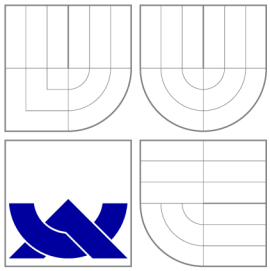
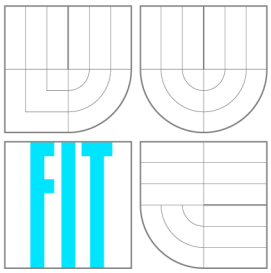


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

WEBOVÉ APLIKACE NA PLATFORMĚ NODE.JS

NODE.JS PLATFORM WEB APPLICATIONS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL ČERVENÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2016

Zadání bakalářské práce/18553/2015/xcerve15

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Červený Pavel**
Obor: Informační technologie
Téma: **Webové aplikace na platformě Node.js**
Web Applications on the Node.js Platform
Kategorie: Web

Pokyny:

1. Seznamte se s platformou Node.js pro tvorbu serverové části webové aplikace.
2. Prostudujte aplikační rámce Express.js, Angular.js, případně další související a seznamte se s databází MongoDB.
3. Navrhněte architekturu webové aplikace demonstrujícího využití výše uvedených technologií. Rozsah systému konzultujte s vedoucím.
4. Implementujte navrženou aplikaci na výše uvedené platformě.
5. Proveďte srovnání způsobu návrhu a tvorby systému na dané platformě oproti tradičním platformám s relační databází.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Rauch, G.: Smashing Node.js: JavaScript Everywhere, Wiley, 2012
- Žára, O.: JavaScript - Programátorské techniky a webové technologie, Computer Press, 2015

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Burget Radek, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato bakalářská práce se zabývá návrhem webové aplikace na platformě Node.js za použití dalších nástrojů a technologií, využívaných pro vývoj podobných aplikací. Hlavním cílem je vytvoření prototypu vzdělávací aplikace, pomocí které se uživatelé budou moci naučit základním dovednostem programovacího jazyka Javascript. V textu jsou postupně popsány technologie pro vytvoření aplikace, teorie provádění zdrojového kódu ve webové aplikaci, návrh aplikace, její implementace a testování.

Abstract

This bachelor's thesis deals with a design of a Node.js platform web application, using additional tools and technologies, which are used in this kind of development. The main aim of this application is to create a prototype of an educational application, which is going to be used for learning basics of Javascript language. This thesis is divided into several sections. Firstly it describes the technologies which are used to create this application, then a theory for executing source code in web application, design of the application itself, implementation and testing.

Klíčová slova

Node.js, Javascript, MEAN.js, Angular.js, Express.js, MongoDB, sandbox, jednostránková aplikace, uživatelská rozhraní, webové aplikace

Keywords

Node.js, Javascript, MEAN.js, Angular.js, Express.js, MongoDB, sandbox, single page application, user interfaces, web applications

Citace

ČERVENÝ, Pavel. *Webové aplikace na platformě Node.js*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Burget Radek.

Webové aplikace na platformě Node.js

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Červený
17. května 2016

Poděkování

Rád bych poděkoval panu Ing. Radku Burgetovi, Ph.D. za trpělivost a odborné rady při tvorbě této bakalářské práce.

Obsah

Kapitola 1

Úvod

Technologie pro tvorbu webových aplikací se vyvíjejí nepředvídatelně rychlým tempem. Od prvního představení programovacího jazyka Javascript, jehož účelem bylo vytvářet interaktivní rozhraní pro komunikaci s uživatelem, po vytvoření platformy Node.js, která vychází částečně z tohoto jazyka, a pomocí které se již nevytváří klientská část, ale ta serverová, a dává tak vzniknout novým možnostem zacházení s webovou tvorbou. Zároveň s Node.js přichází nové typy databází, jenž zpracovávají data do formátu javascriptových objektů a tím udávají společný cíl, tj. používat jediný programovací jazyk napříč celou aplikací.

Tento dokument popisuje tvorbu webové aplikace postavené na platformě Node.js. Účelem této aplikace je vytvořit vzdělávací nástroj pro výuku základů programovacího jazyka Javascript, za pomoci nejen platformy Node.js, ale celého balíku technologií známého pod zkratkou MEAN.js.

Druhá kapitola popisuje právě tyto technologie obsažené v balíku MEAN.js a na závěr nabídne krátké přirovnání k některým konvenčním technologiím, běžně používaným pro tvorbu webových aplikací.

Třetí kapitola se věnuje teorii týkající se vytváření výukového nástroje a použití technologií pro spouštění zdrojového kódu ve webové aplikaci. Na konci kapitoly jsou představeny některé typy existujících řešení vytvářené aplikace.

Čtvrtá kapitola řeší návrh takového typu aplikace z hlediska specifikace požadavků a způsobu práce s aplikací, případně možnosti ukládání dat.

Pátá kapitola předkládá popis již hotové implementace aplikace a ukazuje některé důležité části společné s krátkými ukázkami jejich řešení. Závěr kapitoly je určen pro testování.

Šestá kapitola prezentuje rozdíly mezi relační databází a nerelační databází použitou v tomto projektu a se věnuje diferenciacím v modelování a implementaci databáze. Zároveň poukazuje na odlišnosti mezi jazykem PHP a platformou Node.js.

Sedmá a poslední kapitola se věnuje závěru a zhodnocení práce.

Kapitola 2

Použité technologie

2.1 Node.js

Platforma Node.js je javascriptové běhové prostředí postavené na javascriptovém interpretu *V8*[1]od *Google Chrome*. Její funkcí je vytvářet vysoce škálovatelné aplikace, běžící na serverové části aplikace, nebo síťové aplikace.

Pro vytvoření souběžnosti (tj. spouštění více procesů najednou) na serverové části aplikace se používají hlavně dvě techniky: a to tzv. *vlákna* (angl. threads) nebo *události* (angl. events).

Vlákna se používají např. ve webovém serveru Apache. Nicméně problém s vlákny je ten, že při větším počtu se snižuje výkon celé aplikace, navíc nastává velká spotřeba operační paměti.

V druhém případě existují tzv. emitory, které generují události, dále existuje smyčka, jež postupně načítá tyto události nebo čeká až nějaké nastanou a potom pro každou událost spustí její obslužnou funkci, která vrátí nějaký stav. Tímto způsobem může být více akcí navázáno pouze na jedno vlákno. Rozdíl času při spouštění obslužné funkce a přepínáním mezi vlákny je značný. Tato architektura se nazývá *událostmi řízená* (angl. event-driven architecture).

Node.js je jednovláknový a implementuje tzv. asynchronní, událostmi řízené, běhové prostředí pro Javascipt. Asynchronní nebo-li neblokující, tj. např. při načítání dat se nečeká na to, až se operace dokončí a hned se přechází na další akci. Až se dokončí načítání, výsledek se analyzuje ve smyčce, která zpracovává události.

Dalšími důležitými znaky Node.js jsou:

- znovupoužitelný kód
- transport dat ve formátu JSON[2]
- rychlost, neboť se využívá interpret V8

[1]V8 (angl. *V8 Javascript engine*) je open-source projekt napsaný v jazyku C++ pro překlad Javasciptu, který byl použit v několika projektech včetně CouchDB, MongoDB nebo Node.js.

Překladač V8 překládá kód do strojového kódu předtím, než se kód spouští. To je rozdíl oproti tradiční technikám zpracování Javasciptu, kdy se kód interpretuje, tj. přímo převádí na spustitelné příkazy. Tento přístup zpracování umožňuje při překladu použít některé optimalizační techniky, které zlepšují výkon tohoto překladače. Všechny tyto vlastnosti se pozitivně odrážejí na celkovém výkonu.

[2]Javascript Object Notation - javascriptový zápis objektu a také formát pro přenos dat ve stylu klíč-hodnota.

Součástí ekosystému platformy Node.js je několik důležitých nástrojů, které se vyplatí znát. Jde hlavně o nástroj NPM (Node package manager), který se používá ke správě balíčků, rozšiřující možnosti aplikace pomocí znovupoužitelného kódu. Seznam všech balíčků, které se dají nainstalovat tímto nástrojem je možné nalézt na adrese: https://npmjs.org/. Jinou možností, jak se k tomuto seznamu dostat, je příkazová řádka NPM. Pro instalaci balíčku se používá příkaz *npm install název_balíčku*. Tento nástroj je nainstalován společně s Node.js.

Node.js lze využít pro tvorbu jednostránkové aplikace, chatovací služby, aplikačního rozhraní či datového streamování.

