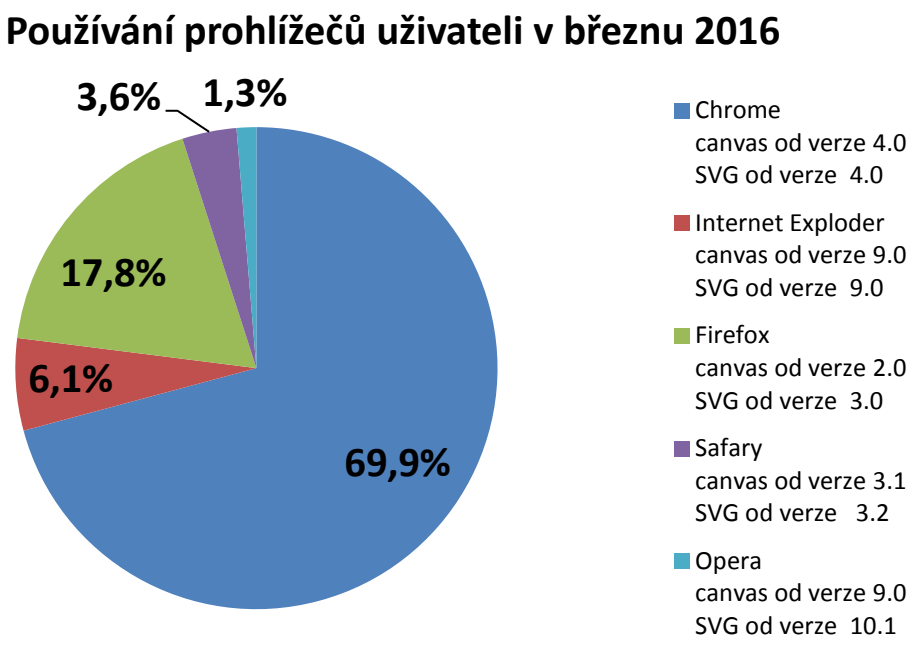
2.5.4 Závěrečné porovnání

Tabulka 2.2 porovnává jednotlivé technologie pro vizualizaci dat. V tomto porovnání nejlépe vychází SVG, které vyhovuje ve všech uvedených možnostech.

	canvas	**SVG**	**HTML a CSS**
Vykreslení sloupcového grafu	ano	ano	ano
Vykreslení koláčového grafu	ano	ano	ne
Vykreslení spojnicové grafu	ano	ano	ne
Javascriptové události se mohou vázat k vykresleným elementům	ne	ano	ano
Ovlivňování barev grafu pomocí CSS	ne	ano	ano
Pro základní kreslení není nutné používat Javascript	ne	ano	ano
Responzivnost	ne	ano	ano

Tabulka 2.2: Porovnání technologií pro vizualizaci dat.

Graf 2.1 zobrazuje podporu technologií v jednotlivých prohlížečích. HTML a CSS nebudeme uvažovat, protože ty jsou podporovány všemi dnes používanými prohlížeči. Mohli bychom zobrazit podporu CSS3, ale ta se liší podle jednotlivých druhů značek a nemůžeme přesně určit, které značky bychom mohli při kreslení grafu potřebovat a které ne.



Graf 2.1: Porovnání podpory jednotlivých prohlížečů [32] a [33]

V dnešní době podporuje canvas i SVG většina používaných prohlížečů. Obě technologie nepodporuje například Internet Explorer verze 8 a starší. Tyto verze ale podle [34] v březnu 2016 používalo pouze 0.3% všech prohlížečů. Pokud tedy nepotřebujeme stoprocentní podporu všech prohlížečů, jsou technologie canvas a SVG zajímavou variantou.

3 Rešerše existujících pluginů

V Angularu již existuje několik modulů pro zobrazování dat. Zde si některé z nich projdeme a shrneme si jejich klady a zápory. Pro porovnání jsme si vybrali tyto moduly:

- tc-angular-chartjs
- angular-charts
- angularjs-nvd3-directives
- angular-nvd3

3.1 Tc angular chartjs

Modul tc-angular-chartjs [35] je skupina Angular direktiv využívajících knihovnu chartjs [26]. Tc-angular-chartjs nám tuto knihovnu pouze obaluje direktivami, abychom ji mohli snadno zapojit do našeho projektu. To je velká výhoda, protože kdybychom použili přímo chartjs, museli bychom si direktivy vytvářet ručně. Takto na jednom řádku vytvoříme direktivu, které v první parametru můžeme předat veškeré nastavení a v druhém data. Nemusíme se tak starat o náročné zapojování do projektu. Celou ukázku použití direktivy zobrazuje Kód 3.1. Přípravu dat pro tuto direktivu vidíme v ukázce Kód 3.2.

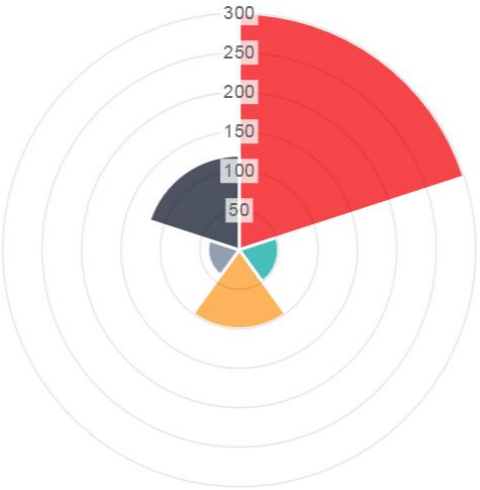
```
<canvas tc-chartjs-pie chart-data="chartData" chart-
options="{segmentShowStroke: false}" width="25" height="25"></canvas>
```

Kód 3.1: Použití direktivy tc-angular-chartjs pro koláčový graf

```
$scope.chartData = [
  { value: 10, color: "#ff0000" },{ value: 25, color: "#ff9999" },
  { value: 50, color: "#ccff99" },{ value: 12, color: "#cc6699"}
];
```

Kód 3.2: Vytvoření dat pro direktivu tc-angular-chartjs

Tc-angular-chartjs zobrazuje data pomocí HTML5 canvasu. Canvas můžeme chápat jako kreslící plátno, do kterého můžeme kreslit pomocí čar a křivek. Samotné kreslení probíhá tak, že si pomocí Javascriptu od HTML elementu canvas (který vložíme do stránky) vyžádáme objekt a do něj pomocí metod kreslíme. Více je toto téma rozepsáno v Kapitole 2.5 Vizualizace dat. Obrázek 3.1 zobrazuje příklad koláčového grafu.



Obrázek 3.1: Ukázka použití Tc-angular-chartjs pro polární graf

Pro kreslení grafu můžeme použít direktivy zobrazené v následujícím seznamu. Jak vidíme, můžeme kreslit velké množství grafů. I to by mohl být důvod, proč si ke kreslení vybrat právě tuto knihovnu.

- tc-chartjs-line - spojnicový graf
- tc-chartjs-bar - sloupcový graf
- tc-chartjs-radar - paprskový graf
- tc-chartjs-polararea - polární graf
- tc-chartjs-pie - koláčový graf
- tc-chartjs-doughnut - prstencový graf

Výhodou tc-angular-chartjs je velké množství grafů, které můžeme kreslit. Další výhodou je její snadné použití. Nevýhodou je, že při kreslení do canvasu si nemůžeme pomocí CSS tříd měnit barvu jednotlivých částí grafu. U této knihovny také chybí popisky bodů (sloupců, výsečí...), které by se mohli zobrazit, když nad ně uživatel najede kurzorem myši. To považuji za ohromnou chybu, protože v knihovně chartjs, ze které tyto direktivy vychází, je možné popisek zobrazit. Poslední nevýhodou je, že nemůžeme zobrazit pokročilé grafy, jako je například skupinový sloupcový graf.

3.2 Angular charts

Další možností, jak v Angularu kreslit grafy, je použití modulu angular-charts [36]. Ke kreslení grafů využívá knihovnu D3 [37]. Tyto grafy obsahují popisky, které se dají zobrazit například při najetí nad bod grafu. Další jejich předností je, že je možné do nich vložit funkce, které se vykonají při nějaké události. Příkladem takové události může být najetí kurzorem myši nad bod, nebo kliknutí na bod grafu.

Angular-charts může zobrazit tyto typy grafů:

- Pie - koláčový graf
- Bar - sloupcový graf
- Line - spojnicový graf
- Point - graf, ve kterém jsou vyznačeny pouze body
- Area - spojnicový graf, vykreslující plochy pod čarou

Použití modulu je už o něco náročnější. Velké množství nastavení nemá výchozí hodnotu, a tak musíme vytvořit rozsáhlý objekt s nastavením. Kromě toho je znát, že první graf, který autor vytvořil, byl spojnicový. Data se totiž zadávají ve formátu pro spojnicový graf, což působí u ostatních typů grafů velmi nepřehledně. Celé použití koláčového grafu si ukážeme v následující ukázce Kód 3.3. Kód 3.4 nám pak ukazuje, jak vytvořit data pro daný graf.

```
<div ac-chart="pie" ac-data="data" ac-config="config" id='chart'
class='chart'></div>
```

Kód 3.3: Použití modulu angular-charts pro koláčový graf

```
$scope.data = { data : [
  { x : "Žáci", y: [100] }, { x : "Učitelé", y: [300] },
  { x : "Pracovníci", y: [351] }, { x : "Ostatní", y: [54]}
]}
```

Kód 3.4: Vytvoření dat pro koláčový graf

Další velkou nevýhodou Angular charts je nepřehledná a také neúplná dokumentace. V dokumentaci například neexistuje ukázka, jak vytvořit výše použitý koláčový graf a jak ho nastavit. Abychom zjistili, jak vytvořit koláčový graf, musíme nahlédnout do zdrojových kódů grafu. Ty jsou ale v jednom jediném souboru, a tak se v nich velmi špatně orientuje. Závěrem nám Obrázek 3.2 ukáže použití knihovny na koláčovém grafu.

Obrázek 3.2: Ukázka použití Angular charts pro koláčový graf

3.3 Angularjs nvd3 directives

Angularjs-nvd3-directives [38] je skupina direktiv využívajících knihovny NVD3 [39] a D3 [37]. Při vytváření grafů je možné si vybrat z velmi rozsáhlé skupiny grafů. Také je možné zobrazit pokročilé grafy, jako je například skupinový sloupcový graf. V angularjs-nvd3-directives můžeme zobrazit všechny základní grafy:

- spojnicový
- sloupcový (skupinový)
- koláčový
- kombinace spojnicového a sloupcového grafu
- bodový graf
- a mnoho dalších

Velkou nevýhodu těchto direktiv je, že se obtížně používají. Některým direktivám nestačí pouze předat data, která mají vykreslit. Místo toho musíme vytvořit funkce, které tato data vrací. Vše si opět ukážeme na příkladu. V ukázce Kód 3.5 vytváříme přípravu funkcí, které budou vracet data. Kód 3.6 již ukazuje samotné použití direktivy pro koláčový graf.

```
$scope.pieData = [{ key: 'Jablek', y: 12}, { key: 'Hrušek', y: 51 }];

/* funkce vrací data na x a y souřadnicích */
$scope.xFunction = function() {
 return function(d) {
    return d.key;
 };
};

$scope.yFunction = function() {
 return function(d){
   return d.y;
 };
};
```

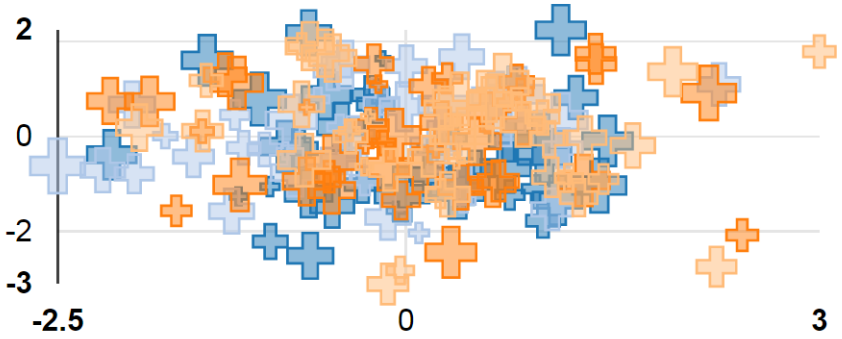
Kód 3.5: Příprava dat pro koláčový graf

```
<nvd3-pie-chart id="chart-id" data="pieData" width="80" height="80"
x="xFunction()"    y="yFunction()"></nvd3-pie-chart>
```

Kód 3.6: Použití direktivy pro koláčový graf

Pokud chceme vytvářet pokročilé grafy s velmi specifickými požadavky, pravděpodobně se rozhodneme pro tyto direktivy. Musíme ale uvážit, zda skutečně potřebujeme používat takto pokročilé grafy. Při použití těchto grafů musí prohlížeč stáhnout nejen Angularjs-nvd3-directives, ale i knihovny NVD3 a D3. Výhodou těchto

direktiv je jejich komunita. V tuto chvíli mají na Githubu přes 11 stovek hvězd a 313 kopií repozitáře (fork). Podle Githubu tak mají největší komunitu ze všech porovnávaných pluginů. Obrázek 3.3 zobrazuje příklad bodového grafu.



Obrázek 3.3: Ukázka bodové grafu v Angularjs-nvd3-directives

3.4 Angular nvd3

Angular-nvd3 [40] je obdoba direktiv Angularjs-nvd3-directives. Obě jsou založeny na knihovnách NVD3 [39] a D3 [37]. Angular-nvd3 můžeme považovat za jinou implementaci knihoven NVD3 [39] a D3 [37] do Angularu. Zobrazuje tedy podobné typy grafů, jako Angularjs-nvd3-directives. Angular-nvd3 si uvádíme pro porovnání dvou rozdílných implementací připojení těch samých knihoven do Angularu. Kód 3.8 ukazuje přípravu dat pro graf. V ukázce Kód 3.7 už vidíme samotné použití dané direktivy:

```
<nvd3 options="pieOptions" data="pieData"></nvd3>
```

Kód 3.7: Použití direktivy pro koláčový graf

```
$scope.pieData = [{ key: 'Jablek', y: 12}, { key: 'Hrušek', y: 51 }];

$scope.pieOptions = {
 chart: {
   type: 'pieChart',
   height: 500,
   x: function(d) { return d.key; },
   y: function(d) { return d.y; },
   showLabels: true
 }
};
```

Kód 3.8: Příprava dat pro koláčový graf

Snažíme se zde zobrazit ta samá data, jako u příkladu použití Angularjs-nvd3-directives. Rozdíl je ale v nastavení. Zatímco Angularjs-nvd3-directives velkou část nastavení předávala pomocí atributů direktivy, v Angular-nvd3 se snažíme předat vše v jednom

objektu. Předávání všech nastavení v jednom objektu je spíše starší vzor hodně používaný v jQuery pluginech. Angular se snaží více zaměřovat právě na předávání pomocí atributů. Na atribut můžeme například navázat různé kontrolní funkce, které nám budou hlásit změnu daného atributu. Když ale vše posíláme v jednom objektu, musíme si sami ručně dopsat kontrolu, u kterého atributu došlo ke změně.

Výhody a nevýhody této knihovny jsou podobné knihovně Angularjs-nvd3-directives. Oproti ní má ale Angular-nvd3 mnohem menší komunitu na Githubu a předávání veškerého nastavení přímo v objektu. S touto knihovnou se špatně pracuje a velkou část kódu píšeme do Javascriptu. Při porovnání implementace připojení knihoven NVD3 [39] a D3 [37] rozhodně vyhrává Angularjs-nvd3-directives. Má větší komunitu a více využívá principů Angularu.

3.5 Závěrečné porovnání modulů

Porovnání všech modulů ukazuje Tabulka 3.1. Modul pro vizualizaci vybereme podle toho, jak složité grafy chceme zobrazit. Pravděpodobně na zobrazení dat nepoužijeme direktivy tc-angular-chartjs. Většina špatných vlastností těchto direktiv vychází z použití canvasu. Podle mého názoru je v dnešní době možné canvas nahradit SVG a HTML. Pokud máme malý projekt, s jednoduchými grafy, pravděpodobně se rozhodneme pro knihovnu Angular-charts. Potřebujeme-li kreslit náročnější grafy a záleží nám na komunitě kolem knihovny, zvolíme Angularjs-nvd3-dir.

	tc-angular-chartjs	**angular-charts**	**angularjs-nvd3-dir**	**angular-nvd3**
Počet typů grafů	6	5	15+	15+
Způsob vykreslení	HTML5 canvas	HTML, SVG, CSS	HTML, SVG, CSS	HTML, SVG, CSS
Responsibilita	ano	ano	ano	ano
Animace	ano	ano	ano	ano
Zobrazení popisků	ne	ano	ano	ano
Možnost nastavení barev pomocí CSS	ne	ano	ano	ano
Kombinace typů grafů	ne	ne	ano	ano

Tabulka 3.1: Porovnání knihoven pro zobrazování dat v Angularu

4 Server a komunikace s klientem

V rámci této práce bylo potřeba vytvořit klientskou a serverovou aplikaci. Klientská aplikace je napsaná v Angularu a věnuje se jí následující kapitola. V tuto chvíli nám stačí vědět, že dashboard je klientská aplikace, a že po serverové aplikaci požaduje data. V této kapitole si vysvětlíme, jak funguje serverová aplikace a komunikace se serverem.