2.2 Express.js

Express.js je webový framework, který je založený na modulu *http* v Node.js a komponentách frameworku *Connect.js*. Tyto komponenty se nazývají *middleware* a jsou to základní stavební kameny tohoto frameworku. Tzv. middleware umožňuje vývojáři použít pro svůj projekt již vytvořený a otestovaný kód, který řeší většinu základních problémů jako např. rozbor *HTTP*[3] požadavků, rozbor *cookies*[4], práce s proměnnými v rámci jednoho sezení, jednoho přihlášení (*sessions*), tvorba rout, vyjmutí parametrů z *URL*[5], zvládání chyb a další.[8]

Middleware jsou funkce, které mají přístup k objektům *response* (res), *request* (req) a funkci *next()*, která slouží k případnému spuštění dalšího middleware. Express.js obsahuje tzv. zásobník všech middleware funkcí, které se postupně spouštějí při každém zaslaném HTTP požadavku. Middleware může být použit ke spouštění kódu, ke změnám v objektech response a request, k ukončení cyklu spouštění dalšího middleware, tj. k tomu, že daná middleware funkce po vykonání své činnosti nespustí funkci next(), nebo v pokračování ve vykonávání funkcí ze zásobníku. Middleware lze připojit k aplikaci pomocí metod *app.use()* nebo *app.VERB()* (*app.get* nebo *app.post*, tj. aktivace v závislosti na metodě zaslaného HTTP požadavku – *GET* nebo *POST*).[14]

Aplikace v Express.js se většinou skládá z několika po sobě jdoucích kroků(5.1):

- vložení závislostí, popř. knihoven třetích stran
- vytvoření objektu Express.js
- konfigurace aplikace, nastavení proměnných aplikace
- připojení k databázi
- definování funkcí middleware
- vytvoření rout
- samotný start aplikace, nastavení portu a jména

Express.js se instaluje pomocí *NPM* příkazem *npm install express*. Soubory se stejně jako všechny knihovny instalované tímto nástrojem uloží do složky *node_modules* v kořenu aplikace.

[3] Hypertext Trasfer Protocol - aplikační protokol pro přenos obsahu, základ pro funkci World Wide Web

[4] Také "browser cookies"- malá část dat vytvořená webovou stránkou a uložena u uživatele v prohlížeči na předem definovanou dobu

[5] Uniform Resource Locator - řetězec znaků s definovanou strukturou, který slouží k jednoznačné identifikaci zdrojů informací

2.3 Angular.js

Angular.js je javascriptový framework určený k vytváření uživatelských prostředí.

V tomto projektu se tento text zabývá verzí frameworku *1.x.x.* Je to zde připomenuto z toho důvodu, že nová verze *2.0.0* poměrně zásadně mění funkce některých částí, popř. je úplně odstraní.

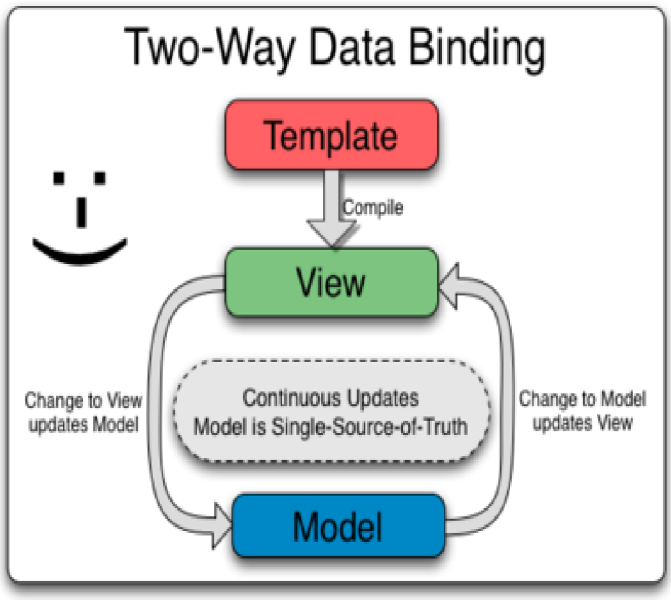
Každá Angular.js aplikace se skládá z hlavního konfiguračního souboru, který je většinou pojmenován jako *NPM* app.js, a dalších, které jsou rozděleny podle jejich funkcí a obsahují vlastní kód. Jsou to *direktivy* (angl. directives), *kontroléry* (angl. controllers), *filtry* (angl. filters) a *služby* (angl. services).

Začátek aplikace se určí pomocí modulu. Tento modul se definuje pomocí metody *angular.module(„název_aplikace“, [])*, název_aplikace je určen z *HTML*[6] direktivy *ng-app* v šabloně projektu. Poté, co je modul definován, funguje jako kontejner, do něhož mohou být přidávány kontroléry, vlastní direktivy apod.

Angular.js rozšiřuje možnosti HTML pomocí tzv. direktiv, které mohou být definované buď přímo v knihovně, nebo vytvořené vlastním kódem. Některé základní direktivy kromě již zmíněné ng-app je dále např. direktiva *ng-controller*, udávající jméno kontroléru, jenž řídí danou oblast nebo *ng-model*, která synchronizuje data od uživatele s aplikačními daty (hodí se použít např. u textových polí apod.).

Kontroléry spravují data a akce uživatele v oblasti, jež obsluhují, tato oblast je definována pomocí již zmíněné direktivy ng-controller, nebo je definována pro celou oblast šablony, kdy je kontrolér nastaven při vytváření routovacího systému. Kontrolér přistupuje k lokálním datům přes objekt *$scope*, pomocí tohoto objektu lze také vytvářet nové proměnné nebo funkce, ke kterým lze přistupovat v šabloně.

Angular.js definuje důležitý pojem a to tzv. *Two Way Data-Binding* (což lze přeložit jako dvoucestná synchronizace dat).



Obrázek 2.1: Dvoucestná synchronizace dat. Obr. [3]

[6]HyperText Markup Language - značkovací jazyk pro tvorbu webových stránek

Příklad použití: mějme textové pole, jemuž se nastaví model pomocí direktivy ng-model, takže pokaždé, když uživatel zadá do textového pole nějaká data, zároveň se zapíší do modelu (první cesta). Dále mějme nějaký prázdný odstavec, kam budeme vypisovat data (je nutné použít direktivu *ng-bind* pro připojení dat modelu, např. takto * * (druhá cesta). Poté cokoliv, co uživatel zadá do textového pole, se okamžitě zobrazí v tomto prázdném odstavci. Tento proces se vykoná automaticky bez jakéhokoliv dalšího kódu.[7]

V kontrolérech se obecně využívá kromě objektu $scope další tzv. služby (services) neboli závislosti (angl. *dependecy*). Tyto služby je nutné nějakým způsobem do kontroléru vložit, proto existuje návrhový vzor tzv. *Depedency Injection*.

Většina služeb se už, stejně jako u direktiv, nachází přímo v knihovně Angular.js (nicméně je tady možnost si vytvořit vlastní). Služby se kontroléru předávají stejným způsobem jako objekt $scope. Každá služba začíná symbolem dolaru ($). Několik základních služeb: *$http*, tato služba umožňuje vytvářet a zasílat HTTP požadavky na aplikační rozhraní serveru (*$http.get()*), *$location* je služba která pracuje s aktuální cestou a dokáže např. přesměrovat uživatele.

Angular.js je určen pouze pro vytváření aplikací na straně klienta, i přesto má k dispozici vlastní routovací systém, pomocí kterého lze přepínat mezi jednotlivými šablonami a vytvářet dojem, že se aplikace znovu nenačítá.

K tomuto frameworku existují stovky doplňků, které vývojářům usnadňují práci, neboť za ně dokáží vyřešit některé základní i pokročilé problémy. Všechny závislosti se musí uvést jako závislosti v konfiguračním souboru aplikace a v šablonách se musí nacházet všechny odkazy na zdrojové kódy používaných závislostí.

2.4 MongoDB

Databáze MongoDB je multiplatformní, dokumentově orientovaná. Je klasifikována jako *NoSQL*, tzn. že místo *relačních databází*, které ukládají data do tabulek s pevně daným schématem, MongoDB ukládá svá data do tzv. *dokumentů*, jejichž formát je dán jako JSON (MongoDB tento formát ještě rozšiřuje a značí ho jako *BSON*).

MongoDB zavádí pojmy jako např. *kolekce*, tj. kontejner pro dokumenty. Kolekce představují v relačních databázích tabulky. Kolekce mají na rozdíl od relačních databází dynamická schémata, jež nevyžadují plnit. Je možné například vkládat dokumenty, které mají odlišná pole. Nicméně všechny dokumenty v kolekci jsou určeny pro podobný účel.

Dokument se skládá z dat ve formátu BSON, tj. pole ve formátu *klíč-hodnota*. Dokument představuje v relačních databázích záznam v tabulce. Na rozdíl od relačních databází v MongoDB a obecně, v dokumentově orientovaných databázích neexistuje spojování tabulek, respektive dokumentů dohromady. Zavádí se ale pojem *vložený dokument* (angl. embedded document). Funguje to tak, že se do dokumentu vloží přímo ta data, na která by se v relačních databázích muselo odkazovat *cizím klíčem*, nicméně je možné použít i cizí klíče, tj. odkazy na jiné dokumenty, jimž se říká *reference*, a fungují podobně jako v relačních databázích. Každý dokument musí mít primární klíč, v tomto případě je to pole *_id*, které má datový typ *ObjectId*, který vytváří 12 bytový hexadecimální unikátní klíč.