Cílem práce je vytvořit klientskou aplikaci a zabývat se vizualizací dat. Proto se v práci nezabýváme zabezpečením a data považujeme za veřejná. Pokud bychom však chtěli data zabezpečit, použijeme pro komunikaci protokol https a na straně serveru budeme pomocí SESSION kontrolovat, zda je uživatel přihlášen.

Serverová aplikace je napsaná v jazyce PHP a pro ukládání dat využívá databázi MySQL . Tato aplikace zajišťuje uložení a aktualizaci dat, která poskytuje widgetům. Tato data obsahují:

- Konfigurační část
- Datovou část

Konfigurační část obsahuje informace o vlastnostech widgetu. Datová část obsahuje vlastní data, která má widget zobrazit. Zpravidla se jedná o statistická data, která mají být zobrazena grafovými widgety. Touto cestou si můžeme poslat nejen data, ale i další nastavení widgetu. Data si ve widgetu můžeme také obnovit. Můžeme například:

- nastavit widgetu, aby se každých 5 sekund obnovoval
- plynule měnit čas do další obnovy v závislosti na rychlosti změny dat

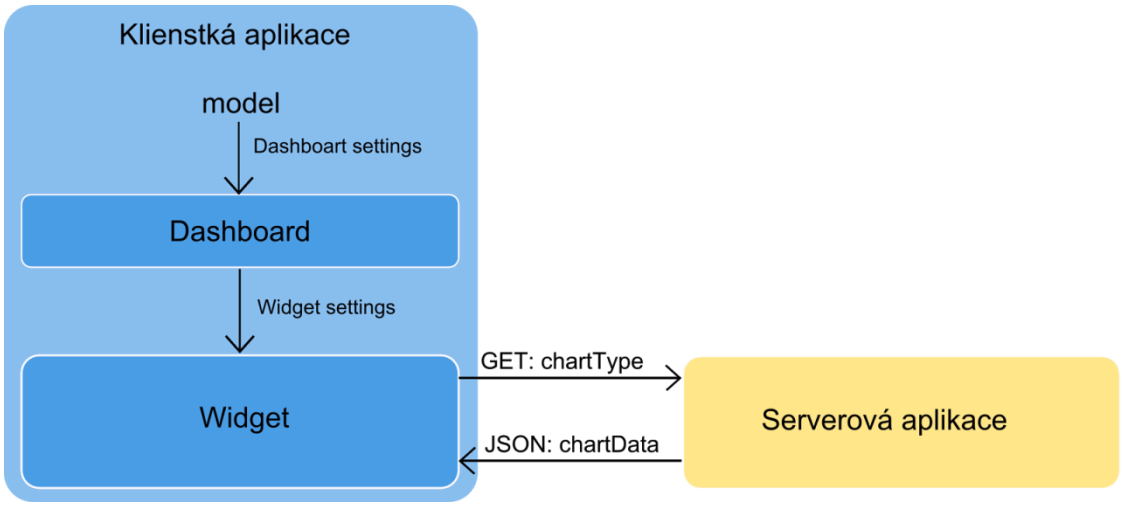
Plynulou změnu do další obnovy můžeme použít například v případě, že se data na serveru začala rychleji měnit (zkrátíme periodu), nebo že data jsou dlouhodobě neměnná (prodloužíme periodu). Tyto závěry se vytvářejí nad daty, ke kterým má přístup právě serverová aplikace, takže je vhodné, aby mohla toto nastavení změnit.

Abychom mohli pochopit, jak a na jaké vrstvě komunikuje klientská a serverová aplikace, musíme si něco napsat o nastavení klientské aplikace. Nastavením myslíme počáteční konfiguraci aplikace. V ní nastavujeme, jaké widgety se mají v dashboardu zobrazit. U jednotlivých widgetů pak uvádíme jejich název, jakého jsou typu nebo na jaké adrese si mají stáhnout data která mají zobrazit. Uvedeme si příklad. Chceme vytvořit velmi jednoduchý dashboard jedním grafovým widgetem. Graf by nám měl zobrazovat data uložená na adrese *kukral.eu/?type=prehled-o-stavu-studentu*. Nastavení tohoto dashboardu vidíme v ukázce Kód 4.1. Nastavení dashboardu obsahuje pole widgetů, které má zobrazit. V našem případě pole obsahuje pouze jeden widget *barchartNg*, který označuje sloupcový graf.

```
dashboard: {
 name: 'Jednoduchý dashboard',
 widgets: [{
  type: 'barchartNg',
  settings: {
    title: 'Sloupcový graf',
    relativeUrl: '/?type=prehled-o-stavu-studentu'
   }
} /* zde by mohli být další widgety */]};
```

Kód 4.1: ukázka nastavení dashboardu s jedním widgetem

Nastavení v ukázce Kód 4.1 je předáno dashboardu. Ten si z nastavení uloží svůj název a poté začne vytvářet jednotlivé widgety. Při vytvoření widgetu je mu dashboardem předáno jeho počáteční nastavení. V naší ukázce se widgetu předává jeho název *Sloupcový graf* a adresa */prehled-o-stavu-studentu*. Jakmile je widget vytvořen, tak už může fungovat zcela samostatně. K tomu vede i implementace jednotlivých widgetů, kde je oddělen *$scope* dashboardu od widgetu. Oddělení *$scope* [1] nám plní v tomto případě roli oddělení dat, která má přístupná dashboard a která má přístupná widget. Nedochází tak k jejich prolínání. Dashboard a widget spolu sdílejí data pouze pomocí parametrů předaných při vytváření widgetu. Oddělením *$scope* a jednosměrnou komunikací pomocí parametrů jsme dosáhli jednosměrného data bindingu (více v Kapitole 2.4.2 Obousměrný data binding). Widget může kdykoliv reagovat na změnu parametru pomocí speciálních kontrolních funkcí Angularu. Může také reagovat na události, poslané z dashboardu. Widget je na dashboardu zcela nezávislý, ale může od něj kdykoliv přijmout další nastavení. Kompletní proces předávání nastavení ukazuje Obrázek 4.1.



Obrázek 4.1: Komunikace klientské a serverové aplikace

Nyní si napíšeme, proč bylo nutné proces předávání nastavení tak důkladně probrat a jak souvisí s komunikací obou aplikací. Díky samostatnosti jednotlivých widgetů dashboard nemusí zjišťovat, zda widget potřebuje nějaká další data ze serverové aplikace. Celou tuto komunikaci řešíme až na úrovni widgetu. Podrobně si popíšeme komunikaci klientské a serverové aplikace, kterou znázorňuje Obrázek 4.1. Widget po vytvoření pošle GET požadavek serverové aplikaci. V parametru *type* serverové aplikaci předá, o jaká data žádá. Komunikaci si opět ukážeme na příkladu z odstavce výše. V tomto příkladu žádáme o data o Přehledu stavu studentů. Do parametru *type* tak vložíme */?type=prehled-o-stavu-studentu*. Serverová aplikace si přečte obsah hodnoty *type*. Přečte si, že jde o *prehled-o-stavu-studentu* a vyhledá v databázi data pro tuto statistiku. Tato data pak pošle zpět klienstké aplikaci. Příklad dat pro sloupcový graf vidíme v ukázce Kód 4.2.

```
{
 "bars": [
   [{"data":134, "name": "přihláš."},{"data":94, "name": "přijat."}, …
další data ...]
 ],
 "unit":"k",  "grid":"1"
}
```

Kód 4.2: Ukázka dat, která vrací serverová aplikace

V poli bars (sloupce) máme uložená všechna data o sloupcích, která se mají v grafu vykreslit. Také ale vidíme další předané hodnoty, jako je hodnota *unit*. V té serverová aplikace říká, že poslané hodnoty jsou v řádech tisíců (k = tisíc). V těchto dodatečných datech můžeme widgetu poslat jakákoliv nastavení widgetu. Příklad takového nastavení může být už výše zmiňovaná perioda, za jak dlouho si má widget zaktualizovat svá data.

Jak z příkladu vidíme, klientská aplikace je tak plně oddělena od serverové a komunikace probíhá pouze pomocí GET požadavků. Výhoda tohoto oddělení spočívá v tom, že snadno můžeme vyměnit serverovou aplikaci za jinou. Stačí, aby nová aplikace reagovala na stejné požadavky a vracela správný JSON výstup. Serverová aplikace tak může být velmi rozsáhlá, a nebo se může jednat jen o složku s JSON soubory obsahující toto nastavení. Serverová aplikace nemusí být napsána ani v jazyce PHP, ale můžeme použít i Javascript a node.js, nebo .NET.

Aktuální verze serverové aplikace je tvořena MVC architekturou. Data jsou uložena ve formátu JSON v relační databázi. Z důvodu snadného testování, vývoje a vyzkoušení aplikace jsme však přidali druhou alternativní databázovou vrstvu, která data ukládá do obyčejných textových souborů.

Serverová aplikace je vytvořena v PHP s využitím MVC architektury. Nevyužíváme speciálního backendového frameworku. Je tedy založena pouze na jazyce PHP. Na frontendu se však využívá několik frameworků:

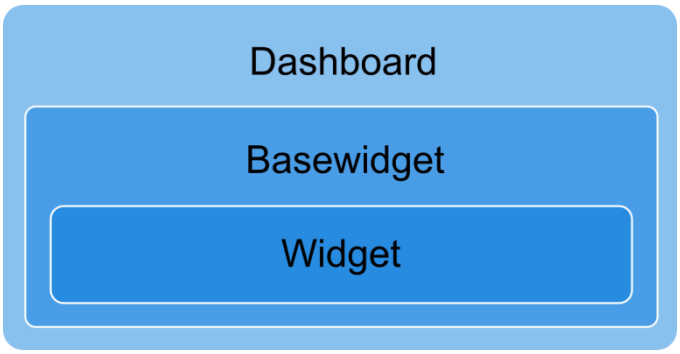
- Angular [41]
- Bootstrap [12]

Kromě části, která zajišťuje komunikaci s widgety, obsahuje aplikace ještě další funkcionalitu. Jedná se o možnost přímo měnit nastavení jednotlivých widgetů z prohlížeče. Aby byla tato část uživatelsky přívětivá, byla navrhnuta také v Angularu. Uživatel si může jednoduše vybrat, které nastavení chce změnit. Toto nastavení pak může libovolně editovat. Nastavení se při změně automaticky ukládá na server. Nepotřebujeme tedy tlačítko *uložit*. K automatickému ukládání se používá technologie AJAX, která poskytuje komunikaci se serverem bez nutnosti obnovit stránku. Díky automatickému ukládání se uživateli nestane, že by zapomněl uložit nové nastavení widgetu. Zároveň se zobrazí varování, pokud uživatel na stránku vstoupí bez povoleného JavaScriptu. Ukázku této aplikace vidíme zobrazuje Obrázek 4.2.

Obrázek 4.2: Ukázka serverové aplikace na změnu nastavení v prohlížeči

5 Modul dashboard

V této práci vytváříme komponentu pro dashboard. Do něho zapracováváme několik widgetů. Dashboard mimo jiné umí na žádost uživatele instance widgetu dynamicky přidávat a mazat. Také zajišťuje správné delegování událostí. Samotné widgety je pak možné použít v dashboardu, ale mohou se použít i zcela samostatně na jiné stránce.



Obrázek 5.1: Ukázka zanoření base widgetu

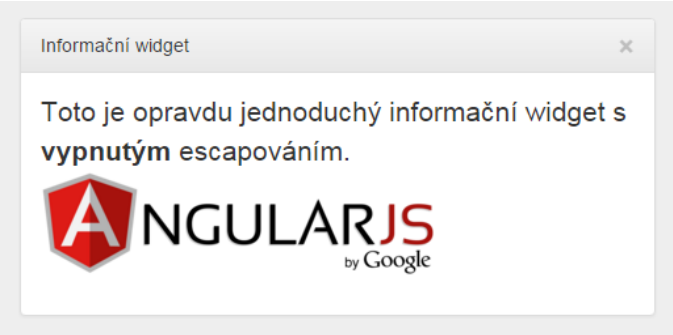
Jak ukazuje Obrázek 5.1, k usnadnění implementace nového widgetu slouží baseWidget, který může obalovat námi implementovaný widget. Kromě toho, že sjednocuje funkcionalitu všech widgetů při pohledu z vnějšku, sjednocuje vzhled a rozšiřuje widget o další funkce. Dochází zde i ke správnému předávání událostí. Správné předávání událostí je v každé aplikaci v Angularu velmi důležité, a proto si ho krátce nastíníme na konkrétním příkladu. Chceme vytvořit do dashboardu widget, který se periodicky v určitých časových intervalech ptá serveru na data a poté se aktualizuje. V Angularu nám periodické spouštění určité funkce (v našem případě funkce na obnovu dat) zajistí služba *$timeout*. Ta zabezpečuje podobné chování, jako funkce *setTimeout* z Javascriptu. Oproti této metodě služba *$timeout* obnoví po svém vykonání *$scope*, takže se nemusíme o jeho obnovení starat spuštěním funkce *$apply*. Služba *$timeout* vrací *promise* a v daném čase se provede obnova dat widgetu. Ale co když se mezi tím, než se *promise* provede, celý widget odstraní? Podle [42] *promise* zůstane i nadále aktivní a může nám v programu tvořit chyby. Proto je potřeba na tyto události správně reagovat a ihned odstraňovat závislosti, které již nejsou potřeba. Ukázku, jak tento problém správně řešit, můžeme najít v ukázkovém widgetu s hodinami.

V rámci práce byly implementovány 3 základní widgety a 3 widgety pro grafy. Nad rámec zadání diplomové práce byly implementovány ještě sloupcové grafy v jQuery a Reactu. Ty byly implementovány proto, abychom mohli vzájemně porovnat obě technologie s Angularem. Základní widgety slouží k tomu, abychom na nich rychle pochopili, jak dashboard pracuje a mohli si implementovat svůj vlastní widget.

Patří mezi ně:

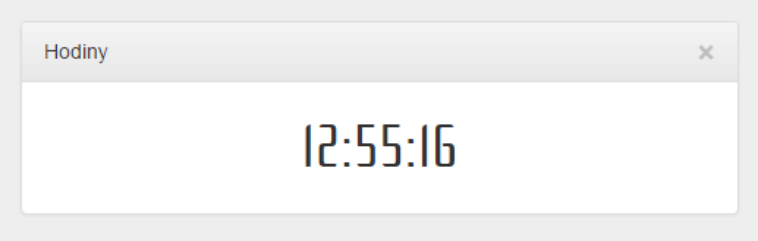
- informační widget
- hodiny
- hightchart widget

Informační widget slouží k prostému zobrazení textu. Můžeme si v jeho nastavení vypnout escapování, které Angular dělá vždy automaticky. K vypnutí escapování je použita metoda z Angularu *toTrustedHTML*. Poté do widgetu můžeme vkládat přímo HTML kód a můžeme tak s textem více pracovat, nebo přidávat obrázky. Pokud escapování vypneme, musíme pak sami zajistit kontrolu, co uživatel do widgetu vkládá. Může do něj totiž vložit např. i Javascript, který by nám mohl aplikaci poškodit. Příkladem informačního widgetu je Obrázek 5.2.



Obrázek 5.2: Informační widget

Widget s hodinami, který ukazuje Obrázek 5.3, slouží k tomu, že zobrazí hodiny v námi zadaném formátu a poté je každou sekundu aktualizuje. Periodické aktualizování je realizováno službou *$timeout*, kterou jsme si popsali už výše. Pokud ale dojde k odstranění widgetu, před jeho odstraněním se obslouží událost odstranění widgetu *$destroy*, a *promise* vytvořený službou *$timeout* se odstraní. Dochází tak ke správnému odstranění widgetu spolu s jeho závislostmi, a tak může tento widget posloužit, jako ukázka reakce na události.