Velmi jednoduchý dokument může vypadat takto:

```
{
    _id: ObjectId(7df72dd8902c),
    name: 'Espresso',
    price: 12.50
}
```

Zdrojový kód 2.1: Dokument v MongoDB

Dotazovací jazyk nad databází je poměrně bohatý, přesto je dobré si uvědomit některé základní příkazy. Všechny se vážou na objekt db, který odkazuje na právě používanou databázi.

Pro vkládání a přidávání do kolekce existují dvě metody *insert()* a *save()*. Rozdíl mezi nimi je, že insert() vždy vytváří nový dokument a save() buď vytvoří dokument, pokud chybí v předaném objektu pole _id, a pokud ne, tak aktualizuje dokument. Používá se: *db.fruit.insert(name: 'banana')*, fruit je název kolekce.

Dále pro vyhledávání a čtení dat se používá metoda *find()*. Použití této metody je velmi široké, lze vyhledávat nejenom podle hodnot, ale také používat různé další vlastnosti, které jsou známé i v relačních databázích. Jsou to matematické a logické operátory a jejich kombinace, také lze použít regulární výrazy.

Příklad: *db.fruit.find(name:$ne: 'banana')*, zobraz všechno ovoce kromě banánů ($ne je operátor *not equal*).

Podobně se používá metoda *update()*, pro aktualizaci záznamů. Dokument lze smazat např. podle jména pomocí metody *remove()*. Pokud je potřeba smazat celou kolekci, je možné použít metodu *drop()*[5].

Hlavní výhoda databáze MongoDB je hlavně dynamické schéma kolekcí. Docílení tohoto v relačních databázích by se špatně odrazilo na výkonu. Dále je to škálovatelnost, lze snadno připojit další server, a tím zvýšit rychlost databáze, a v neposlední řadě to také může být cena, vzhledem k tomu, že MongoDB je open-source projekt a může běžet na operačním systému Linux.

K nevýhodám patří vysoká spotřeba operační paměti (*RAM*), slabší kvalita dokumentace v porovnání s dokumentací k velkým projektům jako např. *Oracle* nebo *Microsoft SQL Server*. Minimum profesionálních nástrojů pro práci s databází. Absence transakcí a atomických operací. Některé tyto nevýhody jsou odůvodněny tím, že do relačních databázích se desítky let investovaly peníze a v podstatě není nutné z komerčního hlediska přecházet na jinou technologii.

2.5 MEAN vs. LAMP

Nadpis této kapitoly je možná trochu zavádějící, účelem není zjistit, jaký balík technologií, ať už je to *MEAN* nebo *LAMP*, je lepší. Má za úkol ukázat jejich rozdíly.

Balík MEAN se skládá z již popsaných technologií a to MongoDB, Express.js, Angular.js a Node.js. Naopak LAMP tvoří *Linux* (jako operační systém, na kterém balík běží), *Apache* (webový server), *MySQL* (relační databáze) a *PHP* (serverový skriptovací jazyk).

Nejprve operační systém. V případě LAMP je to zřejmé, že jde o Linux, což může být z určitého hlediska trochu limitující, na druhou stranu je nutné nezapomenout, že existují další mutace tohoto balíku např. *WAMP* (běžící na operačním systému Windows) a další. Jelikož je platforma Node.js multiplatformní, tak může fungovat kdekoliv.

Dále je potřeba webový server. LAMP definuje Apache jako svůj webový server, což je stabilní a poměrně pokročilá aplikace, pro kterou je vytvořeno mnoho doplňků, které ještě více rozšiřují schopnosti tohoto typu serveru. Na druhou stranu může být Apache pro velký počet požadavků pomalejší, než technologie v balíku MEAN. Ten nepoužívá žádnou takovou pokročilou technologii, ale k vytvoření webového serveru se použije přímo platforma Node.js, respektive framework Express.js, který je na ní postaven. Díky Node.js může být provoz serveru velmi svižný i za malých nároků na zdroje. Problémy mohou nastat při řešení nestandardních konfigurací, jejichž řešení může vyžadovat vytvoření vlastních doplňků pro Node.js.

Databáze v LAMP je relační MySQL. V balíku MEAN je to MongoDB, některé vlastnosti byly již rozebrány v této kapitole.(2.4)

Je nepochybné, že porovnávat serverový jazyk PHP a Angular.js je nesmysl. Pomocí jazyku PHP se vytváří serverový kód, podobně jako se to dělá pomocí frameworku Express.js, na rozdíl od Angular.js, který vytváří uživatelské rozhraní, které je přímo závislé na činnosti uživatele, když vytváří odpovědi na jeho akce.

Zjišťovat, jaký balík je lepší, je bezpředmětné, vždy to závisí na projektu, který se právě vytváří.

Kapitola 3

Použité techniky

Tato kapitola nejprve popisuje techniky a možnosti při spouštění zdrojového kódu, jak v čistém jazyku Javascript, tak na platformě Node.js. Poté uvede i pokročilé techniky zabezpečení a zobrazí některé nástroje, které se dají pro tento účel použít.

Další podkapitola se zabývá pojmem *single page application* nebo-li *SPA* a osvětluje jeho význam ve světě webových aplikací.

Na konci této kapitoly budou představena některá existující řešení, kterými se částečně inspiruje tento projekt.

3.1 Provádění kódu

Pokud má aplikace sama spouštět kód, v tomto případě jde o jazyk Javascript, je potřeba k tomuto účelu použít nějaký program, který tuto práci vykoná za nás. Tento program se nazývá *interpret*[1], jelikož interpretuje kód. Obyčejně je velmi obtížné pokoušet se vytvořit i velmi jednoduchý interpret programovacího jazyka. Naštěstí pro tento případ je to i zbytečné, neboť v jazyku Javascript existuje řešení. Je tím vestavěná funkce *eval()*.

3.1.1 Eval

Funkce eval() funguje velmi jednoduše. Jejím parametrem je řetězec, který obsahuje příkaz, výraz nebo posloupnost příkazů, také může obsahovat proměnné a vlastnosti existujících objektů. Eval() umožňuje dynamicky provádět tento předaný zdrojový kód.

Následující příklad ukazuje práci s funkcí eval() (zároveň se ale doporučuje rozhodně nepoužívat eval() tímto způsobem, protože demonstrovaný příklad zvládne jazyk Javascript i bez použití této funkce):

```
1 var a = 2;
2 var b = 40;
3 var c = eval("a + b"); // 42
```

Zdrojový kód 3.1: Ukázka funkce eval()

[1] Jazyk Javascript není překládán na strojový kód, ale je přímo převáděn na příkazy, které jsou ihned vykonávány pomocí některého z jeho interpretů, které bývají nejčastěji součástí webového prohlížeče.(*Google Chrome - V8*, *Mozilla Foundation - Rhino*)

Zranitelnost

Funkce eval() je velmi mocná, a proto je také snadno zneužitelná. Existuje skutečně velmi málo případů, kdy je použití této funkce nezbytné. Je to jedna z nejvíce nevhodně používaných funkcí[2].

Nicméně největší kámen úrazu při použití této funkce je samozřejmě v její bezpečnosti, respektive velmi snadnému zneužití. Funkce eval() přijme jako parametr cokoliv bez použití jakékoliv validace vstupu, tudíž pokud bude vstup generován od uživatele a přímo předáván jako parametr této funkci, je nemožné předpovídat výstup (předpokládáme, že ne všichni uživatelé budou vkládat to, co po nich chceme a spíše budou hledat cesty, jak zneužít tuto aplikaci). Nejčastěji se bude jednat o útoky typu *cross-site scripting* nebo-li *XSS*[3], tzn. vložení škodlivého kódu nebo pokusu o ovládnutí aplikace.

Eval() nemusí být nebezpečný, tj. do jisté míry. Pokud se použije uvnitř programu, který není závislý na vložení vstupu od uživatele, tak se vyloučí tato zranitelnost. Existují případy, kdy může být prospěšný, např. pokud je třeba vytvořit kompatibilitu se staršími prohlížeči.