Obrázek 5.3: Widget s hodinami

Hightchart widget slouží ke zobrazení spojnicových a sloupcových grafů s využitím knihovny Highchart [43]. Mezi zobrazením sloupcového a spojnicového grafu můžeme

přepínat přímo v zobrazeném widgetu. Graf načítá data ze serverové aplikace. Ke komunikaci mezi serverem a widgetem slouží služba *ChartResource*. Ta zajišťuje odeslání AJAXového požadavku metodou GET na předanou adresu, zpracování požadavku a vrácení výsledných dat. Data se z JSON řetězce převádějí na Javascriptový objekt, se kterým se dá lépe pracovat. Díky službě z Angularu *$resource*, kterou zde také využíváme, je kód služby *ChartResource* krátký a přehledný. Navíc jelikož jde o službu, můžeme ji implementovat jednou a pak ji využívat v různých částech aplikace. Chceme-li obdobné volání napsat například v jQuery, musíme funkci pro vytvoření požadavku vytvářet pořád znovu. Nic nás totiž nevede k tomu, abychom vytvořili a snadno použili již vytvořený kód. Můžeme si sice ručně vytvořit podobnou funkcionalitu, jakou je služba v Angularu, ale to se už snažíme vytvořit v jQuery něco, co Angular implicitně obsahuje a podporuje. Toto je další výhodou Angularu oproti jQuery. Ukázku volání AJAXových požadavků v jQuery a v Angularu vidíme v ukázkách Kód 5.1 a Kód 5.2.

```
// Příklad volání služby pro načtení dat v Angularu
var graphData = ChartResource.send(relativeUrl).get();
graphData.$promise.then(addChart);
```

Kód 5.1: Volání služby pro vytvoření ajaxového požadavku v Angularu

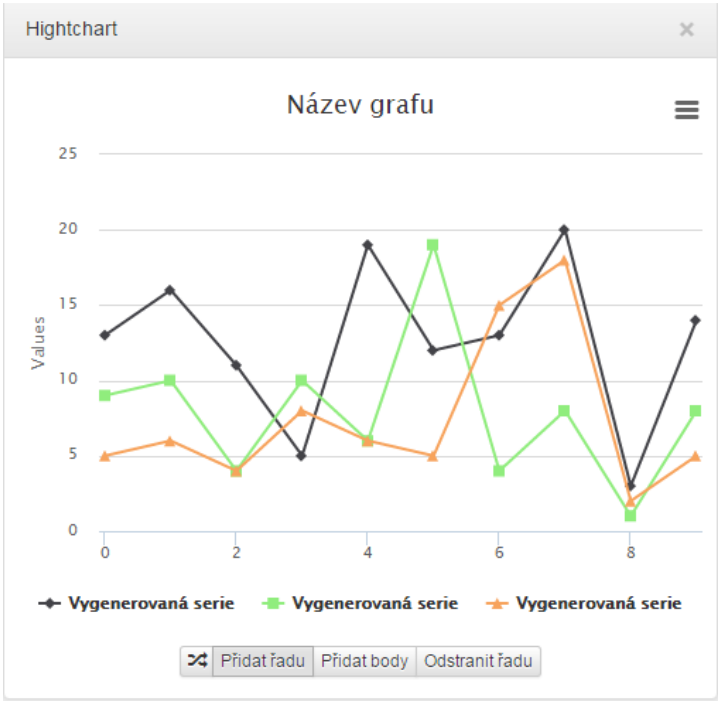
```
// Příklad načtení dat ze serverové aplikace v jQuery
$.ajax({
  url: wwwRoot + relativeUrl,
  dataType: 'json',
  type: 'GET',
  success: function(result){
     //zavolání funkce addChart, co se má vykonat po dokončení požadavku
  }
});
```

Kód 5.2: Vytvoření ajaxového požadavku v jQuery

S daty se také dá pracovat přímo v grafickém rozhraní widgetu. K tomu slouží obslužná tlačítka widgetu. Účelem těchto tlačítek je primárně demonstrace funkcionality. Díky Angularu se dají tyto obslužné funkce napsat velmi krátké a přehledné a obousměrný data binding (viz. Kapitola 2.4.2 Obousměrný data binding) nám zajistí převod dat mezi modelem a viewem. Pomocí obslužných tlačítek můžeme:

- vygenerovat a přidat novou řadu
- přidat nové body do existující řady
- odstranit náhodnou řadu
- přepínat zobrazení sloupcového a spojnicového grafu

Tento widget celkově slouží, jako ukázka, jak načítat data ze serverové aplikace, a jak s nimi pracovat. Ukazuje také, jak se dá snadno zapojit velmi složitá existující knihovna v jQuery do aplikace psané v Angularu. Ukázkou widgetu je Obrázek 5.4.



Obrázek 5.4: Widget s hightchart grafy

6 Grafové widgety

V minulé kapitole jsme nastínili základní widgety, které slouží k tomu, aby demonstrovaly různé použití dashboardu. Nyní přistoupíme k dalším třem widgetům, které již budou řešit pouze zobrazování dat. Jedná se o:

- koláčový graf
- sloupcový graf
- spojnicový graf

Pro každý grafový widget používáme jiný druh vykreslení:

- SVG pro koláčový graf
- HTML canvas pro sloupcový graf
- HTML a CSS3 pro sloupcový graf

Spojnicové a sloupcové grafy jsou přepisovány z existujících jQuery pluginů a koláčový graf je vytvořen přímo bez přepsání. V Kapitole 2.4.7 Konverze jQuery pluginu do Angularu se věnujeme oběma přístupům vytváření pluginů a popisujeme jejich výhody a nevýhody. V této kapitole si nastíníme náročnost přepsání pluginu z jQuery do Angularu a porovnáme si rozdíly vizualizace dat.

Každý námi vytvořený grafový widget má dvě direktivy:

- hlavní (vstupní) - zajišťuje komunikaci a přidává další funkce
- grafová - stará se pouze o vykreslení grafu

Je tomu tak z důvodu lepší dekompozice kódu. Díky rozdělení na dva samostatné celky se můžeme věnovat každé části widgetu zvlášť. Hlavní direktiva zpravidla řeší komunikaci se serverovou aplikací. Upravuje přijatá data a přidává ke grafu další funkce, jako například popisky grafu nebo zobrazení mřížky na pozadí. Grafovou direktivu pouze volá a předává jí její nastavení. Grafová direktiva se již stará o konkrétní vykreslení sloupců, křivek nebo výsečí. Jelikož vytváříme direktivy pro zobrazování dat, nebylo potřeba vytvářet mezi widgetem a dashboardem obousměrnou komunikaci. Widget si pouze nechá předat počáteční nastavení od dashboardu a poté již funguje samostatně. Nastavení obsahuje mimo jiné i URL adresu serverové aplikace, ze které si widget načte data, která má zobrazit.

6.1 Koláčový graf

Jako první si popíšeme koláčový graf. Před tím, než si začneme popisovat samotnou direktivu a obslužné funkce pro vykreslení grafu, musíme si nejdříve něco napsat o počítání souřadnic. Teprve poté můžeme přejít k samotnému kreslení. Ze serverové aplikace dostaneme pouze pole čísel reprezentující hodnoty jednotlivých výsečí, jako v ukázce Kód 6.1, ze kterých se dopočítávají poměry a velikosti výsečí.

```
[2643, 22165, 5118, 785, 510, 572]
```

Kód 6.1: Ukázka dat k vykreslení koláčového grafu.

Jelikož v našem widgetu kreslíme pomocí SVG, pro vykreslení výsečí používáme elementy *path* s atributy *d*. Díky svým funkcím je *path* element vhodný právě pro kreslení výsečí. Můžeme v něm ale nakreslit i kruh, trojúhelník, čtverec či jiný obrazec. Podle [44] můžeme v elementu *path* s atributem *d* použít pro kreslení tyto funkce:

- M (moveto) - přesun na souřadnice
- L (lineto) - nakreslit přímku
- C (curveto) - nakreslí křivku
- A (arc) - oblouk
- Z (closepath) - nakreslí přímku z poslední zadané souřadnice na počáteční

V naší práci každý element *path* zastupuje konkrétní výseč a v atributu *d* pak nastavíme parametry této výseče. V ukázce Kód 6.2 vidíme příklad vykreslení několika výsečí v koláčovém grafu. Abychom mohli graf pomocí elementů *path* vykreslit, musíme vypočítat souřadnice jednotlivých výsečí.

```
<svg>
 <!-- Vykreslení první výseče -->
 <path fill="#468966" d="M200,200  L380,200  A180,180 0 0,1 355,290 z">
 </path>
 <!-- Vykreslení druhé výseče -->
 <path fill="#FFB03B" d="M200,200  L234,23  A180,180 0 0,1 368,135 z">
 </path>
      … další elementy path …
</svg>
```

Kód 6.2: Příklad vykreslení několika výsečí ve formátu SVG

6.1.1 Výpočet souřadnic

Nyní si nastíníme postup, jak z dat vypočítat souřadnice výsečí. Poté, co dostaneme data, je nejdříve musíme přepočítat na úhly. U každého prvku musíme vědět, jak velký úhel v našem grafu zastupuje. Tento úhel dostaneme podle Vzorce 6.1.

$$\alpha = \frac{m_i * 360}{\sum_{i=0}^{n} m_i} [^\circ]$$

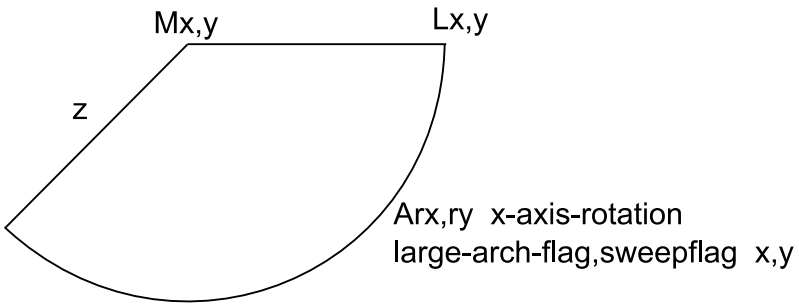
Vzorec 6.1: Určení úhlu kruhové výseče

Význam použitých symbolů:

α ... úhel kruhové výseče
n ... počet všech hodnot v datech (odpovídá počtu kruhových výsečí)
i ... index (pořadové číslo) dané hodnoty
m_i ... hodnota daného prvku, kterou chceme zobrazit v kruhové výseči

Jakmile máme spočítané úhly, můžeme se pustit do samotného počítání souřadnic. Souřadnice pro vykreslení výseče jsou zapsány v parametru *d*, který má dle W3C [44] formát:

Mx,y Lx,y Arx,ry x-axis-rotation large-arch-flag,sweepflag x,y z, jak ukazuje Obrázek 6.1



Obrázek 6.1: Kreslení kruhové výseče a určení souřadnic

Nyní si vysvětlíme, co celý řetězec znamená:

- Mx,y - přesune kreslící bod na souřadnice x,y (v našem případě střed kružnice).
- Lx,y - nakreslí přímku z aktuálních souřadnic do souřadnic x,y.
- Arx,ry x-axis-rotation large-arch-flag,sweepflag x,y - nakreslí oblouk.
- z - Uzavírá výseč. Nakreslí přímku z počátečního do koncového bodu.

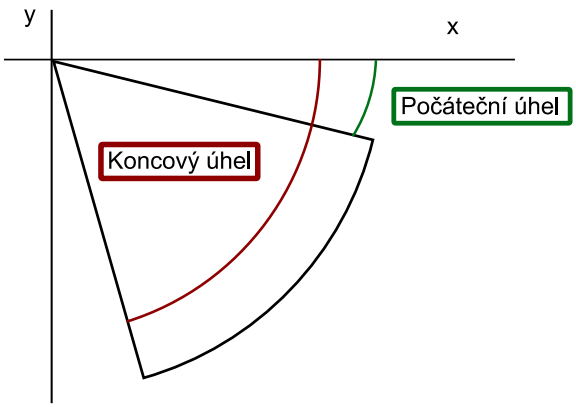
Mx,y přesune kreslící bod na souřadnice x,y v souřadnicovém prostoru. V tomto momentě se ještě žádná křivka nekreslí. V našem případě budou tyto souřadnice středem kruhu, ve kterém budeme u každé výseče začínat. Lx,y poté nakreslí přímku ze středu kruhu na okraj kruhové výseče.

Nyní si popíšeme parametry pro nakreslení oblouku. Jsou to:

- Arx,ry
- X-axis-rotation
- Large-arch-flag
- Sweepflag
- X,y

Arx a ry tvoří eliptický poloměr ve směru x a y. X-axis-rotation znamená, jak moc se bude oblouk otáčet kolem osy x. To se uplatní pouze v momentě, kdy budou mít rx a ry rozdílnou hodnotu, tedy pouze pokud půjde o elipsu. V našem případě budeme kreslit vždy pouze jen kruh, a tak tento parametr nevyužijeme. Large-arch-flag podle [45] může být 0 nebo 1 a nemění vlastnosti úhlu. Pozorováním bylo ale zjištěno, že toto tvrzení je chybné. Tato hodnota rozlišuje, jestli jde o ostrý nebo tupý úhel. Pokud jde o ostrý úhel, musíme ho nastavit na 0, pokud o tupý, tak na 1. Kdybychom nechali hodnotu stále nastavenou na 0 (jak navrhuje [45]), tupé úhly by se nám nevykreslovaly vůbec, nebo by se vykreslovaly chybně. Parametr sweep-flag znamená, v jakém směru chceme vykreslovat oblouk. 0 znamená v protisměru hodinových ručiček, 1 znamená po směru. V našem grafu máme nastavenou hodnotu 1.

Nyní budeme potřebovat určit jednotlivé souřadnice. Střed kruhu Mx,y se rovná součtu poloměru a volného místa kolem grafu. Volné místo nám slouží k vizuálnímu oddělení grafu od jeho widgetu. Dále musíme určit souřadnice Lx,y, tedy souřadnice, ze kterých začínáme kreslit oblouk. Pro zjišťování těchto souřadnic musíme zavést další dvě proměnné. Jsou to počáteční úhel výseče a koncový úhel výseče. Počáteční úhel nám říká, jak bude výseč posunuta oproti počátečnímu stavu (tedy úhlu 0, který odpovídá ose x). Tento úhel používáme k posunutí výseče, aby začínala tam, kde poslední výseč končí. Koncový úhel pak označuje šířku úhlu samotné výseče oproti počátečnímu úhlu. Vše zobrazuje Obrázek 6.2.



Obrázek 6.2: Kreslení kruhové výseče - úhly

Pří vykreslení první výseče je počáteční úhel 0 a koncový úhel se rovná úhlu výseče. U vykreslení dalších výsečí přičítáme k počátečnímu úhlu úhel poslední vykreslené výseče. Koncový úhel je součet počátečního úhlu a úhlu samotné výseče. Počáteční a koncový úhel snadno dopočítáme.

K nakreslení výseče nám ale pouze tyto úhly nestačí. Potřebujeme vypočítat i body na kružnici, které označují počátek a konec výseče. Právě přes tyto body povedeme křivky, vykreslující samotnou výseč. K výpočtu bodů počátku a konce výseče nám pomůže výpočet souřadnic bodu na kružnici. Ten je [46]:

$$x = \cos(t) \qquad y = \sin(t)$$

Vzorec 6.2: Určení bodu na jednotkové kružnici

V tomto vzorci jednotlivé symboly znamenají:

x,y … souřadnice bodu na kružnici
t … úhel daného bodu

Tento vzorec platí pro jednotkovou kružnici v radiánech. My ale potřebujeme vzorec pro kružnici s různým poloměrem ve stupních. Pro přepočet stupňů na radiány použijeme vzorec:

$$t = \frac{\alpha * \pi}{180} \text{ [rad]}$$

Vzorec 6.3: Převod úhlů na radiány

Vzorec 6.3 obsahuje:

t ... úhel kruhové výseče v radiánech
α ... úhel kruhové výseče ve stupních

Abychom ze Vzorce 6.2 vytvořili vzorec bodu na křivce pro obecnou kružnici, vynásobíme obě souřadnice poloměrem a připočítáme polohu středu kružnice. Složením výsledného vzorce a Vzorce 6.3 dostaneme Vzorec 6.4, který vidíme na následující stránce.

$$x = S_x + r * \cos\left(\frac{\alpha * \pi}{180}\right)$$

$$y = S_y + r * \sin\left(\frac{\alpha * \pi}{180}\right)$$

Vzorec 6.4: Určení bodu na kružnici

Nyní si popíšeme symboly v tomto vzorci:

α ... úhel kruhové výseče ve stupních
S_x, S_y … x nebo y souřadnice středu kružnice
r … poloměr kružnice

Nyní už můžeme dopočítat souřadnice bodů počátku a konce oblouku výseče. V ukázce Kód 2.1 vidíme počítání souřadnic v našem grafu.

```
startAngle = endAngle; //vypočítá poč. úhel, pro první výseč je 0
endAngle = startAngle + item.angle; //vypočítá koncový úhel

/* Výpočet bodu počátku oblouku kruhové výseče. PieRadius = poloměr grafu*/
xStart = parseInt(200 + pieRadius*Math.cos(Math.PI*startAngle/180));
yStart = parseInt(200 + pieRadius*Math.sin(Math.PI*startAngle/180));

/* výpočet bodu konce oblouku kruhové výseče */
xEnd = parseInt(200 + pieRadius*Math.cos(Math.PI*endAngle/180));
yEnd = parseInt(200 + pieRadius*Math.sin(Math.PI*endAngle/180));
```

Kód 6.3: Výpočet počátku a konce oblouku kruhové výseče

6.1.2 Vykreslení grafu

Nyní máme vypočítané všechny souřadnice a můžeme se pustit do samotného kreslení výseče. Souřadnice Lx,y (přímka od středu kruhu k oblouku výseče) jsme si určili již v kapitole výše. Jedná se o souřadnice počátku oblouku výseče. Pro vykreslení si určíme další parametry:

- Arx,ry
- x-axis-rotation
- large-arch-flag
- sweepflag
- x,y

Jelikož tvoříme kruh a ne elipsu, jsou obě souřadnice *Arx,ry* rovny poloměru grafu (proměnné *pieRadius*). Parametr *x-axis-rotation* se také uplatňuje pouze u elipsy a je proto 0. Parametr *large-arch-flag* je nastaven na 0, pokud půjde o ostrý úhel (vč. úhlu

180°) a 1 pro tupý úhel. Parametr *sweepflag* je 1, tedy vykreslujeme ve směru hodinový ručiček. Souřadnice *x*,y označují koncový bod oblouku výseče vypočteného výše. V ukázce Kód 6.4 vytváříme výsledný parametr *d*, který nakreslí celou jednu výseč:

```
/* pro výpočet x1 a y1 použije obdobný vzorec s proměnnou startAngle */
x2 = parseInt(200 + pieRadius*Math.cos(Math.PI*endAngle/180));
y2 = parseInt(200 + pieRadius*Math.sin(Math.PI*endAngle/180));

var largeArch = ((endAngle-startAngle > 180) ? 1 : 0);
var d = "M200,200  L" + x1 + "," + y1 + "  A" + pieRadius + "," + pieRadius
+ " 0 " + largeArch + ",1 " + x2 + "," + y2 + " z";
```

Kód 6.4: Vytvoření kompletního parametru *d*

Toto počítání provedeme pro každou výseč v grafu. Tím dostaneme parametry *d* všech výsečí a tedy i elementy *path*. Stačí tyto elementy vypsat do elementu SVG a zobrazí se nám výsledný graf.

Nyní se významně projeví úspora vykreslovacího kódu. Pokud budeme chtít vytvořit elementy *path* v jQuery, budeme muset vytvořit elementy typu *http://www.w3.org/2000/svg* a správně je nastavit a přidat do stránky. Pokud je budeme vytvářet v Angularu, stačí nám v šabloně vypsat tento element a pomocí *ng-direktiv* předat příslušná nastavení. Zde uvádíme pro ukázku Kód 6.5, který slouží k vytvoření jednoho elementu path v jQuery pluginu [29]:

```
/* vytvoření elementu */
var makeSVG = function(tag, attrs, val, title, i, color) {
var $el = $(document.createElementNS('http://www.w3.org/2000/svg', tag));
var $g = $(document.createElementNS('http://www.w3.org/2000/svg', "g"));
var $rep = $('<li><i></i><p></p></li>');
    $rep.find("p").html(title+": " + val);
    $rep.find("i").css({background: color});
    $chart.find("."+$leg.attr("class")).append($rep);
for (var k in attrs){
    $el.attr(k,attrs[k]).attr("data-val", val)
    .attr("data-title",title).attr("id","path"+i).attr("class","path");
    $g.append($el).attr("id","pathCont"+i).attr("class","pathCont");
}
return $g[0];
};

/* přidání elementu do DOM */
var arc = makeSVG("path", {d: d, fill: color}, pieData[i], titles[i], i,
color);
$svg.prepend(arc);
```

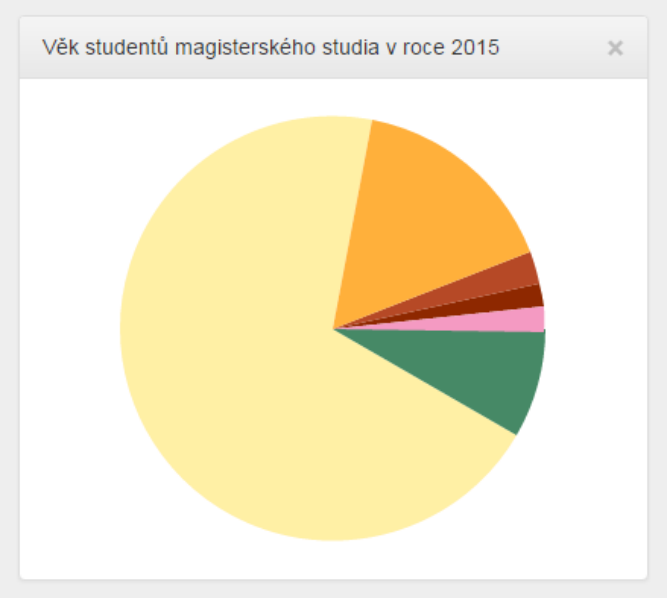
Kód 6.5: Ukázka kódu pro vytvoření path v jQuery

Obdobný kód pak máme zapsaný v Angularu snáze a přehledněji. Porovnání obou kódů je přibližné. Oba kódy vykonávají velmi podobnou, ale nikoliv naprosto stejnou funkcionalitu. Porovnání ukazuje značný rozdíl. V jQuery jde o přibližně 713 znaků (bez komentářů), v Angularu je obdobný kód přibližně na 263 znaků. Úspora kódu je tedy skoro trojnásobná. Navíc ukázkový kód z jQuery je napsaný přímo v programu v Javascriptu. Kód napsaný v našem grafu v Angularu je oddělený v HTML šabloně. To přináší další přehlednost, protože soubor s šablonou je ve formátu HTML, a tak se nám ve vývojovém prostředí zvýrazňuje syntaxe výsledného HTML. Také vizuálně vidíme, jak výsledný kód bude vypadat. V ukázce Kód 6.6 vidíme vykreslení grafu pomocí šablony. Pole *paths*, použité v ukázce Kód 6.6, je definováno ve struktuře *paths = [{1. výseč},{2. výseč}, ...]*.

```
<path ng-repeat="path in paths" ng-attr-d="{{path.d}}" fill="{{path.color}}"
   stroke="{{path.color}}" stroke-width="{{path.strokeWidth}}"
   ng-style="{{path.style}}" ng-mouseenter="mEnter(path)"
   ng-mouseleave="mLeave(path)" ng-mousemove="mMove($event, path)"></path>
```

Kód 6.6: ukázka kódu pro vytvoření path v Angularu

Poté, co máme vytvořené elementy *path* a správně přidané do elementu SVG, již jen stačí napsat obslužné funkce reagující na události. Náš graf reaguje na příchod kurzoru myši nad danou výseč, opuštění dané výseči a pohyb po dané výseči. Pokud je kurzor nad výsečí, zobrazuje se popisek dané výseče. Ukázku, jak poté výsledný graf vypadá, zobrazuje Obrázek 6.3.



Obrázek 6.3: Ukázka widgetu pro koláčový graf

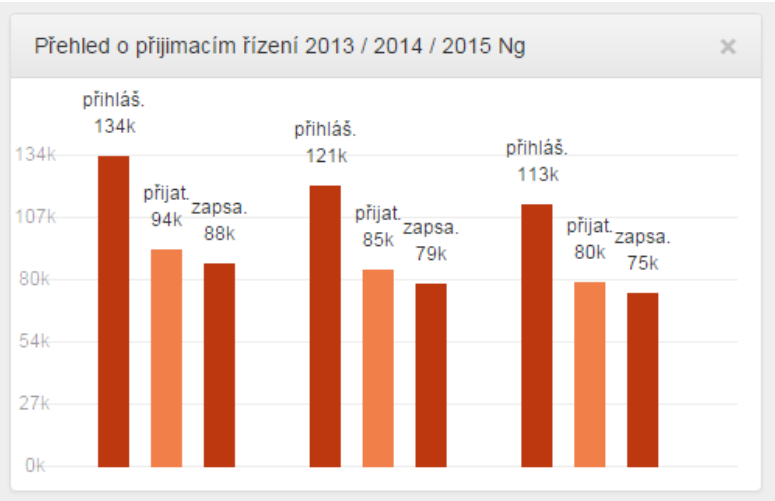
6.2 Sloupcový graf

Dalším grafovým widgetem je sloupcový graf. Jako zobrazovací prvek jsou zde použity samotné HTML elementy. Převážně se jedná o blokové elementy, které zde využíváme ke grafickým účelům. Pro vykreslení grafu jsou použity převážně elementy:

- div
- ul a li (seznamy sloupců a jednotlivé sloupce)
- hr (přímky na pozadí grafu)

Vykreslení grafu pomocí HTML elementů je výhodné, protože se s nimi dobře pracuje, snadno se reaguje na různé události, a k některým animacím se dají používat CSS3 styly. V našem případě je například použit styl Transitions (viz. [47]), který vytváří efekt vysunutí sloupců při načtení grafu. Dalším příkladem použití CSS je zvětšení sloupce při najetí kurzorem myši nad sloupec. Kdybychom graf nedělali přímo v HTML, pravděpodobně bychom museli v Javascriptu pomocí události ošetřit toto najetí myši nad prvek. Takto stačilo pouze přidat do CSS reakci na událost *hover*. Tím se nám obsluha i animace zkrátily na 3 řádky CSS kódu.

Sloupcový graf nezobrazuje pouze samotné sloupce, ale zobrazuje skupiny sloupců. Skupiny nám například umožní lépe porovnat několik statistických údajů v rozmezí více let. Skupiny sloupců nemusí mít stejný počet prvků. Proto je nutné před samotným vykreslením projít data, která chceme zobrazit. Musíme přidělit správnou šířku skupinám i jednotlivým sloupcům z celkového prostoru pro graf. Příklad skupin sloupců v grafu znázorňuje Obrázek 6.4.



Obrázek 6.4: Přehled o přijímacích řízeních za rok 2013, 2014 a 2015 [48]

Před samotným vykreslením a počítáním výšek sloupců je potřeba data ještě jednou projít. Abychom mohli vypočítat výšky sloupců vzhledem k velikosti grafu, je potřeba si zaznamenat nejvyšší sloupec. Výšku sloupce pak počítáme v procentech, abychom dosáhli responsibility výšky grafu. Na počítání šířek je potřeba znát skutečný počet sloupců, počet skupin a maximální počet sloupců v jedné skupině.

Také dochází k vypočítání a vykreslení mřížky, kterou vidíme na pozadí. Jelikož už známe výšku nejvyššího sloupce, můžeme tedy počítat s tím, že nejvyšší sloupec bude mít výšku 100% velikosti grafu. Víme tedy, že mřížku začínáme kreslit od 0 a známe maximální výšku, které mohou sloupce dosahovat. Ta se pak rovná maximální výšce mřížky. Nyní už můžeme mřížku vykreslit. V mřížce kreslíme 6 vodorovných přímek, kdy první označuje 0 a poslední 100% výšky grafu. Mezi 6 přímkami se tak nachází 5 volných ploch vymezujících rozteč mezi přímkami. Můžeme tak jednoduše celkovou výšku grafu rozdělit počtem těchto ploch.

Chceme, aby byl graf pokud možno responzivní. Proto rozteč mezi přímkami uděláme pomocí procent. Tím dosáhneme toho, že pokud by se zvětšila celková velikost grafu např. změnou výšky widgetu, dojde k okamžitému zvětšení mřížky. Prostor mezi jednotlivými mřížkami bude (100% / 5) 20% celkové výšky grafu. Výška přímky od spodní části grafu bude dána pořadím (tedy řadou 0%, 20%, 40%, ...). Nyní máme nakreslené přímky, ale ještě bychom chtěli u každé přímky vypsat hodnotu popisující danou přímku. Tu zjistíme tak, že hodnotu nejvyššího sloupce podělíme procentuálním rozdělením přímek. Pokud bychom počítali hodnoty přímek, jak ukazuje Obrázek 6.4 výše, tak hodnota nejvyššího sloupce je 134k. Proto hodnota nejvyšší přímky je 134k (134 * 100 / 100) a hodnota přímky pod ní je 107k (134 * 80 / 100).

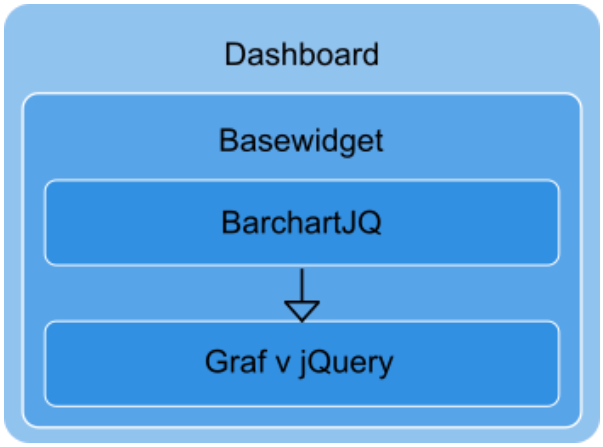
6.2.1 Sloupcový graf v Reactu, Angularu a jQuery

Na tomto grafu si také vyzkoušíme, jak graf přepisovat ze stávajícího pluginu v jQuery do Angularu. V první řadě byl plugin přepisován pouze do Angularu. Tato verze se ale natolik lišila od původního pluginu, že by porovnání používané technologie obou grafů z hlediska počtu řádků a rychlosti nebylo příliš přesné. Proto bylo potřeba přepsat graf i do jQuery, kde bylo dbáno na to, aby se dlouhé kusy kódu rozdělili do kratších, přehlednějších a komentovaných funkcí. Výkonnostní porovnání obou řešení pak nalezneme v následující Kapitole 6.4.2 Výkonnostní testy na nízké úrovni. Zde si rozebereme, jak vypadají obě řešení z pohledu přehlednosti kódu. Navíc si k porovnání ještě přidáme implementovanou verzi grafu v knihovně React. Grafy ve všech třech technologiích jsou obaleny direktivou, která umožňuje s grafy pracovat, jako s widgety dashboardu.

Porovnáváme tedy:

- sloupcový graf v jQuery
- sloupcový graf v Angularu
- sloupcový graf v Reactu

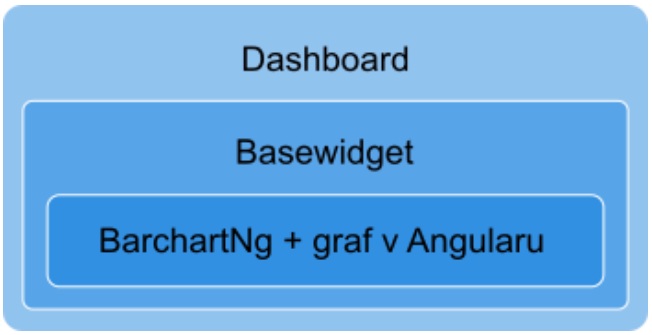
Nejdříve si rozebereme, jak funguje widget grafu v jQuery. Oproti Angularu a Reactu nemá jQuery, ani Angular šablony, ani jsx. Jsx je druh zápisu HTML kódu a Javascriptových funkcí do jednoho souboru, který používá React. HTML se píše přímo do předpřipravených funkcí a před připojením do stránky musí ještě projít kompilací (více o jsx [49]). JQuery ale žádné šablony nemá. Vytváření nových elementů v DOMu (Document Object Model) se tak musí dělat přímo ve výkonném kódu Javascriptu. Tím nám ale výkonný kód roste a stává se nepřehledným. Některé zápisy, které generují stejný HTML kód, jsou podstatně delší. To jsme si to již ukázali v příkladu generování SVG u koláčového grafu. Ještě se podíváme na závislosti grafu v jQuery. Obrázek 6.5 ukazuje závislosti dashboardu, widgetu a grafu v jQuery.



Obrázek 6.5: Závislosti sloupcového grafu v jQuery

Nyní si rozebereme vytváření sloupcového grafu v Angularu. Angular nám přináší řadu různých vylepšení, jako je rozdělení výkonného kódu do třívrstvé architektury. Asi největším přínosem Angularu je obousměrný data binding (viz. kapitola Obousměrný data binding). Ten zajistí změnu HTML podle toho, jak se mění zobrazovaná data. Odpadá nám tedy nutnost po změně dat volat metody, které provedou změnu i HTML stránky. Tím se nám výkonný kód opět o něco zkrátí. Navíc nám odpadá nutnost kontrolovat, zda jsme opravdu po každé změně dat nechali překreslit i HTML stránku. Můžeme se tak plně soustředit pouze na výkonný kód, který pracuje s daty. Samozřejmě dalším přínosem je, že dashboard i graf, jsou implementovány ve stejném frameworku. Odpadá nám tak nutnost připojovat externí knihovny jako jsou jQuery a dostáváme plnou provázanost všech dat pomocí objektu *$scope* a *observers* (funkce pro kontrolu změny proměnné).

Graf závislostí sloupcového grafu v Angularu uvádí Obrázek 6.6. Vypadá obdobně, jako graf závislostí u jQuery. Na rozdíl od grafu v jQuery můžeme spojit poslední dva body barchartNg a samotný graf v Angularu. Jde totiž o tu samou technologii v tom samém widgetu.



Obrázek 6.6: závislosti sloupcového grafu v Angularu

Posledním implementovovaným řešením je graf v Reactu. Tuto knihovnu jsme přidali proto, že je často porovnávána a Angularem [50] a s jQuery [51]. Proto je zajímavé při implementaci našeho grafu technologie porovnat s knihovnou React. Porovnání bude prováděno pouze na příkladech, které jsme vytvořili v rámci tohoto projektu. Porovnání tedy nebude obecné, ale naopak velmi konkrétní.

Jak jsme si již vysvětlili, React používá šablony jsx. Jde o druh zápisu Javascriptu a HTML do jednoho souboru. Co se týče jsx šablon, z mého pohledu jsou velmi přehledné. Jsou napůl cesty mezi HTML šablonou z Angularu, která se standardně odděluje do externího souboru a jQuery, kde se píše práce s daty a HTML stránkou do výkonného kódu. Z mého pohledu byl ale přehlednější způsob, který využívá Angular, tedy oddělovat zobrazovací vrstvu (View) do zvláštního souboru. V implementaci Kód 6.7 vidíme ukázku jsx souboru z naší práce.

```
/* Javascriptový kód */
var Bars = React.createClass({
 render: function() {
   /* část funkce sloužící pro vykreslení sloupců ve sloupcovém grafu */
   return (
     /* v jsx píšeme HTML ve zvláštních funkcích přímo do Javascriptu */
     <ul className="bar-chart bars-react" >{groups}</ul>
   );
 }, /* … další funkce třídy Bars */
});
```

Kód 6.7: Ukázka jsx souboru - kreslení sloupcového grafu

React má oproti Angularu předpřipravenou funkcionalitu pro různé události. Stačí napsat obslužnou metodu s daným názvem a React si ji sám přiřadí a bude ji používat. Například pokud podle [2] vytvoříme v dané třídě funkci s názvem *componentWillMount*, tato metoda bude zavolána ještě před tím, než se začne třída vykreslovat. V naší práci tuto metodu používáme k dopočítání různých dat, jako jsou například výšky a šířky elementů. Obdobně podobnou funkci používáme i v grafech v jQuery a Angularu. Zde ale musíme funkci manuálně vložit na místo, kde ji chceme vykonat. Díky podobným metodám, jako je *componentWillMount,* máme možnost snadno reagovat na tyto události. Používáním předpřipravených metod je kód přehlednější, protože už z názvu metody víme, kdy bude volána a máme i představu, jak by mohla být použita. Při porovnání grafu napsaném v jQuery, Angularu a Reactu, je podle mého názoru nejpřehlednější a nejlíp udržovatelný graf v Reactu.