3.1.2 VM modul v Node.js

VM (*virtual machine*) modul se chová podobně jako vestavěná funkce eval() v Javascriptu, také přijímá řetězce kódu, které následně interpretuje a provádí. Na rozdíl od funkce eval() tady jeho schopnosti nekončí. Obsahuje další pokročilé funkce, které eval() nezvládne, např. dokáže zkompilovat kód a pozdržet jeho vykonání nebo provádět kód v jiném *kontextu*[4], než ve kterém se nachází hlavní kód, toto je hlavní výhoda, co se týče bezpečnosti provádění kódu, respektive přístupu k datům, ke kterým by kód neměl mít přístup.

```
1 var a = 42; // global variable
2 function print(){
3   var b = 44; // local variable
4   console.log(a); // 42, global variable is always accessible
5   var a = 85; // local variable
6   console.log(a); // 85, local variable is going to replace global variable
        with same name
7 }
8 console.log(b); // undefined, not possible to access local context from
    global
```

Zdrojový kód 3.2: Rozdílné kontexty v JS

[2] Douglas Crockford - „eval is Evil. The eval function is the most misused feature of JavaScript. Avoid it“[2]

[3] *XSS* je typ útoku, jehož cílem je umístit do aplikace škodlivý kód za účelem poškození aplikace (v tom lepším případě) nebo získání citlivých údajů uživatelů využívající danou aplikaci. Existuje několik typů XSS, jde o trvalý typ, kdy je škodlivý kód uložen přímo v databázi aplikace (jedná se např. o komentáře u článků, které nebyly dostatečně ošetřeny proti tomuto typu útoku) a odrazový typ, zde se jedná o škodlivý kód, který je připojen k odkazu na známou webovou stránku. Po kliknutí na odkaz se uživateli zobrazí známá stránka, ale zároveň se spustí přidaný kód. O dalších možnostech útoku XSS a obraně je možné se dočíst zde[16].

[4] *Kontext* nebo-li *prováděcí kontext* je jedna z nejdůležitějších vlastností Javacriptu. Kontext proměnné nebo funkce definuje, k jakým datům má přístup a jak s nimi může pracovat. Každý prováděcí kontext obsahuje tzv. *objekt proměnných*, kde jsou definované všechny proměnné a funkce. Vně se nachází globální prováděcí kontext, který je v různých implementacích jiný, ve webových prohlížečích je to objekt *window* (tzv. všechny globální funkce a proměnné jsou vytvářeny jako vlastnosti tohoto objektu).[18]

- *vm.runInThisContext()* - jako první parametr má tato funkce kód ve formě řetězce, druhý parametr jsou nepovinné možnosti, zde je zajímavý zejména parametr *timeout*, který udává dobu, po kterou se bude kód provádět. Tato metoda umožňuje spouštět kód ve stejném kontextu jako zbytek aplikace, tzn. může přistupovat všechna globálně definovaná data, ale nemá přístup k datům lokálně definovaným, to je rozdíl oproti funkci eval().

- *vm.runInNewContext()* - taktéž jako předchozí metoda, má první parametr kód ve formě řetězce. Dalším parametrem je sandbox, což je objekt obsahující data. Z tohoto sandoboxu se poté vytvoří kontext, takže tato data budou přístupná pro spuštěný kód. Pokud je druhý parametr vynechán spustí se kód v prázdném kontextu. V tomto případě nemá spuštěný kód přístup k žádným datům hlavní aplikace.

I když se metoda *vm.runInNewContext()* zdá dostatečně bezpečná, protože neumožňuje přístup spouštěnému kódu k datům v aplikaci a kód běží ve vlastním kontextu, i tak se doporučuje zavést další opatření při spouštění nedůvěryhodného kódu.[12]

3.1.3 Sandbox pro nedůvěryhodný kód

Pokud bychom chtěli zpracovávat, respektive vyhodnocovat nedůvěryhodný kód, tj. kód, který nebyl dopředu validován a je možnost, že se jedná o škodlivý kód, tak budeme potřebovat nástroj, *sandbox*, který spuštění kódu oddělí od ostatních dat v hlavní aplikaci. Předchozí kapitoly ukázaly některé způsoby a zranitelnosti, se kterými se můžeme setkat. Nyní je jasné, že pro tento účel nelze použít jen funkci eval() a také, že není doporučené používat pouze modul *vm* v Node.js, bez jakýchkoliv úprav.

Existující řešení

Nyní budou předvedena některá řešení. Všechna zmíněná mají několik společných rysů. Využívají modul *vm*, nicméně přidávají další bezpečnostní opatření. A proto se dokážou vypořádat s různými typy útoků v jazyku Javascript. Nakonec je ale nutné podotknout, že jsou to nekomerční řešení, tudíž neexistuje garance 100% obrany proti všem typům útoků.

- *vm2*[5] - možnost spouštět i Node.js kód a další pokročilé vlastnosti

- *gf3 sandbox*[6] - podle diskuze je možné prolomit ochranu tohoto sandboxu, přesto je to velmi zajímavý nástroj

- *jailed*[7] - také poměrně pokročilý nástroj, pro větší ochranu využívá tzv. *Web Workers*(3.1.4).

3.1.4 Web Workers

Protože je Javascript od základu *jednovláknový*, bylo by obtížné ho přetransformovat na vícevláknový (vykonávání několika procesů najednou), proto bylo vytvořeno toto rozhraní Web Workers, které umožňuje spustit další úlohy v oddělených vláknech, které budou zcela

[5] https://github.com/patriksimek/vm2
[6] https://github.com/gf3/sandbox
[7] https://github.com/asvd/jailed

nezávislé na hlavním vlákně. Hlavní výhoda této činnosti je, že úlohy, které jsou časově a výkonnostně náročné, mohou běžet samostatně v pozadí, bez toho, aby jakýmkoliv způsobem zpomalovaly činnosti hlavní aplikace.

Funkce

Práce s *Web Workers* je v podstatě jednoduchá. *Worker* se vytvoří pomocí konstruktoru *Worker()*, kterému je jako parametr předán název souboru. V tomto souboru se nachází kód nějaké činnosti, která se spustí ve vlastním vláknu, které má úplně jiný globální kontext(ve webových prohlížečích je tímto globálním kontextem objekt window). Ve vlákně může běžet všechno až na pár výjimek. Tou nejdůležitější je, že kód nemůže přímo manipulovat s *DOM*[8]. Nicméně je možné pracovat s tzv. *WebSockets*(3.2.3) nebo úložištěm jako je *IndexedDB*(3.2.2) atd.[10]

Data si vlákna mezi sebou posílají přes systém zpráv. Worker může vytvářet další vlákna a také pracovat se sítí pomocí *XMLHttpRequest*(3.2.3).

Vyšší zabezpečení pro sandbox

Web Workers přidávají sandboxu postaveném na Node.js další ochranu a to, že je jejich činnost zabalena do vlákna, které je odděleno od hlavního toku programu. Na druhou stranu si ale musíme uvědomit, že i tak absolutní obrana pravděpodobně neexistuje.

3.2 Single Page Application

3.2.1 Historie webových aplikací

Rozdíl mezi webovou stránkou a webovou aplikací není tak jasný, jak by se mohlo zdát. Je to právě z důvodu, že některé webové stránky bývají označovány jako webové aplikace.

Jako webové aplikace se označují weby, které mají podobné charakteristiky jako aplikace pro osobní počítače či mobilní zařízení, ale přesto běží uvnitř webového prohlížeče.

Popularita webových aplikací tkví především v tom, že si uživatel pro práci s touto aplikací nemusí instalovat žádný další software do svého počítače a může aplikaci ihned používat.

Historie webu začíná především v devadesátých letech dvacátého století, kdy na tomto poli došlo k převratným změnám.

V roce 1990 Tim Berners-Lee představil protokol *HTTP*, software HTTP serveru, značkovací jazyk *HTML* a první webový prohlížeč pojmenovaný *WorldWideWeb*.

První webové stránky byly pouze statické, vytvořené pomocí HTML a poskytované prohlížeči pomocí webového serveru. Přechod na jinou stránku bylo možné pomocí hypertextových odkazů, nicméně každá taková operace vyžadovala nahrání stránky ze serveru a znovu načtení stránky prohlížečem.

V roce 1995 představila firma *Netscape* jazyk Javascript, který umožňoval spouštět skripty na straně klienta. To byl začátek dynamické změny bez nutnosti znovu načítat stránku a tudíž zjemnit práci uživatele s webovou stránkou. V těchto časech se, ale stále jednalo vždy o velmi jednoduché skripty, které vykonávaly pouze základní úkony.

[8]Document Object Model - objektový model dokumentu, nebo-li reprezentace (X)HTML nebo XML dokumentu

Příchod technologie *Flash* od společnosti *Macromedia* a malých aplikací nazvaných tzv. *Java applety*, by bylo možné nazvat počátky webových aplikací. Tyto aplikace byly vizuálně velmi působivé a mohly být zcela nezávislé na webovém serveru.

V roce 2005 se poprvé použila zkrakta *AJAX*(3.2.3). Tato technologie umožňuje získávat data z webového serveru bez znovu načítání stránky. A začínají se objevovat první důležité javascriptové frameworky[9].