6.2.2 Propojení Reactu a Angularu

Napojení Reactu do projektu v Angularu není triviální záležitostí. Musíme myslet na všechny vazby ve *$scope*, *attrs* a pod. Naštěstí existuje předpřipravené řešení, které toto propojení udělá za nás. Můžeme se tak spolehnout, že toto propojení funguje a plně se věnovat implementaci daného grafu. Při implementaci jsme použili direktivu [52], která propojila dashboard s aplikací v Reactu. Bohužel při propojování narážíme na problém, že v *ngReact* je nastavené zpracování atributů, jako Angular výrazů [41]. Konkrétně jsme na tento problém narazili v používané verzi 0.2.0 ve funkci *reactDirective*. V ukázce Kód 6.8 vidíme úryvek kódu z této funkce.

```
var props = {};
propNames.forEach(function(propName) {
   props[propName] = scope.$eval(attrs[propName]);
});

renderComponent(reactComponent, applyFunctions(props, scope), $timeout,
elem);
```

Kód 6.8: Úryvek kódu z funkce *reactDirective*

Tento kód zpracuje hodnotu atributu, jako výraz v Angularu. Výsledek pak předá Reactu, jako atributy. Tento výraz by dobře fungoval, pokud bychom direktivě *ngReact* předávali výraz

1 + 2 + 3 + 3

protože do Reactu by se přímo dostal výsledek tohoto výrazu 9. Pokud ale v atributu předáme např. objekt s daty

```
{firstName: "Petr", lastName: "Kukrál"}
```

vykoná se objekt, jako výraz a Reactu je předána výsledná hodnota, která je v tomto případě *undefined*. Stejný případ nastane při zpracování řetězce. Proto je nutné tento problém vyřešit. Máme tyto možnosti:

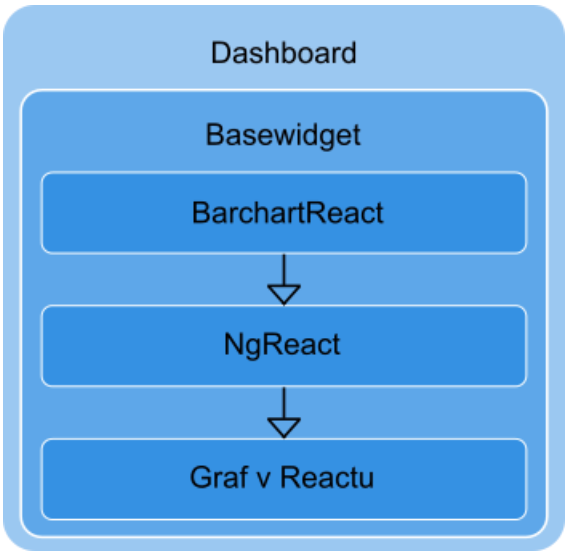
- zasáhnout do kódu *ngReact*
- změnit způsob komunikace
- vytvořit výraz, který data vrací

První variantou je zasáhnout do kódu *ngReact* a upravit ho tak, aby k tomuto zpracování nedocházelo. Toto řešení je odzkoušené a je plně funkční. Nastává ale problém, že není možné komponentu aktualizovat například pomocí nástroje Bower [7]. Při aktualizaci buď dojde k neaktualizaci některých souborů, celé komponenty, nebo k přemazání naší opravy. Další možností, jak problém vyřešit, je změnou druhu předávání dat. Místo toho, abychom data předávali přes atributy *attrs*, je předáváme pomocí *$scope*. U *$scope* totiž nedochází k vyhodnocování proměnných, jako u *attrs*. Data se tedy předávají ve správném formátu. Tím ale narušíme význam toho, že se atributy mají předávat direktivě opravdu jako atributy. Proto nakonec volíme jiné konečné řešení, které se vrací k předávání dat pomocí atributů, jak vidíme v ukázce Kód 6.9.

```
attrs.opts = function(){
   return opts;
};
```

Kód 6.9: Ukázka řešení předávaných atributů direktivě *ngReact*

Když se nad touto funkcí vykoná příkaz *$scope.$eval(attrs[propName])* dojde k prostému vrácení nastavení *opts, o* které jsme celou dobu usilovali. Data se nám tak předávají pomocí atributů a dojdou do Reactu ve správném formátu. Obrázek 6.7 ukazuje zapojení grafu napsaného v Reactu do dashboardu. Oproti zapojení jQuery a Angularu musela ještě přibýt závislost na výše uvedenou direktivu *ngReact*.



Obrázek 6.7: Závislosti sloupcového grafu v Reactu

6.3 Spojnicový graf

Posledním typem grafu je spojnicový graf. Pro kreslení je zde použitá technologie HTML canvas (více v Kapitole 2.5.1 HTML canvas). Graf nám zajišťuje vykreslení více různých křivek. Každá křivka je dána body, kterými prochází. Ze serverové části dostaneme pouze soupis všech bodů. První bod je brán, jako počáteční a poslední bod je brán, jako koncový bod křivky.

Každý bod v grafu je dán hodnotou na X-ové a Y-ové ose. Při najetí nad bod se zobrazí jeho popisek. Ten je tvořen hodnotou bodu na Y-ové ose a nastavitelným textem, který si můžeme vložit u každého grafu jiný. Příklad ukázky spojnicového grafu ukazuje Obrázek 6.8, kde červená křivka ukazuje počet přihlášek v jednotlivých letech a modrá křivka ukazuje počet přijatých uchazečů. Legenda grafu je tvořena popisky jednotlivých bodů zobrazených při přesunutí kurzoru nad bod.

Obrázek 6.8: Počet přihlášek na vysoké školy a počet přijatých uchazečů [48]

6.3.1 Vylepšení spojnicového grafu

Tento graf byl vytvářen podle [53], přepisováním pluginu z jQuery do Angularu. Původní graf ale není naprosto stejný s verzí v Angularu a to ani v programové, ani ve funkční části. Námi vytvořený graf dokáže zobrazit data stejně, nebo velmi podobně, jako [53]. Kromě toho, že byl přepsán do Angularu, má námi vytvořený graf ještě přidané vlastnosti:

- členění kódu do funkcí
- nezávislost na atributu *id*
- proměnlivý počátek souřadnic
- procentuální rozložení hodnot na osách X a Y
- zobrazení jednotek

Námi vytvořený graf má kód členěný do komentovaných funkcí. Mnoho menších pluginů v jQuery používá při implementaci pouze jednu jedinou funkci. Ta pak zajišťuje veškerou funkcionalitu pluginu. Takto je to i u obou pluginů [29] a [53], které sloužily jako předlohy pro vytváření sloupcového a spojnicového grafu. U větších pluginů a knihoven se ale bez více funkcí neobejdeme. Takto je to např. u [43], kde by jedna funkce určitě nestačila. Proto jsme i u našich grafů v Angularu použili rozdělení do více funkcí. Graf je tak připraven na to, že se bude dále rozvíjet. V komentovaných funkcích se navíc můžeme lépe orientovat.

V původním pluginu [53] existovala závislost na *id* grafu. Každý námi vytvořený graf musel mít unikátní *id*, které se muselo předávat pluginu v jeho nastavení. Pokud jsme si tak chtěli vytvořit na stránce tři spojnicové grafy, museli jsme vygenerovat tři různá *id*, pro každý graf jiné. Pokud by se nám *id* u některých grafů shodovala, dojde k nepředvídatelným stavům, kdy se pravděpodobně dva ze tří grafů se stejným *id* vůbec nezobrazí, nebo se zobrazí chybně. Tuto závislost jsme odstranili díky Angularu. V jQuery bylo *id* potřeba k tomu, abychom propojili data vložená do pluginu s elementem na stránce, kde se měl graf zobrazit. Angular nám propojení obou částí plně zajistí a my se o něj tak nemusíme starat.

Při použití reálných dat jsme zjistili, že [53] není vhodný na zobrazení některých typů dat. Jedná se o data, která se zobrazují na krátkém úseku, daleko od počátku souřadnic 0. Příkladem takového zobrazení je Obrázek 6.8 výše. Zde zobrazujeme na ose X roky od 2010 až do roku 2015. Jde tedy o malý úsek na ose X (2015 - 2010 = 5 let) daleko od počátku 0. Plugin [53] je implementovaný tak, že zobrazuje data vždy od počátku souřadnic. Tím dochází k tomu, že na ose X jsou hodnoty od roku 0 do roku 2015. Osa X je pak přeplněná hodnotami a celková data jsou zobrazována pouze v několika procentech celkové šířky grafu. Musíme tedy počítání hodnot zobrazených na osách

upravit. Náš graf v Angularu si proto sám zvolí, jestli bude data zobrazovat od počátku souřadnic 0, nebo za počátek souřadnic zvolí nejnižší zadanou hodnotu v grafu. Toto určení vidíme v ukázce Kód 6.10.

```
var diffConstant = 2; //určuje, kdy se má použít poč. 0, a kdy bude poč.
nejnižší zadaná hodnota
if((diffConstant * (max - min)) > (max - 0)) {
   min = 0;
}
```

Kód 6.10: Určování počátku souřadnic pro osu X nebo Y

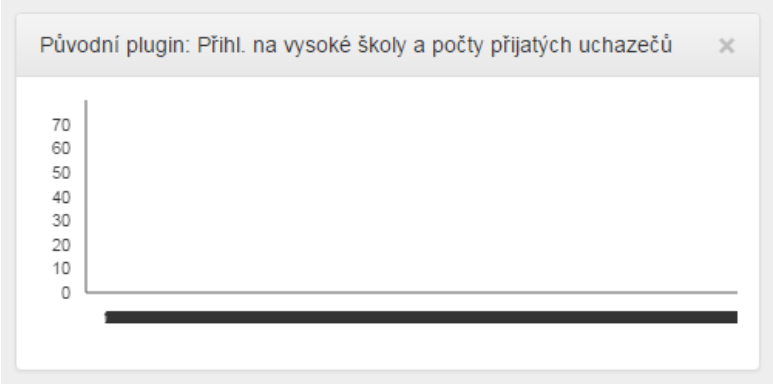
Pokud je dvojnásobný rozdíl nejvyšší a nejnižší zobrazované hodnoty v grafu větší, než rozdíl maximální hodnoty v grafu od nuly, bude počátkem souřadnic zvolena 0. V opačném případě bude ponechán na minimální zadané hodnotě v grafu. Jinými slovy tento zápis říká, že pokud by se data měla zobrazovat pouze v malé části grafu, tak se má použít místo počátku souřadnic 0, počátek v nejnižší zadané hodnotě. Kdy se má použít jaký počátek nám ovlivňuje konstanta *diffConstant*. Čím bude větší, tím více se bude používat častěji počátek souřadnic od 0. Díky této úpravě tak můžeme zobrazovat data v grafu daleko od 0.

Po použití grafového widgetu na reálných datech se objevil další problém. Ten souvisel s tím, že místo toho, aby se na osách X a Y zobrazovalo pouze pár hodnot s vhodnými rozestupy, zobrazovaly se všechny hodnoty od počátku souřadnic do maximální hodnoty. Problém si vysvětlíme na příkladu běžecké olympiády. Chtěli bychom na ose X zobrazit pořadí běžců v několika desetiletích, např. od roku 1970 do roku 2010. [53] by to vykreslil tak, že by na osu X přidal 2010 hodnot (2010 let zobrazí od nuly po jednom roku). My bychom ale chtěli, aby se tam přidaly jen některé roky, např. 1970, 1980 atd. Je tedy potřeba vytvořit nějakou funkcionalitu, která by nám tuto řadu hodnot vrátila. Řešení vidíme v ukázce Kód 6.11.

```
/* Nakreslí popisky X-ových os*/
 for(var i = 0; i <= 10; i = i + 2) {
    var percent = i*10;
    var pointX = (mmXY.maxX - mmXY.minX)/100 * percent + mmXY.minX;
    c.fillText(pointX.toFixed(0) + o.unitX, getPointX(pointX, o) + 10,
        o.graph.height - o.gridPaddingY + 20);
}
```

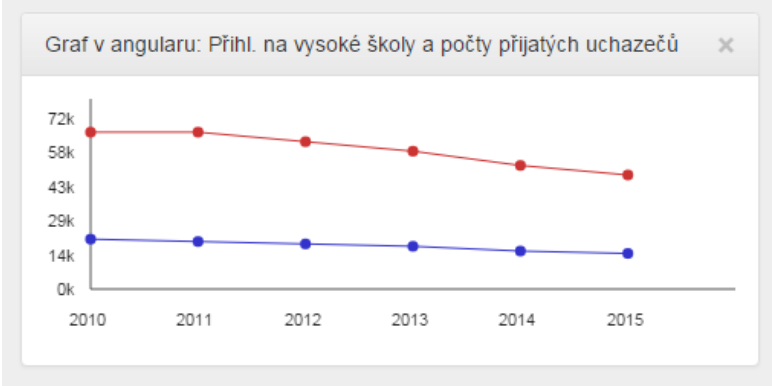
Kód 6.11: Výpočet hodnot, které se mají zobrazit na ose X

Díky této opravě zobrazujeme pouze některé hodnoty. Porovnání původního pluginu a předělané verze grafu na těch samých datech ukazuje Obrázek 6.9 a Obrázek 6.10. V obou grafech se zobrazuje počet přihlášek a počet přijatých uchazečů na vysoké školy v jednotlivých letech.



Obrázek 6.9: Původní verze pluginu v jQuery

Jak již můžeme z obrázků vidět, dalším vylepšením bylo přidání jednotek. Pokud bychom chtěli zobrazovat velká čísla (např. tisíce nebo miliony) a zobrazení desítek a jednotek nehraje roli, tak je možnost použít zkrácený zápis čísla. Např. 14 000 lze zapsat zkráceně 14k. Obrázek 6.10 ukazuje použití zkráceného zápisu na ose Y.



Obrázek 6.10: Implementace pluginu v Angularu

6.3.2 Počítání souřadnic

Další velmi významnou opravou bylo zjištění pozice kurzoru uživatele při najetí nad graf. Při přesunu kurzoru nad bod v grafu totiž potřebujeme zobrazit popisek daného bodu. Pokud pro vykreslení grafu používáme HTML, nemusíme tento problém řešit. Pouze řekneme, že se má popisek zobrazit nad elementem, na který ukazuje kurzor. Pokud ale používáme k vykreslení canvas, jako v tomto případě, je potřeba určit, kde na jakých souřadnicích, se kurzor nachází. Musíme také určit souřadnice bodu v grafu. Pokud se tyto souřadnice bodu a kurzoru protínají, je potřeba zobrazit popisek.

Musíme tedy zjistit:

- souřadnice kurzoru
- souřadnice bodu grafu
- je kurzor nad bodem grafu

Nejdříve si řekneme, jak zjistit, na jakých souřadnicích se nachází bod. Při kreslení grafu používáme metody *getPositionX* a *getPositionY*. Těmto metodám předáváme hodnotu, kterou v grafu chceme zobrazit a ony nám vrátí souřadnice dané hodnoty. Nyní je berme jako uzavřené metody, které takto fungují, ale nepotřebujeme znát jejich implementaci. Díky těmto metodám tedy zjistíme, na jaké souřadnici se bod grafu nachází.

Nyní potřebujeme zjistit, na jakých souřadnicích se nachází kurzor uživatele. K tomu nám pomáhá událost *mousemove* (pohyb kurzoru nad elementem) navázaná ke grafu. Ta nám detekuje, zda se uživatelský kurzor nachází nad grafem. Pokud ano, tak se aktivuje pokaždé, když uživatel pohne s kurzorem. Pokud tato událost nastane, musíme určit, zda se uživatel nachází nad některým bodem grafu. Událost nám vrací objekt, který obsahuje souřadnice, na kterých se kurzor nachází. Bohužel tyto souřadnice jsou v rámci celého okna prohlížeče. Musíme je tedy ještě dopočítat na souřadnice v rámci našeho grafu. To uděláme tak, že zjistíme souřadnice polohy grafu na stránce, a ty odečteme od aktuálních souřadnic kurzoru. Úryvek kódu, který tuto funkcionalitu zajišťuje vidíme v ukázce Kód 6.12:

```
var wrapRect = graph.offset();

var mouseX = parseInt(e.pageX - wrapRect.left); //X-ová souřadnice kurzoru
var mouseY = parseInt(e.pageY - wrapRect.top); //Y-ová souřadnice kurzoru
```

Kód 6.12: Zjištění souřadnic kurzoru uživatele

Rozdíl oproti [53] je ten, že [53] pro počítání souřadnic kurzoru používá funkce *event.client* a my používáme funkce *event.page*. Podle [54] spočívá rozdíl v tom, že *event.client* vrací souřadnice kurzoru od horního a levého okraje okna prohlížeče a *event.page* od horního a levého okraje stránky. Pro vysvětlení si uvedeme příklad. Když budeme mít stránku vysokou např. 6000 px a okno prohlížeče 800 px a ukážeme na konec stránky kurzorem, dostaneme rozdílné hodnoty. *Event.pageY* vrátí 6000 px a *event.clientY* vrátí 800 px. Jenže použitá funkce offset (viz. [3]) vrací souřadnice od začátku stránky, nikoliv okna prohlížeče. Bylo tedy potřeba vyměnit funkci *event.client* za *event.page*, která vrací souřadnice ze stejného počátku. Tato oprava se projeví až na větších stránkách.