V roce 2014 je představena aktualizace jazka HTML5(3.2.2), která přinesla velké změny a byl to další důležitý krok k vytváření SPA.[4]

3.2.2 HTML5

Značkovací jazyk HTML5 aktualizoval poslední verzi HTML 4.01 z roku 1999. Přestože hotová specifikace byla dokončena až v roce 2014, do webových prohlížečů byl jazyk implementován roky před tím.

Aktualizace přináší poměrně rozsáhlé změny a vylepšení. Přibyly nové *tagy*[10], jež mají zpřehlednit kód webových stránek, jedná se například o *nav* nebo *footer*, které mohou být použity místo často používaného konstruktu: * *, nebo tagy jako např. *video* a *audio*.

Kromě nových tagů, specifikace odhaluje aplikační rozhraní, které lze použít současně s jazykem Javascript. Patří sem např. rozhraní:

- *Canvas* - pro vkládání vektorové grafiky, nebo pro kreslení pomocí Javascriptu.
- *Web Storage* - pro ukládání dat do webového prohlížeče ve tvaru klíč-hodnota, toto rozhraní je podobné *Cookies*, ale na rozdíl od nich podporuje rozšířenou kapacitu dat.
- *Geolocation* - pro práci s geolokací (zjišťování polohy).
- *IndexedDB* - pro práci s jednoduchou databází v prohlížeči.

V současné době se již pracuje na nástupci, na verzi 5.1, která by měla být hotova na konci roku 2016.

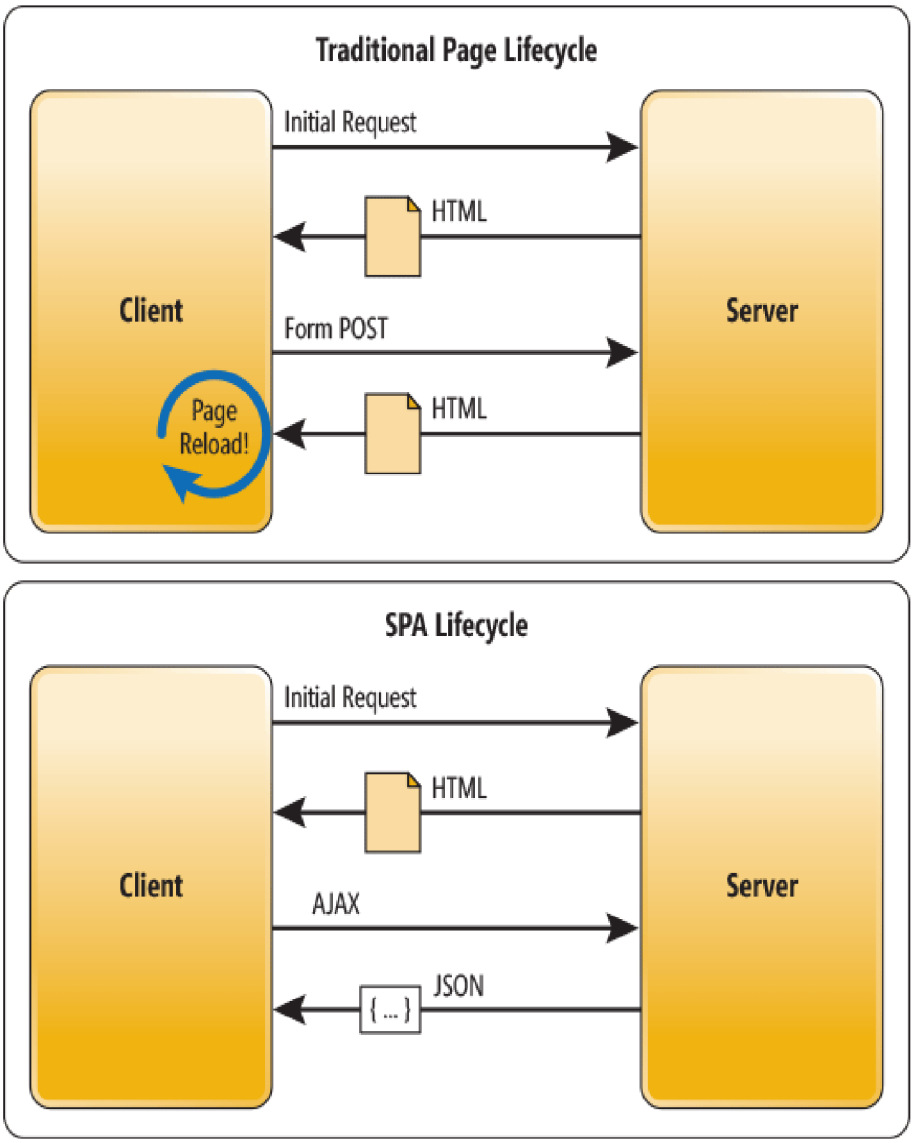
3.2.3 SPA

V definici je SPA označována jako aplikace, která má mít podobné vlastnosti a chování jako aplikace pro osobní počítač nebo mobilní zařízení. V praxi to znamená, že při prvním načtení webové aplikace se kromě statické HTML stránky načtou i všechny nezbytné soubory, ať už jsou to javascriptové soubory, CSS[11] soubory apod. Při práci s takovou aplikací se už webová stránka znovu nenačítá a všechny ostatní soubory jsou dynamicky načítány na základě uživatelových akcí, popř. se v závislosti stránka dynamicky překresluje. Následují obrázek představuje rozdíl mezi tradičními webovými stránkami a konceptem SPA.

[9] Knihovny nebo třídy znovupoužitelného kódu

[10] Značka - základní blok jazyka HTML

[11] Cascading Style Sheets - nebo-li kaskádové styly, používají se pro nastavení způsobu zobrazení elementů na stránkách vytvořených v jazycích (X)HTML nebo XML



Obrázek 3.1: Rozdíl mezi životním cyklem tradiční webové stránky a SPA. Obr. [9]

AJAX

AJAX (Asynchronous Javascript and *XML*[12]) je pravděpodobně nejrozšířenější možností, jak může komunikovat SPA se serverem. Tato dnes poměrně stará technologie je součástí téměř každé SPA. Její funkcí je asynchronně (v pozadí) nahrávat data ze serveru, bez toho aby se webová stránka musela pokaždé znovu načíst. Přestože se ve jméně objevuje XML, obecně se data posílají spíše ve formátu *JSON* (tj. poté AJAJ), popř. v dalších formátech.

[12]Extensible Markup Language - značkovací jazyk, který definuje pravidla pro formátování textu, který je zároveň přehledný pro člověka i stroj

```
1 //Text v souboru "ajax_info.txt" = "hello"
2 var xhr = new XMLHttpRequest();
3   xhr.onreadystatechange = function() {
4     if (xhr.readyState == 4 && xhr.status == 200) {
5       console.log(xhr.responseText); // hello
6     }
7   };
8   xhr.open("GET", "ajax_info.txt", true);
9   xhr.send();
```

Zdrojový kód 3.3: Jednoduchý příklad AJAXu

I přes nesporné výhody takového postupu, existují některé nevýhody. Jde např. o vyšší počet vyměňovaných HTTP požadavků, může proto dojít ke zpomalení, i přesto že jsou vyměňované požadavky malé. Dále např. pokud má uživatel nestabilní internetové připojení, může hrozit zpomalení nebo ztráta HTTP požadavků, které probíhají i po načtení stránky, a tak narušení práce uživatele. Nakonec i v dnešní době existuje skupina uživatelů, kteří mají v internetovém prohlížeči zakázán jazyk Javascript, i třeba z bezpečnostních důvodů. Těmto uživatelům potom nepůjde používat aplikaci využívající tuto technologii.

WebSockets

Tato technologie vytváří oboustranné spojení mezi klientem a serverem, kteří si potom mohou vyměňovat data v reálném čase. Tento způsob vychází částečně z technologie AJAX(3.2.3) a *Comet* (nestandardizované řešení na podobném principu). Je to řešení pro vytváření *real-timových* aplikací jako např. *chatů* nebo právě pro některé typy SPA. Výhoda WebSockets je také v tom, že na straně klienta je tato technologie implementována v jazyku Javascript.

Na straně serveru je to už komplikovanější, protože nestačí pouze jednoduchý HTTP server, ale specializovaný, který podporuje tuto technologii, nicméně existují implementace v jazyku *Python* nebo v Node.js.

Javascriptové frameworky

I když je možné vytvořit klientské rozhraní pouze v čistém Javascriptu, HTML a *CSS* bez použití jakýchkoliv pomocných knihoven, tak by se dalo říci, že je to nepraktické a vývojáře to zbytečně zdržuje od budování funkcionality aplikace. Proto existuje nespočet knihoven, tj. frameworků, které výrazně ulehčí důležité části práce na aplikaci. Důležité je vybrat ten správný.[6]

- *Angular.js*(2.3)
- *Ember.js* - framework, byl vypuštěn v roce 2011 a je primárně určen pro vývoj uživatelských aplikací pro web nebo pro vývoj aplikací pro osobní počítače. Za zmínku stojí např. aplikace Apple Music. Ember.js se sestává z pěti hlavních konceptů.
  Jde o:
  - *Routes* – stav aplikace na dané URL
  - *Models* – každá routa má přiřazený objekt model, obsahující asociovaná data aktuálního stavu aplikace

- *Templates* – šablony pro vytváření HTML částí aplikace, pomocí šablonového systému *HTMLBars* (odvozenina *Handlebars*[13])
- *Compontens* – vlastní HTML tagy, chování je vytvořeno pomocí Javascriptu a vzhled pomocí HTMLBars
- *Services* – objekty, které uchovávají permanentní data

Podobně jako Angular.js i Ember.js používá *dvoucestné vázání dat*[17]

- *React.js* – framework, který byl vypuštěn v roce 2013, za kterým stojí vývojáři z Facebooku a který vytváří uživatelské rozhraní, jak pro Facebook, tak Instagram. Je to jeden z nejrychleji rostoucích frameworků. Je vhodný pro vytváření a renderování komplexních uživatelských rozhraní. Hlavním znakem je koncept tzv. *virtual DOM*[1]. Funguje to tak, že se vytvoří kopie *DOM*, poté se při každé změně DOM (uživatelská akce) porovnají virtuální DOM a DOM, vyhodnotí se změny a pouze tyto se aplikují na DOM v prohlížeči. Další výhodou je používání tzv. *komponent*. Jde o znovupoužitelné kusy kódu, které upravují DOM. Existuje i celá knihovna těchto komponent.

- *Backbone.js* - na rozdíl od výšše zmíněných Backbone.js není tak komplexní. Je to odlehčený framework, který dodává strukturu kódu pro klientskou část a tak získává lepší přehlednost a udržitelnost pro dlouhodobé používání. Tento framework je i přes svoji jednoduchost často využíván mimojiné pro tvorbu SPA. Backbone.js je vyspělý nástroj s rozsáhlou škálou doplňků. Součástí je několik důležitých objektů jako *Model* obsahující kromě dat i jejich logiku, jako např. validaci nebo konverze, *Collection* uspořádaný seznam objektů Model, *Router* mapující funkcionalitu na jednotlivé URL, *View* znovupoužitelná část HTML uživatelského rozhraní sloučeného s objektem Model.[15]

REST

REST (Representational State Transfer) navrhnul a popsal Roy Fielding (spoluautor protokolu HTTP) ve své dizertační práci v roce 2000. REST je architektura navržená pro přístup k datům, není orientovaná jako *XML-RPC* nebo *SOAP*, které implementují vzdálené volání procedur.

Všechny zdroje mají své *URI* indentifikátory a REST definuje čtyři základní metody pro přístup k nim. Tyto metody jsou označeny jako *CRUD* (Create, Retrieve, Update, Delete), tj. vytvoření, získání, aktualizace a mazání dat. Systémy typu REST obvykle komunikují pomocí protokolu HTTP, tj. tyto základní CRUD metody jsou implementovány jako klíčová slova HTTP protokolu a to respektive *POST*, *GET*, *PUT* a *DELETE*.

př. (klíčové HTTP slovo a URI – identifikátor zdroje)

- *GET* /api/get/car – vrátí auto
- *POST* /api/insert/car – vloží nové auto
- *PUT* /api/update/car – aktualizuje auto (respektive nahradí stávající údaj novým)
- *DELETE* /api/delete/car – odstraní auto

[13] javascriptová knihovna pro vytváření šablon

Nevýhody SPA

Nevýhody týkající se SPA většinou plynou z toho, že se při vývoji webových technologií, prohlížečů a nástrojů se nepočítalo s vytvářením takového typu aplikace.

- *Historie webového prohlížeče* - protože je SPA definována jako jednostránková aplikace, tak tento koncept narušuje funkci tlačítek „Vpřed a Zpět“ ve webovém prohlížeči. Prohlížeč v tomto případě nemá informace o historii práce v aplikaci, protože URL aplikace je stále stejná. Tj. i když již uživatel několikrát klikl v aplikaci, potom tlačítko „Zpět“ odmaže všechna kliknutí a odkáže uživatele na poslední záznam ve své historii. Tento problém může být vyřešen např. pomocí HTML5(3.2.2), konkrétně pomocí funkcí *pushState()* a *replaceState()*, které dokáží přímo měnit historii ve webovém prohlížeči.

- *Pomalé počáteční načtení* - při prvním načtení SPA se před vykreslením HTML stránky načítají všechny soubory funkcionality aplikace, případně soubory frameworku a s ním související, což bývá značný počet, a proto může dojít k určitému zpomalení.

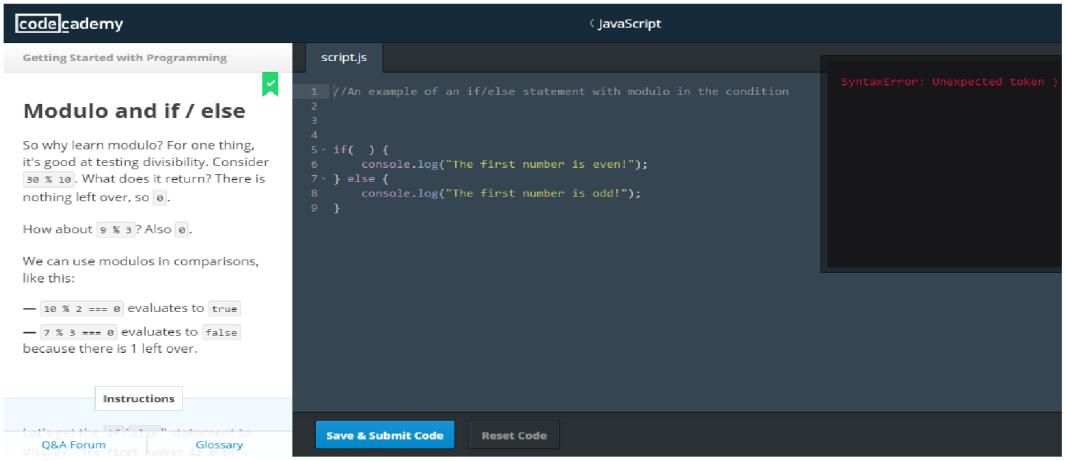
- *Automatické testování* - testovací framework se v tomto případě musí speciálně nastavit. Je to hlavně z důvodu asynchronního získávání dat, kdy je důležité nastavit, jak dlouho bude testovací framework čekat na vykonání činnosti. Také hrozí, že nakonec splnění testu bude trvat delší čas, než u obyčejné webové stránky.

3.3 Existující řešení

3.3.1 Codecademy

Webové výukové centrum *Codecademy* je pravděpodobně nejrozsáhlejší a nejpokročilejší platforma, která je stále zdarma. Nabízí výuku více jak sedmi programovacích jazyků a souvisejících témat, jako je příkazová řádka nebo tvorba webových stránek. Jednotlivé kurzy jsou rozděleny na souvislé podkapitoly, které jsou inteligentně řazeny od jednodušších ke složitějším.

Aby mohl uživatel aplikaci používat musí se zaregistrovat. Ve svém profilu si pak může prohlédnout kolik získal „odznaků“ za splněné kapitoly v jednotlivých výukových kurzech a které další může ještě získat. Uživatel získává pocit, že se nejen učí, ale i hraje hru. Při práci s editorem a při vyplňování úkolů má uživatel k dispozici nápovědu a když si neví rady, může si pomoci možností doplněním kódu od samotné aplikace.



Obrázek 3.2: Ukázka prostředí aplikace Codecademy.[14]

3.3.2 Khan Academy

Tato webová aplikace není přímo zaměřena na výuku programování (v této oblasti nenabízí tolik výukových kurzů jako dříve zmíněné Codecademy), ale i tak je zde několik kurzů pro výuku databázového jazyka *SQL*[15] nebo vytváření grafik pomocí jazyku Javascript.

Khan Academy nabízí kurzy matematiky, vědy, ekonomie apod. Jejich účelem je podpořit studenty, kterým např. nevyhovuje udávané tempo při vyučování ve škole a také pro rozšíření obzorů ve volném čase.

Pro studium ve zvolených kurzech není nutná registrace, ale být registrovaný samozřejmě přináší některé zajímavé bonusy. Může to být opět sbírání bodů za splnění úkolů v dané kapitole. Pokud je uživatel registrován, sleduje aplikace jeho pohyb ve vybrané kapitole a pokud např. rozpracuje nějaký úkol a nedokončí ho, při další přihlášením mu aplikace nabídne, aby pokračoval tam, kde přestal.

Jednotlivé kapitoly jsou kombinovány jako úkoly na doplnění. K dispozici jsou i výuková videa.