Nyní známe pozici bodu v grafu a pozici kurzoru. Stačí nám zjistit, zda se kurzor nachází nad daným bodem. To uděláme tak, že projdeme pozice všech bodů a porovnáme je s pozicí kurzoru. Pozici bodů i aktuální pozici kurzoru si můžeme předpočítat, abychom tak dosáhli vyššího výkonu aplikace. Pokud se obě pozice shodují, můžeme popisek bodu zobrazit. Pokud bychom ale tento postup v této podobě použili zjistíme, že najet kurzorem na střed bodu je velmi obtížné. Proto by se nám hodilo, kdyby se popisek zobrazoval i v momentě, kdy budeme kurzorem myši velmi blízko bodu. K tomu nám slouží součet čtverců rozdílu pozice kurzoru a pozice bodu v grafu. Součet čtverců používáme proto, že se nám snáze projeví rozdílná vzdálenost X-ové či Y-ové souřadnice. Celý postup vidíme v ukázce Kód 6.13.

```
var dx = cursorX - pointX; //rozdíl X-ových souřadnic kurzoru a bodu grafu
var dy = cursorY - pointY; //rozdíl Y-ových souřadnic kurzoru a bodu grafu

var showPointConstant = 16;
if (dx * dx + dy * dy < showPointConstant )
    showPoint();
```

Kód 6.13: Počítání souřadnic pro zobrazení popisku

Jak daleko musí být kurzor od bodu, aby se popisek bodu zobrazil, nám ovlivňuje proměnná *showPointConstant*. Čím větší bude, tím dříve se popisek zobrazí. Alternativou k tomuto výpočtu by bylo, že bychom vypočítali hraniční souřadnice bodu. Pokud by se kurzor nacházel v těchto souřadnicích, popisek by se zobrazil.

6.4 Porovnání Angularu, Reactu a jQuery

V následující kapitole porovnáme React, jQuery a Angular, protože jde o velmi diskutované téma [55] a [56]. Porovnání těchto technologií se věnujeme již v Kapitole 6.2 Sloupcový graf a v některých dalších kapitolách popisující jednotlivé grafy. Zde shrneme obecné porovnání těchto technologií. Poté porovnáme jednotlivé technologie v počtu napsaných řádek a nakonec porovnáme výkon sloupcových grafů v jQuery a Angularu.

Než dojdeme k samotnému porovnání, je důležité si shrnout, co jednotlivé technologie umožňují. JQuery [3] je knihovna, která mimo jiné umožňuje měnit a vykreslovat HTML do stránky, reagovat na události a tvořit animace. Angular je framework. Nabízí nám kompletní nástroje pro vytváření klientské aplikace s využitím MVC. React je knihovna, která se zaměřuje na vykreslení a změnu HTML. Jejím cílem je vytvořit pouze View vrstvu. Vytvářet Controllery či modely ovšem neumožňuje.

	Angular	**React**	**jQuery**
Práce s HTML DOM	ano	ano	ano
Animace	ano	ano	ano
Provázání dat s vykreslovaným HTML	ano	ano	ne
HTML šablony	ano	ano	ne
AJAX	ano	ne	ano
RESTful API	ano	ne	ne
Podpora MVC architektury	ano	ne	ne
Rok vydání	2009	2011	2005
Typ	framework	knihovna	knihovna

Tabulka 6.1: Porovnání některých vlastností jednotlivých technologií

Všechny tři technologie jsou natolik rozdílné, že je nelze plně porovnat. Nemůžeme například porovnávat AJAXové požadavky jednotlivých technologií, protože React tyto požadavky vůbec neumožňuje. Můžeme pouze porovnat vybrané části jednotlivých technologií a způsob, jak se aplikace v těchto technologiích vytváří. Porovnání vybraných částí jednotlivých technologií ukazuje Tabulka 6.1.

Nyní se již zaměříme na konkrétní porovnání jednotlivých technologií přímo v naší práci s využitím vlastní implementace grafu v Reactu, Angularu i jQuery. Díky tomu si je můžeme porovnat. Jednotlivé grafy porovnáváme ze dvou hledisek:

- Porovnání rozsahu
- Porovnání výkonu

Nejdříve si všechny tři implementace porovnáme podle rozsahu, tedy podle počtu řádek kódu a počtu znaků. Poté si je na dvou různých testech porovnáme z hlediska výkonu.

6.4.1 Porovnání rozsahu

Všechny tři technologie porovnáváme podle počtu řádek, které zabrala implementace sloupcového grafu. Velikost aplikace je ve webových technologiích důležitým ukazatelem. Klientská aplikace se musí před spuštěním celá stáhnout a čím je aplikace větší, tím déle se bude stahovat. Nyní si porovnáme počty řádek. Měření proběhlo pomocí nástroje [57]. Měří se pouze výkonný kód, tedy vše, co je v souborech jsx a js v kořenové složce daného widgetu. Do měření nebyly započítány HTML šablony.

	jQuery	**Angular**	**React**
Počet řádek	311	300	290
Počet znaků	10 401	10 173	8 957

Tabulka 6.2: Srovnání jQuery, Angular a React sloupcového grafu

Tabulka 6.2 ukazuje, že největší rozdíl nastává u Reactu. Ten má přibližně o 1300 znaků méně znaků než Angular a jQuery. I když nejde o výrazný rozdíl, z měření vyplývá, že widget napsaný v Reactu bude kratší, než widget napsaný v jQuery a Angularu. Je nutné zmínit, že plugin v Reactu byl odlišně komentován, což mohlo k rozdílné délce přispět. React má spoustu předpřipravených názvů funkcí, které jsou jasně popsané v dokumentaci a není potřeba je tedy znovu komentovat. Délka kódu byla u všech tří widgetů srovnatelná.

6.4.2 Výkonnostní testy na nízké úrovni

Další možností, jak můžeme widgety porovnat, je pomocí výkonnostních testů. Konkrétně se zaměříme na porovnání výkonu widgetu implementovaného v jQuery a widgetu implementovaného v Angularu. Test provedeme na widgetu pro sloupcový graf. Výkon měříme na nízké úrovni, testování je tedy zakomponováno přímo do jednotlivých grafů. Při testech měníme data, která mají jednotlivé grafy zobrazovat. Měřená veličina je pak čas, který trvá změna dat a jejich vykreslení v daném grafu. Budeme testovat tři různé scénáře, ve kterých bude docházet k různým změnám dat. Tyto scénáře jsou:

- vykreslení úplně jiné řady sloupců
- odstranění některých existujících sloupců
- změny hodnot tří vybraných sloupců

Předpokladem je, že vykreslování úplně jiné řady sloupců zabere více času, než pouhá změna několika sloupců. Při změně pouze několika sloupců totiž nedochází k tak velké změně výsledného HTML. Musíme si uvědomit, že změna HTML je časově velmi

náročná. S touto změnou totiž souvisí i změna HTML DOM, navázání nových reakcí na události a vykreslení daného HTML prohlížečem. Proto je důležité, aby bylo výsledné HTML měněno co možná nejméně.

Nyní si popíšeme výkonnostní testy se sloupcovým grafem v jQuery. Pokud upravíme data v Angularu, dojde k automatické změně vykreslovaného HTML. JQuery ale nic takového neobsahuje. Když po vykreslení grafu změníme data, graf v jQuery se nijak nezmění. Proto bylo pro potřeby tohoto testu nutné dopsat speciální funkcionalitu, která tato data bude měnit. Tento test tedy probíhá ve dvou fázích:

- odstranění všech existujících sloupců
- vykreslení požadované řady

Odstraněním všech sloupců uvolníme graf pro nová data, která poté vykreslíme. Jelikož dochází k odstranění všech sloupců, nedochází tak v jQuery testu k rozdílu mezi jednotlivými testy. Přibližně stejně by měl dopadnout test na vykreslení úplně jiné řady sloupců a test na změnu pouze tří sloupců. Chtěli jsme poukázat na to, že u jQuery neprobíhá žádná kontrola již vykreslených dat s novými daty. Pokud totiž v již vykresleném grafu pomocí jQuery chceme změnit jedinou hodnotu, musíme smazat celý graf a vykreslit ho znovu. Pokud bychom si kontrolu již existujících dat napsali ručně, přiblížili bychom se Angularu. V této práci se ale nesnažíme vytvořit u jQuery podobnou funkcionalitu, jako nabízí Angular. Proto se snažíme změnit graf v jQuery co možná nejmenším úsilím. I tak jsme museli dopsat funkce, které v testu Angularu nejsou potřeba.

jQuery	**1. test**	**2. test**	**3. test**	**4. test**	**5. test**	**průměr (ms)**
Vykreslení úplně jiné řady	4	3	3	5	5	**4**
Odstranění dvou sloupců, zbytek ponechán beze změny.	7	2	2	4	3	**3,6**
Změny tří hodnot za jiné hodnoty	4	2	2	3	3	**2,8**
Celkově	11	4	4	7	6	**6,4**

Tabulka 6.3: Výkonnostní testy sloupcového grafu v jQuery

Tabulka 6.3 nám potvrdila, že jednotlivé testovací scénáře změny dat probíhají v podobném čase. U Angularu by tomu tak být nemělo. V Angularu by se měly časy jednotlivých scénářů více odlišit. Minimálně by se měl lišit čas překreslení celé řady sloupců od úprav jen některých z nich.

Při normálním běhu Angularu se díky obousměrnému data bindingu kontroluje, jaké hodnoty se v datech změnily. Pokud je hodnota změněna, změní se tak výsledné HTML. Angular nám ale přesně nespecifikuje, kdy ke kontrole dojde. Jelikož tvoříme test, ve kterém počítáme milisekundy, musí ke kontrole a překreslení dojít okamžitě po změně dat. Abychom okamžitě po změně dat vyvolali změnu HTML, musíme podle [58] spustit funkci *$digest*. Ta vyvolá okamžitou kontrolu dat a překreslení výsledného HTML. Bohužel, kontrola všech dat v daném widgetu, včetně všech závislostí je rozsáhlá a může se projevit na výsledných časech.

Angular	**1. test**	**2. test**	**3. test**	**4. test**	**5. test**	**průměr (ms)**
Vykreslení úplně jiné řady	11	22	17	19	26	**19**
Odstranění dvou sloupců, zbytek ponechán beze změny.	9	12	12	9	9	**10,2**
Změny tří hodnot za jiné hodnoty	9	9	8	8	9	**8,6**
Celkově	18	21	20	17	18	**18,8**

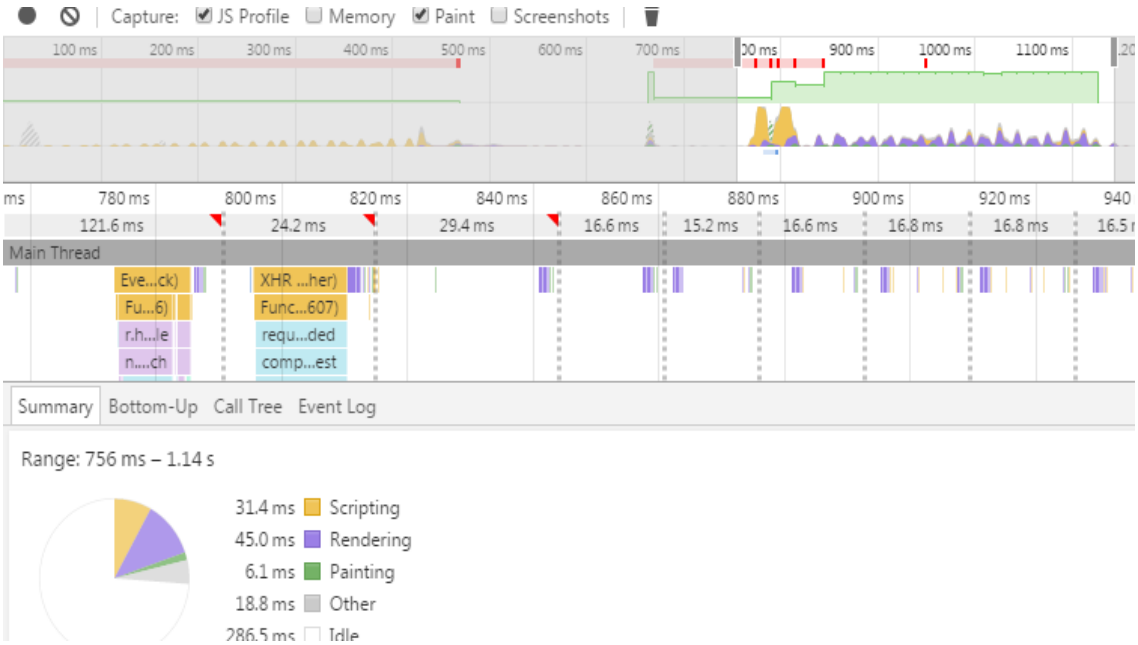
Tabulka 6.4: Výkonnostní testy sloupcového grafu v Angularu

Tabulka 6.4 ukazuje, že testy dopadly dle očekávání. Funkce pro změnu jedné řady za úplně jinou řadu je časově přibližně dvakrát náročnější, než změna pouze několika hodnot v grafu. Tím se liší Angular od jQuery, kde tyto hodnoty byly podobné. Čas pro odstranění dvou sloupců v grafu je podobný času změny velikosti tří sloupců. V porovnání s jQuery je Angular přibližně trojnásobně pomalejší. Může to být z důvodu, že při testech jsme si vyžádali okamžitou a úplnou kontrolu všech dat, nastavení a kontrolu všech událostí v grafu. Závěrem tedy můžeme říci, že v tomto testu porovnání rychlostí je rychlejší jQuery. Nicméně nesmíme zapomenout, že u jQuery jsme museli dopisovat další funkcionalitu na změnu dat, která u Angularu nebyla potřeba. To přináší další náklady na programování a je velká pravděpodobnost, že u dvou jQuery pluginů od různých autorů bude tato změna probíhat odlišně.

Testy výkonnosti jsou plně automatické. Proto si je můžeme spustit v demu aplikace na stránce *Výkonnostní testy jQuery a Angularu*. Výsledky testů jsou ukládané do konzole prohlížeče. Například pokud bychom si výsledky chtěli zobrazit v prohlížeči Chrome, stačí stisknout klávesovou zkratku *Control + Shift + C* a zobrazit kartu *Console*.

6.4.3 Výkonnostní testy na úrovni prohlížeče

Na úrovni prohlížeče testujeme již jednotlivé rychlosti měřené přímo prohlížečem. Pro testování jsme vybrali prohlížeč Google Chrome, protože podle statistiky v Kapitole 2.5 Vizualizace dat ho používá přibližně 70% uživatelů. Chrome dokáže změřit, jak dlouho trvalo překreslení celého grafu. Pro toto měření jsme si v našem demu připravili speciální stránku *Výkonnostní testy v Reactu, Angularu a jQuery*. Ukázku měření zobrazuje Obrázek 6.11. V demu aplikace máme na této stránce připravené tlačítko, po jehož zmáčknutí se graf překreslí novými hodnotami. My budeme měřit dobu od zmáčknutí tohoto tlačítka (událost click) do překreslení grafu.



Obrázek 6.11: Ukázka měření rychlosti překreslení grafu

Jako měřící nástroj používáme nástroj Timeline [59]. Ten nám dokáže měřit několik druhů časů. Nás bude zajímat součet časů:

- doba trvání vykonávání scriptu (scripting)
- změna DOM, připojení a vykreslení stylů (rendering)
- vykreslení jednotlivých pixelů na obrazovku (painting)

Všechny tři časy porovnáme v technologiích:

- jQuery
- Angular
- React

	1. test	2. test	3. test	4. test	5. test	Průměr
scripting	21,6	21,2	31,2	23,8	23,8	**24,32**
rendering	56,5	44,7	58	59	51,7	**53,98**
painting	8,8	6,2	9,3	9,6	7,8	**8,34**
celkový čas	86,9	72,1	98,5	92,4	83,3	**86,64**

Tabulka 6.5: Test výkonnosti jQuery (ms)

Jako první testujeme jQuery. Výsledky testů zobrazuje Tabulka 6.5. Test by měl v porovnání s Angularem a Reactem dopadnout tak, že jQuery bude mít nejkratší dobu trvání scriptu a nejdelší renderování (vykreslení). Krátký čas běhu scriptu vychází z předpokladu, že skript v jQuery se nemusí zabývat tím, která data již byla vykreslena a které zatím ne. Jelikož se ale vykreslí všechna data úplně znovu, renderování by mělo být nejdelší, protože React a Angular překreslují pouze HTML, které se mění.

	1. test	2. test	3. test	4. test	5. test	Průměr
scripting	24,7	28,2	34,7	32,4	31,4	**30,28**
rendering	35,2	42,6	45,6	36,4	45	**40,96**
painting	5,8	6,5	7	5,7	6,1	**6,22**
celkový čas	65,7	77,3	87,3	74,5	82,5	**77,46**

Tabulka 6.6: Test výkonnosti Angular (ms)

Tabulka 6.6 ukazuje, že Angular je opravdu pomalejší ve vykonání scriptu než jQuery. Vykonání je průměrně o 6 ms delší. Naopak vykreslení grafu (rendering) je o přibližně 13 ms kratší. Potvrdil se tedy předpoklad, že Angular bude vykreslovat data rychleji, než jQuery. Celkově je přibližně o 9 ms rychlejší, než graf v jQuery.

	1. test	2. test	3. test	4. test	5. test	Průměr
scripting	23,1	27,7	29,7	30,4	31,5	**28,48**
rendering	48,8	56	44,7	61,3	57,9	**53,74**
painting	7,8	9,6	5,9	9,5	8,7	**8,3**
celkový čas	79,7	93,3	80,3	101,2	98,1	**90,52**

Tabulka 6.7: Test výkonnosti React (ms)

Tabulka 6.7 dokazuje, že React v tomto testu dopadl nejhůře. Čas vykreslení má shodný s grafem v jQuery, ale převyšuje ho o několik milisekund v době běhu scriptu. Delší doba běhu scriptu může být způsobena zapojením, kdy mezi grafový widget a samotný graf v Reactu musí být ještě zapojena direktiva *ngReact*, která obě části propojuje.

7 Testování

Celé jádro aplikace je pokryto automatickými testy. Jako testovací nástroj je použita Karma [10] s podporou Jasmine [41]. Karma je testovací nástroj v příkazové řádce, používaný v Angularu pro jednotkové testy. Jeho hlavní výhodou je, že spojuje vykonávání aplikace v prohlížeči a spouštění jednotkových testů. V Karmě můžeme spouštět testy v různých prohlížečích. To je výhodné, protože se změnou prohlížeče může dojít k odlišnému chování aplikace. Proto díky testům v Karmě víme nejen to, že aplikace funguje, ale také že funguje i na jiném prohlížeči, než který používáme při vývoji aplikace.

Jasmine nám vytváří strukturu testovacího kódu. Obsahuje řadu Javasciptových funkcí, které nám pomohou lépe strukturalizovat kód do oddělených funkcí. Příkladem je funkce *describe*, která odděluje testovací kód různých částí aplikace. Dalším příkladem je funkce *beforeach*, která spouští určitý kód před každým spuštěním testu. Abychom Jasmine lépe pochopili, ukážeme si vše na příkladu testu přímo z dashboardu v ukázce Kód 7.1.

```
// Nový test, kde testujeme Dashboard, jako celek.
// Abychom ho oddělili od ostatních testů, uzavřeme ho do funkce describe.
describe('Dashboard test', function() {
   // Testovací aplikaci chceme načíst před každým spuštěním testu
   beforeEach(module("dashboardApp"));
   // Sem napíšeme kód, který nám bude testovat Dashboard
}
```

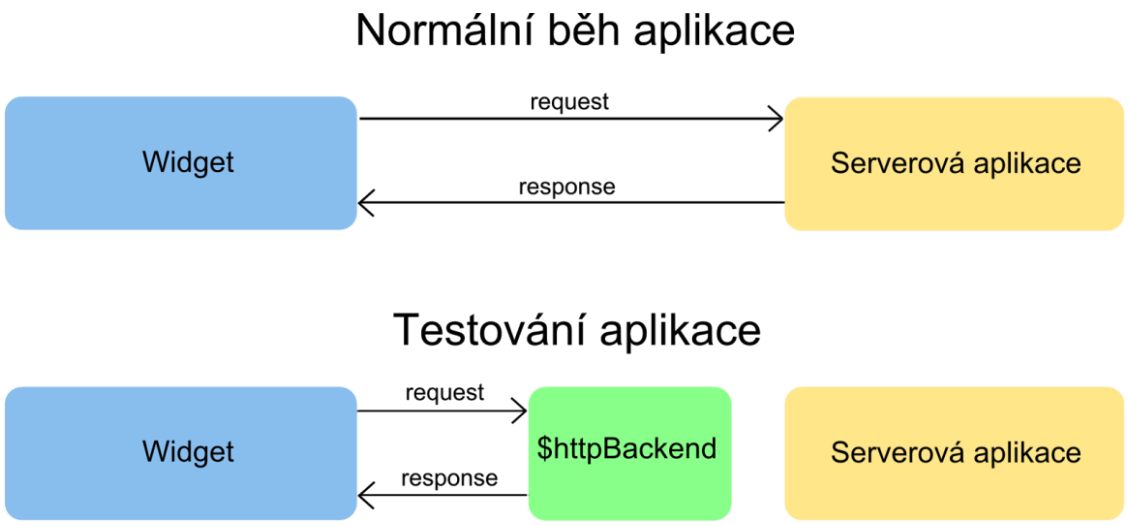
Kód 7.1: Ukázka Jasmine na testování Dashboardu

Při testování dashboardu dochází i k načítání připojených widgetů. Když tedy testujeme dashboard a naplníme ho widgety, testujeme najednou:

- základní funkce dashboardu
- správné napojení widgetů
- vykreslení jednotlivých widgetů
- komunikaci se serverem

Jedním testem dashboardu jsme tedy ověřili funkcionalitu celé aplikace. To je výhodné, protože až budeme testovat jednotlivé widgety a dashboard zvlášť, testy mohou procházet, ale aplikace při tom nemusí fungovat správně. Díky tomuto testu jsme schopní otestovat celou aplikaci, jako celek a to ještě v závislosti na jednotlivých prohlížečích. Kromě otestování celé aplikace jsou jednotlivé widgety otestovány i samostatně. V aplikaci se nyní nachází 16 oddělených souborů, které testují jednotlivé widgety, Controllery a dashboard. V každém souboru je řada testů, ověřujících danou funkcionalitu.

Některé widgety pro své fungování potřebují komunikovat se severovou aplikací. Například grafové widgety si ze serverové aplikace stahují data, která mají zobrazovat. To nám tvoří závislost testů klientské aplikace na serverové, protože bez dat ze serverové aplikace nemůže widget správně fungovat. Pokud se tedy objeví chyba na serveru, dochází k pádu testů v klientské aplikaci. Abychom tuto závislost zrušili, Angular nám nabízí službu *$httpBackend* [41], která slouží jako imaginární serverová aplikace. Když při testování pošle widget GET požadavek, nedojde k vykonání tohoto požadavku. Angular pouze předá volání službě *$httpBackend*. Ta nám rovnou vrátí odpověď na požadavek (např. vrátí data widgetu) a nedochází tak k žádné komunikaci se serverovou aplikací. Díky tomu stabilita serverové aplikace neovlivňuje testy klientské aplikace. Příklad funkce služby *$httpBackend* ukazuje Obrázek 7.1.



Obrázek 7.1: ukázka služby *$httpBackend*

Další možností, jak testovat aplikaci v Angularu, jsou end2end testy. Tyto testy podle [21] slouží k tomu, aby testovaly aplikaci z pohledu uživatele. Píšeme tak testy, kde se snažíme simulovat chování uživatele v aplikaci. Například můžeme napsat test, kde vyplníme přihlašovací formulář do aplikace, klikneme na tlačítko odeslat a kontrolujeme, zda se uživatel do aplikace přihlásil. Podle [21] end2end testy odchytí další chyby způsobené integrací jednotlivých částí aplikace.

Od jednotkových testů se tyto testy liší tím, že nemáme přístup přímo do implementace aplikace. V end2end testech tak nemůžeme měnit vlastnosti aplikace přímou změnou některé její hodnoty. Také nemůžeme testovat jednotlivé funkce aplikace. Můžeme ale manipulovat s prohlížečem, klikat na elementy kurzorem myši nebo vyplňovat formulářová pole přímo v prohlížeči. Díky možnosti manipulace s prohlížečem, můžeme nahlédnout do jeho historie, či zkontrolovat, že nás aplikace přesměrovala na správnou stránku. Cílem end2end testů je co nejvíce nasimulovat chování uživatele v aplikaci a zkontrolovat, že se aplikace chová správně.

Pro tyto testy v Angularu slouží nástroje Protractor [11] a Jasmine. Protraktor je program, který poskytuje funkce pro práci s prohlížečem a funkce pro hledání a manipulaci s elementy na stránce. Příkladem takovýchto funkcí je funkce *browser.get("nová url")*, která přesměruje prohlížeč na požadovanou URL adresu. Dalším příkladem je funkce *element.click()*, která kurzorem myši klikne na daný element, například na tlačítko.

Musíme ale dávat pozor na odlišnosti od Karmy. Karma může pracovat pouze s vygenerovaným HTML. To znamená, že vše co je v HTML, můžeme najít a ovlivnit. A to i části HTML, které jsou pro uživatele skryté. Na rozdíl od Karmy Protraktor pracuje pouze s tím, co vidí uživatel. Tedy například pokud chceme otestovat titulek, který se uživateli zobrazí až při najetí kurzorem nad element, Karma je schopná ho na stránce najít. U Protraktoru tomu tak není. Protraktor nám vypíše zprávu, že žádný takový titulek neexistuje. Tento rozdíl je dobré si uvědomit, než začneme vytvářet end2end testy. Jelikož jsou tyto testy odlišné od jednotkových testů, jsou i tyto testy přidány do naší aplikace. Pokrývají zde některé další části, které by jednotkové testy pokryly jen velmi obtížně, nebo vůbec. V ukázce Kód 7.2 vidíme příklad testu, který kontroluje přesměrování stránky.

```
it('Pokud jsme na index.html, stránka nás přesměruje na /view1',
function() {
  browser.get('index.html');
  expect(browser.getLocationAbsUrl()).toMatch("/view1");
});
```

Kód 7.2: Kontrola přesměrování pomocí end2end testů

Nyní si porovnáme testovací nástroje Angularu a jQuery. I jQuery nabízí možnosti testování pluginů. Podle [3] se na testování jQuery používají QUnit testy. Ty nabízí řadu možností, jak testovat pluginy pomocí jednotkových testů. Stejně tak, jako u testování Angularu, i zde je možné používat různé podpůrné funkce. Příkladem je výše zmiňovaná funkce *beforeEach*, která se spustí před každým testem. QUnit testy také nabízí rozdělení testů do modulů. To přináší větší přehlednost a možnost spustit jen vybraný modul.

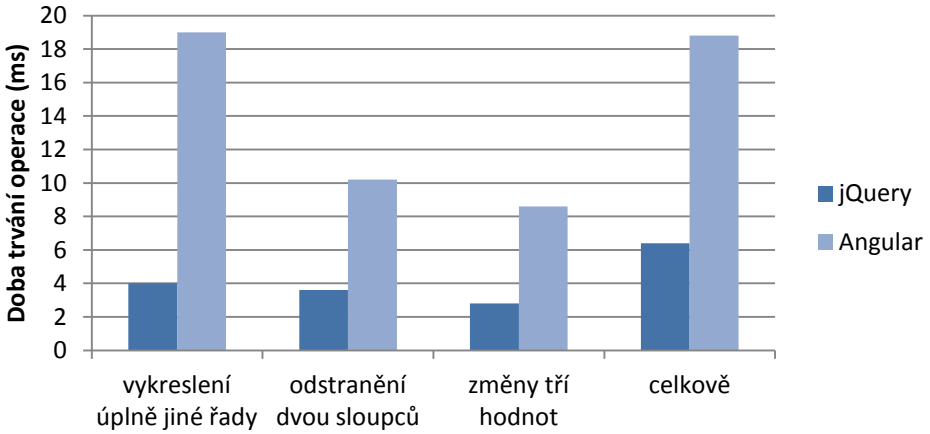
	Angular	jQuery
Jednotkové testy	ano	ano
Rozdělení jednotkových testů do modulů	ano	ano
End2end testy	ano	ne
Možnost imaginární serverové aplikace	ano	ne
Automatické načítání testované aplikace	ano	ne

Tabulka 7.1: Porovnání vybraných vlastností testovacích nástrojů pro jQuery a Angular

Jak ukazuje Tabulka 7.1, testovací nástroje Angularu nabízí stejné, nebo podobné možnosti jako QUnit. Oproti QUnit ale přináší řadu dalších vylepšení, jak aplikaci testovat. Příkladem může být výše zmiňovaná funkce *$httpBackend* sloužící, jako imaginární serverová aplikace. Další velkou výhodu přináší sama Karma. Stačí jí pouze nastavit, v jakých složkách se aplikace nachází a ona si sama aplikaci načte a spustí ještě před testem. Podle [3] ale QUunit funguje odlišně. Zde si musíme testy přidat přímo do testované části stránky, nebo vytvořit novou stránku, kde si testy spustíme. To přináší další čas, který musí programátor přípravou QUnit testů strávit. Nyní ale probíráme pouze jednotkové testy. Angular oproti jQuery již v základním použití přináší možnost psát end2end testy. Tyto testy nám v základním použití jQuery nenabízí. Může tedy jít o další důvod, proč plugin napsaný v jQuery přepsat do Angularu.

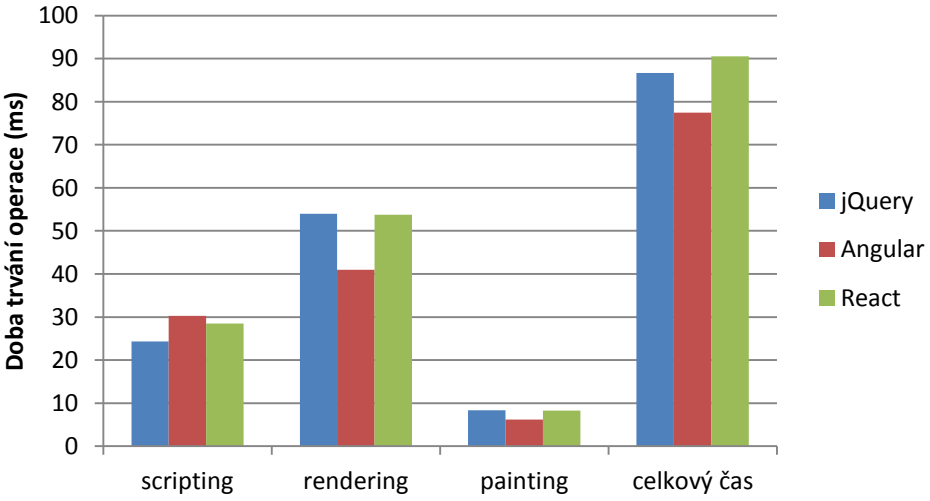
8 Dosažené výsledky

V této práci jsme se zabývali implementací Dashboardu a vizualizací dat ve frameworku Angular. Shrnutím samotné implementace se budeme zabývat v Závěru. Zde si uvedeme dosažené výsledky při porovnání jednotlivých technologií a shrneme si automatické testy aplikace. Nad rámec Diplomové práce byly implementovány grafy v technologiích JQuery a React. Díky tomu si jednotlivé technologie můžeme porovnat přímo na případu vizualizace dat. V této kapitole si shrneme výsledky z tohoto porovnání.



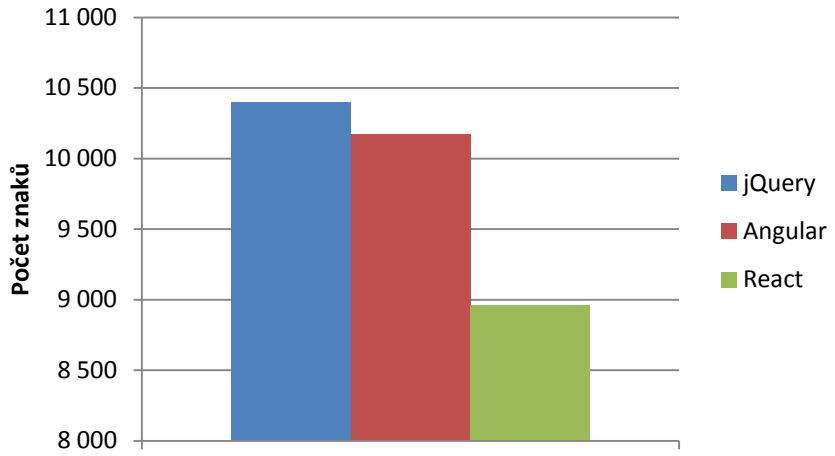
Graf 8.1: Porovnání jQuery a Angularu na nízkoúrovňových operacích

V Kapitole 6.4.2 Výkonnostní testy na nízké úrovni testujeme jQuery a Angular na nízkoúrovňových operacích. Graf 8.1 zobrazuje, že v testu výkonnosti je Angular pomalejší než jQuery, což může být způsobeno použitím funkce *$digest* při testování, ale na druhou stranu je kód aplikace lépe navržený a čitelnější.



Graf 8.2: Porovnání jQuery a Angularu a Reactu na úrovni prohlížeče

V druhém testu popisovaném v Kapitole 6.4.3 Výkonnostní testy na úrovni prohlížeče se už zaměřujeme na měření času přímo v prohlížeči. Do porovnání, které zobrazuje Graf 8.2, nám přibyla i technologie React. Test dopadl dle očekávání. JQuery je nejrychlejší při vykonávání scriptu. Angular a React kromě kontroly závislostí navíc porovnávají data a vykreslují pouze ta, která se změnila. JQuery je ale náročné na samotné vykreslení (rendering). To z důvodu, že se překreslují i ty sloupce, u kterých se hodnota nezměnila. Ve výsledných časech je pak nejrychlejší Angular. Naopak nejpomalejší je React. To může být způsobeno tím, že graf v Reactu je do aplikace napojen přes direktivu ngReact, která umožňuje obě technologie propojit. NgReact tak může mít vliv na čas vykonávání scriptu grafu.