Další z výhod pro českého uživatele je, že aplikace je i v české mutaci (*Khanova škola*), to znamená překlad textů a české titulky nebo dabing k anglickým, výukovým videím.

3.3.3 Portál Learn

Tento portál je jeden ze skupiny webových stránek nazvaných „Learn“ + „název programovacího jazyka“(např. *LearnJava* nebo *LearnC*). Je to celkem sedm výukových kurzů.

Jednotlivé webové stránky jsou vzhledové velmi jednoduché, ztrácí se komfort z výukových aplikací dříve zmíněných.

Obsluha je také řešena jednoduše. Na úvodní straně jsou zobrazeny jednotlivé kapitoly, které odkazují přímo na výukový text. Přímo v tomto textu jsou krátké úryvku kódu, které je možné vyzkoušet spuštěním v editoru, který se zobrazí, bohužel velmi nešťastně, naspodu současné stránky.

K umocnění jednoduchosti se uživatel nemusí registrovat, i když ta možnost tady je.

[14] https://www.codecademy.com/

[15] Structured Query Language - strukturovaný dotazovací jazyk, který se používá pro práci s daty v relačních databázích

Inspirace z představených řešení

Aplikace Codecademy a KhanAcademy jsou pravděpodobně jedny z nejlepších aplikací určené pro výuku. Je to hlavně kvůli jejich rozhraní pro řešení jednotlivých úkolů. Webová stránka je rozdělena na editor, ve kterém je úryvek kódu, jenž se má doplnit, okno konzole, kde se vypisují výsledky, případně zobrazuje grafika a také vhodně umístěna nápověda.

Aplikace portálu Learn vyjadřují minimalismus a jednoduchost.

Kapitola 4

Návrh

Tato kapitola se věnuje specifikacím požadavků na funkcionalitu aplikace, které budou vyobrazeny ve speciálním tzv. diagram případů užití (angl. Use case diagram).

Dále se zabývá obecnou strukturou aplikace, respektive, jak jsou jednotlivé části do sebe zapojeny a jak jsou na sobě závislé. Důležité je také neopomenout strukturu souborů, která je u takovýchto projektů velmi důležitá.

Další část navrhne specifikaci aplikačního rozhraní (dále jen API), kde bude k dispozici seznam možných koncových bodů, pro přístup k datům a specifikaci datového modelu aplikace, která schématicky naznačí data uvnitř databázové aplikace.

Poslední stránky osvětlí rozhodnutí o realizaci grafického uživatelského rozhraní aplikace.

4.1 Specifikace požadavků

Cílem tohoto projektu bylo navrhnout aplikaci na platformě Node.js, která bude sloužit jako vzdělávací portál pro výuku programovacího jazyka Javascript.

Aplikace má uživateli nabídnout možnost si zaregistrovat profil, ve kterém budou uchována jak jeho osobní data, tak celkový postup při řešení jednotlivých výukových kapitol. Uživatel zároveň má možnost svá osobní data v profilu změnit.

Přístup k výukovým úkolům bude mít uživatel ve svém profilu, kde bude také zobrazeno, jaké úkoly již dokončil, jaký byl celkový čas řešení problému a jaký počet bodů za splnění získal. Zároveň budou k dispozici celkové statistiky ke každé kapitole, které pokrývají více úkolů najednou.

Během plnění problému daného úkolu, bude mít uživatel pomoc ve formě, buď komentovaného zdrojového kódu, nebo pomocných textů v určité nadepsané části obrazovky. Zároveň bude mít uživatel možnost sledovat svůj postup na výstupní konzoli, která zobrazí výsledek operace, popř. vypíše chyby a varovná hlášení, vždy když uživatel spustí překlad svého řešení problému.

Za každý bodovaný úkol získá hráč určitý počet bodů a také se mu započítá čas, který bude zobrazovat, jak dlouho uživateli trvalo vyřešení daného problému.

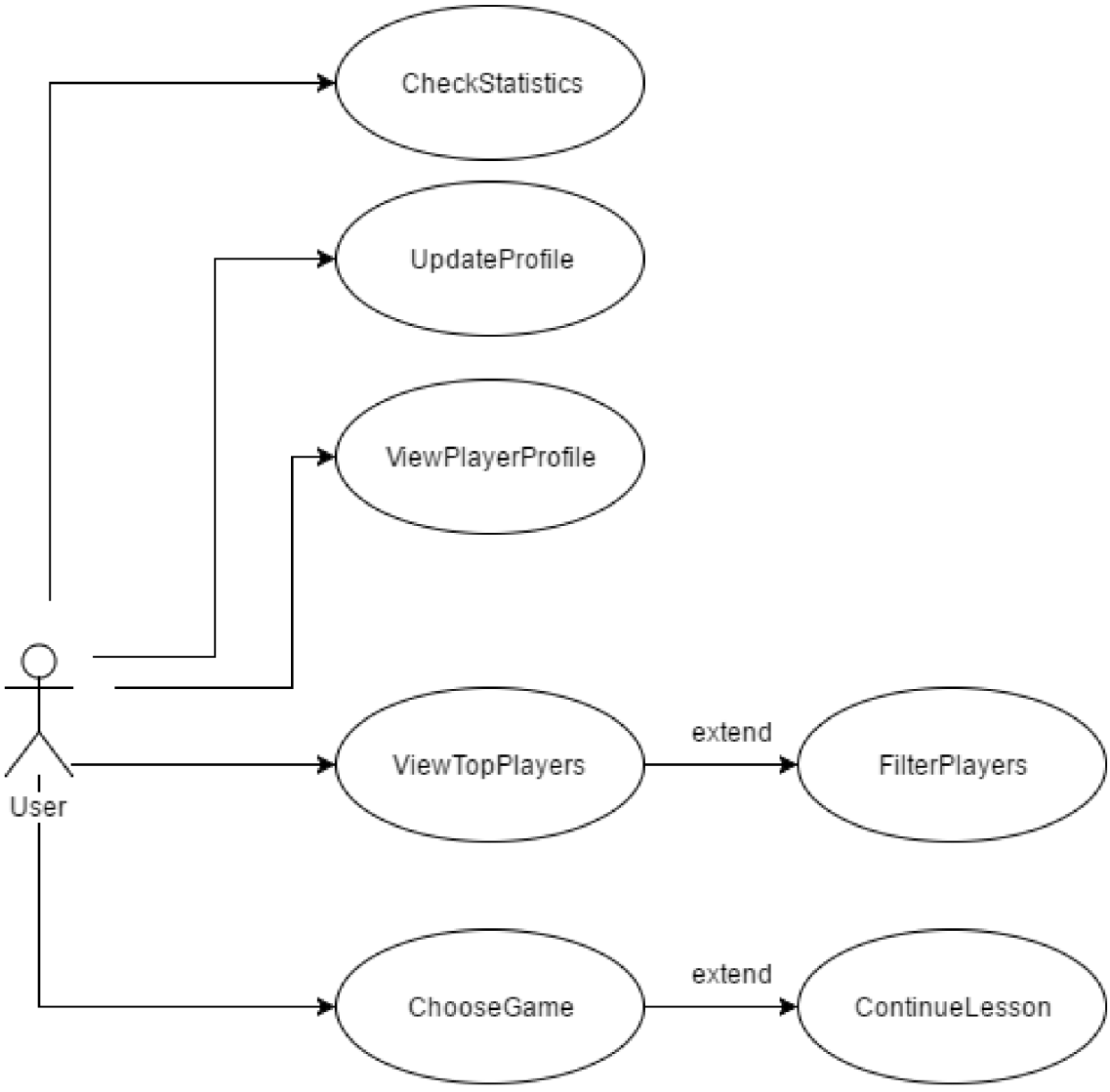
Uživatel bude mít také možnost porovnat své úspěchy s ostatními hráči, tzv. bude moci porovnat své nejlepší časy a maximální zisk bodů. popř. si zobrazí profil jiného hráče.

4.1.1 Diagram případů užití

Tento typ diagramu zobrazuje chování systému z pohledu uživatele, tj. nepopisuje jak jsou jednotlivé části implementovány.

Součástí každého diagramu jsou tyto charakteristiky: aktér (angl. actor), např. uživatel, dále případy užití (angl. use cases), tj. jednotlivé činnosti, které může uživatel vykonat a vztahy (angl. relations).

Následují obrázek zachycuje diagram případů užití mezi uživatelem a systémem, který popisuje tato práce.



Obrázek 4.1: Diagram případů užití.