Graf 8.3: Porovnání počtu znaků implementace v jQuery a Angularu a Reactu

Jelikož se jedná o klientskou aplikaci, která se před spuštěním musí stáhnout do prohlížeče, shrneme si závěr z Kapitoly 6.4.1 Porovnání rozsahu, kde se zaměříme na porovnání všech tři technologií v počtu znaků. Toto porovnání zobrazuje Graf 8.3. Největší rozdíl nastává u Reactu, který zabírá oproti Angularu, či jQuery mnohem méně znaků. Jeho stažení do prohlížeče bude trvat nejkratší dobu. Nejhůře naopak dopadl plugin v jQuery.

```
INFO [karma]: Karma v0.12.37 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 49.0.2623 (Windows 7 0.0.0)]: Connected on socket ny7EckMHuCk332IFiRNC
with id 83054133
................
Chrome 49.0.2623 (Windows 7 0.0.0): Executed 16 of 16 SUCCESS (0.404 secs / 0.392
secs)
```

Obrázek 8.1: Ukázka automatických testů

Webová aplikace je pokryta testy s využitím nástrojů Karmy a Protractor. Karma testuje aplikaci pomocí jednotkových testů, jak zobrazuje Obrázek 8.1 a protractor testuje aplikaci end2end testy (více v Kapitole 7 Testování). Pro testování jsme použili službu

$httpBackend, která simuluje serverovou část a v kombinaci s Karmou nám vytváří testy nezávislé na serverové aplikaci. End2end testy naproti tomu simulují chování uživatele a poskytují odlišný postup, jak aplikaci testovat.

Závěrem si shrneme, v jakých případech je lepší používání jQuery, Reactu, nebo Angularu. Dle provedených experimentů je lepší používání jQuery tam, kde vytváříme serverově orientovanou aplikaci. To potvrzuje i zdroj [20]. Tedy aplikaci, kde větší část včetně renderování HTML vytváříme na serveru. Javascript v tomto případě používáme jen pro drobné úpravy stránky, jako jsou validace a animace. Pokud ale vytváříme jednostránkovou aplikaci (singlepage aplication), tedy aplikaci ve které většinu času zůstaneme na jedné stránce, je lepší podle [20] použít novější technologii Angular. Ta nám umožní vytvořit přehlednou a dobře upravitelnou klientskou aplikaci, která se serverem komunikuje zpravidla pouze za účelem výměny dat.

Jak jsme si již napsali výše, React a Angular nejde přímo porovnat. Angular pravděpodobně použijeme tam, kde vyžadujeme odzkoušený framework se zavedenými postupy implementace aplikace a testování. React použijeme v momentě, kdy chceme využívat podle [56] snadno pochopitelnou knihovnu a nevadí nám kombinace s jinými knihovnami. Co se týče výkonu, obě technologie jsou podle našeho měření podobné. Aplikace v Reactu je ale znatelně menší, React tedy můžeme použít tam, kde potřebujeme opravdu nízký přenos dat při prvním načtení aplikace. React taky použijeme v aplikacích, kde potřebujeme vyšší výkon, který Reactu dodává podle [50] zavedení virtuálního DOM (Document Object Model).

9 Závěr

Cílem práce bylo vytvořit sadu modulů pro vizualizaci dat v Angularu. Podle mého názoru jsem zadání splnil a nad rámec práce jsem implementoval ukázku vizualizace v Reactu. V práci ukazuji, jaké technologie je možné použít při vizualizaci na webu. Také jsem nastínil postup, jak je možné přepsat do Angularu některý z existujících pluginů. V tuto chvíli je vyvíjen Angular 2, avšak práci jsem vytvořil v Angularu 1, neboť Angular 2 je stále ještě v Beta verzi. Zdrojové kódy aplikace jsem zveřejnil na portálu Github a uvolnil jsem je i pod nejpoužívanějšími licencemi GPL a MIT.

V této práci jsem se naučil implementovat pokročilé webové aplikace ve frameworku Angular. V práci využívám nejlepší postupy (best practices), jako jsou například balíčkovací systémy, automatické testy přímo pro Angular či systém pro správu verzí. V direktivách odděluji šablonu (View) do samostatných souborů a pomocí oddělení sdílených dat zapouzdřuji direktivy do nezávislých celků. Tato práce obsahuje návod na přepis jQuery pluginů do Angularu, který najdeme v Kapitole 2.4.7 Konverze jQuery pluginu do Angularu.

Nad rámec zadání diplomové práce jsem implementoval graf v technologii React a propojil ho s Angularem. Věnoval jsem se i porovnání obou technologií, protože jde o velmi diskutované téma, kde porovnávám obě technologie z pohledu vizualizace dat. Závěry z tohoto porovnání najdeme v Kapitole 8 Dosažené výsledky. Práce nám může také sloužit jako návod, jak napojit knihovnu React do existující aplikace v Angularu.

Serverová a klientská aplikace byly vyvíjeny samostatně a zcela nezávisle. Komunikace probíhá pouze na úrovni HTTP požadavků, ve kterých si aplikace vyměňují data ve formátu JSON. Díky tomu je možné serverovou aplikaci nahradit zcela jinou aplikací a to i v jiné technologii, jako je například .NET, nebo nodejs. Funkce klientské aplikace přitom zůstanou zachovány.

Přehled zkratek

Responzivní - pojem pocházející z angličtiny. Do češtiny ho překládáme jako přizpůsobivý. Termín responzivní web nám označuje webovou stránku, která se svojí velikostí přizpůsobuje velikosti okna prohlížeče.

Dashboard - pochází z angličtiny. Do češtiny ho volně překládáme jako nástěnka nebo přístrojová deska. Dashboard nám ve firmě může sloužit k rychlému přehledu o situaci podniku. Může nám například ukazovat klíčové statistiky a ukazatele.

Widget - je část dashboardu, ve kterém se sleduje určitá vlastnost systému. Dashboard je zpravidla tvořen více widgety. Ve widgetu můžeme sledovat například klíčové statistiky nebo ukazatele.

Jsx - je druh zápisu HTML kódu a Javascriptových funkcí do jednoho souboru, který používá například React. HTML se píše přímo do předpřipravených funkcí a před připojením do stránky musí ještě projít kompilací (více o jsx [49])

$scope - je objekt v Angularu, který zajišťuje předávání hodnot mezi šablonou a řídícím prvkem (controllerem nebo direktivou). Kromě funkce přepravky nabízí i mnoho funkcí, které nám usnadňují sledování proměnných či posílání událostí. Více o scope v Kapitole 2.4.4 Scope.

Observer - pochází z anglického jazyka. Do češtiny ho překládáme jako pozorovatel. V Angularu je *observer* funkce, která nám zajistí kontrolu změny například u atributu.

Direktiva - nám v Angularu umožňuje vytvářet nové značky a atributy. Pomocí direktivy můžeme oddělit kód z aplikace do znovupoužitelného celku. Více o direktivách najdeme v Kapitole 2.4.5 Direktivy.

DOM - je zkratka Document Object Model. Tvoří strukturu dokumentu, díky které můžeme přistupovat k jednotlivým elementům na stránce [33].

jqLite - je malá část jQuery knihovny, která tvoří jádro frameworku Angular. Angular ho využívá ke změně elementů na stránce [1].

PDO - je zkratka PHP Data Objects. PDO slouží jako rozhraní pro přístup k databázím v PHP. Pokud SŘBD implementuje PDO, v PHP můžeme s databází pracovat pomocí PDO metod.

SŘBD - je zkratka Systém pro řízení báze dat. SŘBD zajišťuje práci s databází a tvoří rozhraní mezi aplikačními programy a uloženými daty. SŘBD neobsahuje bázi dat, pouze s ní může pracovat.

Použitá literatura

[1] P. Kozlowski a P. B. Darwin, Mastering Web Aplication Development with Angular, Packt Publishing, 2013.

[2] P. O'Shannessy, „React", [Online]. Available: https://facebook.github.io/react/docs/component-specs.html#lifecycle-methods. [Přístup získán 7 únor 2016].

[3] J. Resig, JQuery Kuchařka programátora, Brno: Computer Press, 2010.

[4] S. Chacon a B. Straub, Pro Git, Apress, 2009.

[5] P. Kukrál, „Dashboard Github", [Online]. Available: https://github.com/Jeriiii/op-dashboard. [Přístup získán 9 červenec 2015].

[6] „Npm", [Online]. Available: https://www.npmjs.com/. [Přístup získán 8 březen 2016].

[7] „Bower", [Online]. Available: http://bower.io/. [Přístup získán 3 březen 2016].

[8] „Angular Seed Github" [Online]. Available: https://github.com/angular/angular-seed. [Přístup získán 26 červenec 2015].

[9] „Nodejs", [Online]. Available: https://nodejs.org/en/. [Přístup získán 1 březen 2016].

[10] F. Ziegelmayer, „Karma", [Online]. Available: https://karma-runner.github.io/0.13/index.html. [Přístup získán 22 únor 2016].

[11] „Protractor", [Online]. Available: https://angular.github.io/protractor/#/. [Přístup získán 19 únor 2016].

[12] M. Otto, „Bootstrap", [Online]. Available: http://getbootstrap.com/. [Přístup získán 28 prosinec 2015].

[13] W. J. Gilmore, Velká kniha PHP 5 & MySQL, Brno: Zoner Press, 2005.

[14] M. Hujer, „Jaké novinky přinese PHP 7 Zdroják", 8 červen 2015. [Online]. Available: https://www.zdrojak.cz/clanky/jake-novinky-prinese-php-7/. [Přístup

získán 28 červen 2015].

[15] V. Dajbych, „K čemu je dobrý Typescript Zdroják,“ 16 září 2016. [Online]. Available: https://www.zdrojak.cz/clanky/k-cemu-je-dobry-typescript/. [Přístup získán 5 březen 2016].

[16] M. Krátký a J. Dvorský, „Úvod do programování“, 2015. [Online]. Available: http://www.cs.vsb.cz/kratky/courses/2004-05/udp/presentation/udp-10_6.pdf. [Přístup získán 7 únor 2016].

[17] „V8 Google Developers“, [Online]. Available: https://developers.google.com/v8/. [Přístup získán 16 březen 2016].

[18] M. Malý, „Single page apps Zdroják“, 1 června 2011. [Online]. Available: https://www.zdrojak.cz/clanky/single-page-apps-a-reseni-problemu-s-historii/. [Přístup získán 5 leden 2016].

[19] D. Ondell, Javascript, průvodce programování ajaxových aplikací, Brno: Computer Press, 2010.

[20] A. Freeman, AngularJS Pro, APress, 2014.

[21] B. Green a S. Seshadri, AngularJS, O'Reilly Media, 2013.

[22] B. Borek, „Úvod do architektury MVC Zdroják“, 7 květen 2009. [Online]. Available: https://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/. [Přístup získán 28 prosinec 2015].

[23] „Confirm W3Schools“, [Online]. Available: http://www.w3schools.com/jsref/met_win_confirm.asp. [Přístup získán 1 leden 2016].

[24] „Angular Github“, [Online]. Available: https://github.com/angular/angular.js. [Přístup získán 25 březen 2016].

[25] „Canvas W3C“, [Online]. Available: https://www.w3.org/wiki/HTML/Elements/canvas. [Přístup získán 2 únor 2016].

[26] N. Downie. [Online]. Available: http://www.chartjs.org/. [Přístup získán 17 prosinec 2015].

[27] „Html DOM Events W3Schools“, [Online]. Available: http://www.w3schools.com/js/js_htmldom_events.asp. [Přístup získán 3 únor 2016].

[28] G. Kunz, „ChartistJS“, [Online]. Available: http://gionkunz.github.io/chartist-js/. [Přístup získán 16 listopad 2015].

[29] S. Tarimo, „Css Charts“, [Online]. Available: http://thysultan.com/projects/cssCharts/. [Přístup získán 13 leden 2016].

[30] P. Lewis a S. Thorogood, únor 2016. [Online]. Available: https://developers.google.com/web/fundamentals/design-and-ui/animations/animations-and-performance#css-vs-javascript-performance.

[31] C. Mills, „Developer Mozlila“, 29 únor 2016. [Online]. Available: https://developer.mozilla.org/en-US/Apps/Fundamentals/Performance/CSS_JavaScript_animation_performance. [Přístup získán 5 březen 2016].

[32] „HTML5 SVG W3Schools“, [Online]. Available: http://www.w3schools.com/html/html5_svg.asp. [Přístup získán 11 únor 2016].

[33] „Html DOM W3Schools“, [Online]. Available: http://www.w3schools.com/js/js_htmldom.asp. [Přístup získán 6 únor 2016].

[34] „The Internet Explorer Browser W3Schools“, [Online]. Available: http://www.w3schools.com/browsers/browsers_explorer.asp. [Přístup získán 7 březen 2016].

[35] C. Craig, „Tc angular chartjs“, [Online]. Available: http://carlcraig.github.io/tc-angular-chartjs/. [Přístup získán 12 únor 2016].

[36] C. Kulkarni, „Angular charts Github“, [Online]. Available: https://github.com/chinmaymk/angular-charts. [Přístup získán 11 únor 2016].

[37] M. Bostock, „D3JS“, [Online]. Available: https://d3js.org/. [Přístup získán 9 prosinec 2015].

[38] C. Maurer, „Angular NvD3 directives“, [Online]. Available: http://cmaurer.github.io/angularjs-nvd3-directives/multi.bar.chart.html. [Přístup získán 27 listopad 2015].

[39] „NvD3“, [Online]. Available: http://nvd3.org/. [Přístup získán 25 únor 2016].

[40] „Angular NvD3 Github“, [Online]. Available: https://github.com/krispo/angular-nvd3. [Přístup získán 1 únor 2016].

[41] R. Branas, AngularJS Essentials, Packt Publishing, 2014.

[42] B. Nadel, „Don't Forget Ben Nadel“, 25 říjen 2013. [Online]. Available: http://www.bennadel.com/blog/2548-don-t-forget-to-cancel-timeout-timers-in-your-destroy-events-in-angularjs.htm. [Přístup získán 12 leden 2016].

[43] T. Hønsi, „Hightchart“, [Online]. Available: http://www.highcharts.com/. [Přístup získán 7 leden 2016].

[44] „Path W3C“, 16 srpen 2011. [Online]. Available: https://www.w3.org/TR/SVG/paths.html#PathElement. [Přístup získán 3 leden 2016].

[45] J. Khan, „RIA Lab“, 28 červen 2011. [Online]. Available: https://jbkflex.wordpress.com/2011/07/28/creating-a-svg-pie-chart-html5/. [Přístup získán 18 leden 2016].

[46] J. Suchomel, „Parametrický popis křivek“, 2014. [Online]. Available: http://15122.fa.cvut.cz/?download=matematika/resene_priklady/parametricky_popis_krivek.pdf. [Přístup získán 7 leden 2016].

[47] „CSS3 Transitions W3Schools“, [Online]. Available: http://www.w3schools.com/css/css3_transitions.asp. [Přístup získán 7 leden 2016].

[48] „Ministerstvo školství, mládeže a tělovýchovy“, [Online]. Available: http://dsia.uiv.cz/vystupy/f2/f24.xls. [Přístup získán 17 leden 2016].

[49] V. Miksu, „JSX O moderních webových aplikacích“, [Online]. Available: https://www.dzejes.cz/react-jsx.html. [Přístup získán 14 únor 2016].

[50] C. Harrington, „Codementor“, [Online]. Available: https://www.codementor.io/reactjs/tutorial/react-vs-angularjs. [Přístup získán 12 únor 2016].

[51] D. Steigerwald, „React Zdroják“, 19 květen 2014. [Online]. Available:

https://www.zdrojak.cz/clanky/proc-facebook-react-zabil-jquery/. [Přístup získán 15 leden 2016].

[52] P. B. Kasper, „NgReact Github“, [Online]. Available: https://github.com/ngReact/ngReact. [Přístup získán 8 březen 2016].

[53] D. Vingrief, „JQueryScript“, 9 září 2015. [Online]. Available: http://www.jqueryscript.net/other/Basic-Line-Chart-Plugin-with-jQuery-and-Canvas-linechart-js.html. [Přístup získán 28 říjen 2015].

[54] B. Nadel, „JQuery events Ben Nadel“, 8 březen 2010. [Online]. Available: http://www.bennadel.com/blog/1869-jquery-mouse-events-pagex-y-vs-clientx-y.htm. [Přístup získán 25 leden 2016].

[55] D. Lamb, „jQuery vs. AngularJS Airpar“, [Online]. Available: https://www.airpair.com/angularjs/posts/jquery-angularjs-comparison-migration-walkthrough. [Přístup získán 18 leden 2016].

[56] Z. Kuhn, „Smahingboxes“, 21 říjen 2015. [Online]. Available: http://smashingboxes.com/blog/choosing-a-front-end-framework-angular-ember-react. [Přístup získán 15 leden 2016].

[57] B. Huber, „Wordcount Netbeans“, 22 duben 2007. [Online]. Available: http://plugins.netbeans.org/plugin/1531/wordcount. [Přístup získán 15 prosinec 2015].

[58] S. Panda, „Sitepoint“, 7 květen 2014. [Online]. Available: http://www.sitepoint.com/understanding-angulars-apply-digest/. [Přístup získán 28 listopad 2015].

[59] „Timeline Google Chrome“, [Online]. Available: https://developer.chrome.com/devtools/docs/timeline. [Přístup získán 5 únor 2016].

Přílohy

Seznam Obrázků

Seznam tabulek

Seznam grafů