4.2 Struktura aplikace

Při vytváření webové aplikace se programátor většinou nespoléhá pouze na jednu technologii a vlastní úsilí, ale je nasnadě kombinovat jich několik dohromady, za účelem rychlejší tvorby

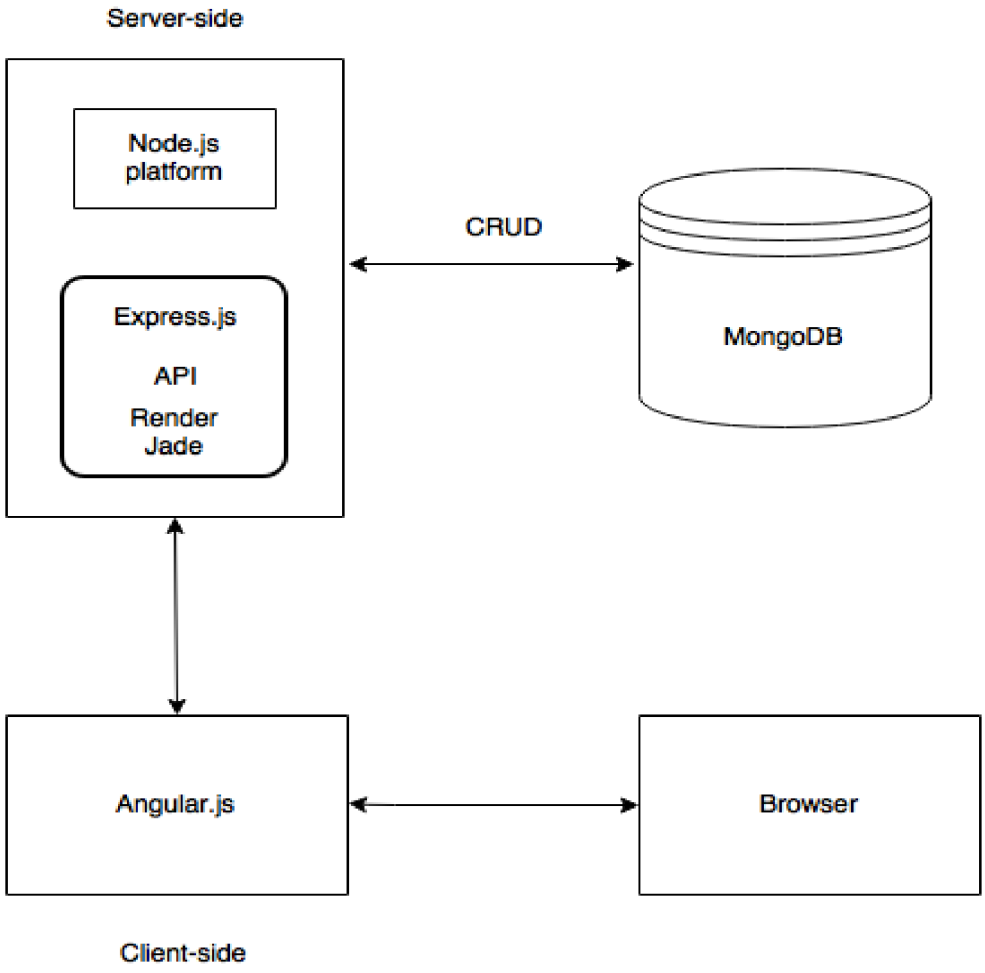
a využití znovupoužitelného, otestovaného kódu.

Jinak tomu není ani v případě tohoto projektu. Jak bylo řečeno v předchozích kapitolách, balík MEAN.js tvoří celkem čtyři technologie. Tyto technologie můžeme rozdělit v rámci jejich nasazení podle toho, jestli tvoří serverovou nebo klientskou část.

Hlavní je zde platforma Node.js, na které běží aplikační framework Express.js, ten je určen hlavně ke konfiguraci webového serveru, vytvoření REST API a načítání souborů, které jsou uloženy na serveru, webovému prohlížeči. V případě, že webová aplikace využívá tzv. šablonový systém, jehož šablony jsou vytvořeny pomocí speciálního jazyka (v tomto projektu se jedná o jazyk *Jade*) a je nutné tento jazyk přeložit na serveru, pak má framework Express.js i tuto funkci.

Další technologií je databáze MongoDB, která může a nemusí být spuštěna na stejném serveru jako ten webový. A nakonec framework Angular.js, který se používá pro vytváření uživatelských rozhraní a komunikaci se serverem pomocí REST API.

Následují velmi zjednodušený obrázek ukazuje symbiotické propojení čtyř hlavních technologií balíku MEAN.js.

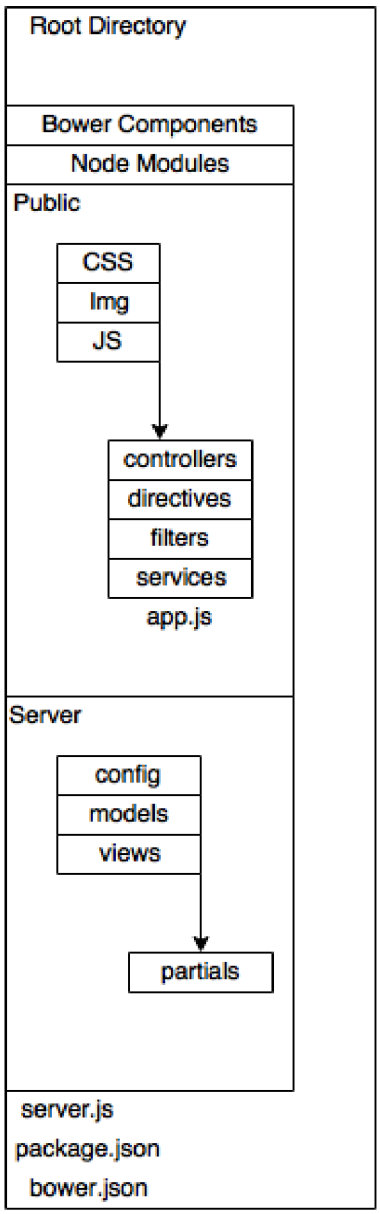


Obrázek 4.2: Struktura webové aplikace.

4.2.1 Adresářová struktura

Tato struktura může být u každého projektu odlišná, ale pokud se jedná o webovou aplikaci, existují některá základní pravidla, která se vyplatí dodržovat.

Předchozí kapitola rozdělila aplikaci podle toho jestli se akce provádějí na webovém serveru, tj. serverová část nebo u klienta, klientská část. Takto bude rozdělena i adresářová struktura.



Obrázek 4.3: Adresářová struktura.

Adresářová struktura se skládá z doplňků, např. soubory frameworku Angular, pro klientskou část, jsou to *Bower Compontents*, instalované nástrojem *Bower* (není vyžadované, aby se doplňky pro klientskou část instalovaly právě tímto nástrojem). Dále doplňky pro serverovou část *Node Modules*, instalované pomocí NPM.

Poslední tři soubory jsou *server.js*, což je inicializační soubor, který spouští webový ser-

ver, dále *package.json*, což je konfigurační soubor pro nástroj NPM, který obsahuje základní data o projektu, jeho jméno, autora apod., ale hlavně obsahuje všechny závislosti (angl. depedencies), které se mají nainstalovat, při prvním spuštěním projektu. Poslední soubor je konfigurační soubor pro nástroj Bower a to *bower.json*, podobně jako package.json, tento soubor také obsahuje závislosti, které se budou instalovat.

Adresář *Public* obsahuje soubory klientské části aplikace, jsou zde kaskádové styly, obrázky a soubory frameworku Angular.js v adresáři *JS*.

Existuje více způsobů, jak ukládat soubory frameworku Angular.js, v tomto případě jsou soubory ukládány podle jejich funkcí. Hlavní spouštěcí soubor je nazván *app.js*.

Adresář *Server* je pro soubory serverové části, je to například specifikace pro REST API, schémata databázových objektů (adresář *models*) nebo šablony a šablonové části v adresáři *views*. V tomto případě jsou šablony uloženy v serverové části, ale to jen proto, že jsou zde také vykreslovány. Obecně mohou být i v klientské části, např. pokud se nepoužije na jejich tvorbu jazyk Jade, nebo podobný.

4.3 Aplikační rozhraní

Aplikační rozhraní vytváří spojení mezi klientskou a serverovou vrstvou. Jeho hlavní funkcí jsou operace CRUD (Create, Retrieve, Update, Delete) nad databází.

Následující tabulka zobrazuje všechny koncové body, pro přístup ke zdrojům.

Zdroj	**Popis**
GET /api/users/single/:id	Získej uživatele podle ID
GET /api/users/single/:username	Získej ID uživatele podle uživatelského jména
GET /api/users/all	Získej všechny uživatele
GET /api/tasks/single/:id	Získej úkol podle ID
GET /api/tasks/all	Získej všechny úkoly
POST /api/users/games/update	Aktualizuj hru
POST /api/users/profiles/update	Aktualizuj profil
POST /api/users/login	Přihlas uživatele
POST /api/users/signup	Registruj uživatele
GET /api/users/logout	Odhlas uživatele
POST /api/games/run_code	Spusť zdrojový kód

Tabulka 4.1: API koncové body.

V příloze je možné nalézt celkový souhrn koncových bodů a přenášených dat.(A.1)

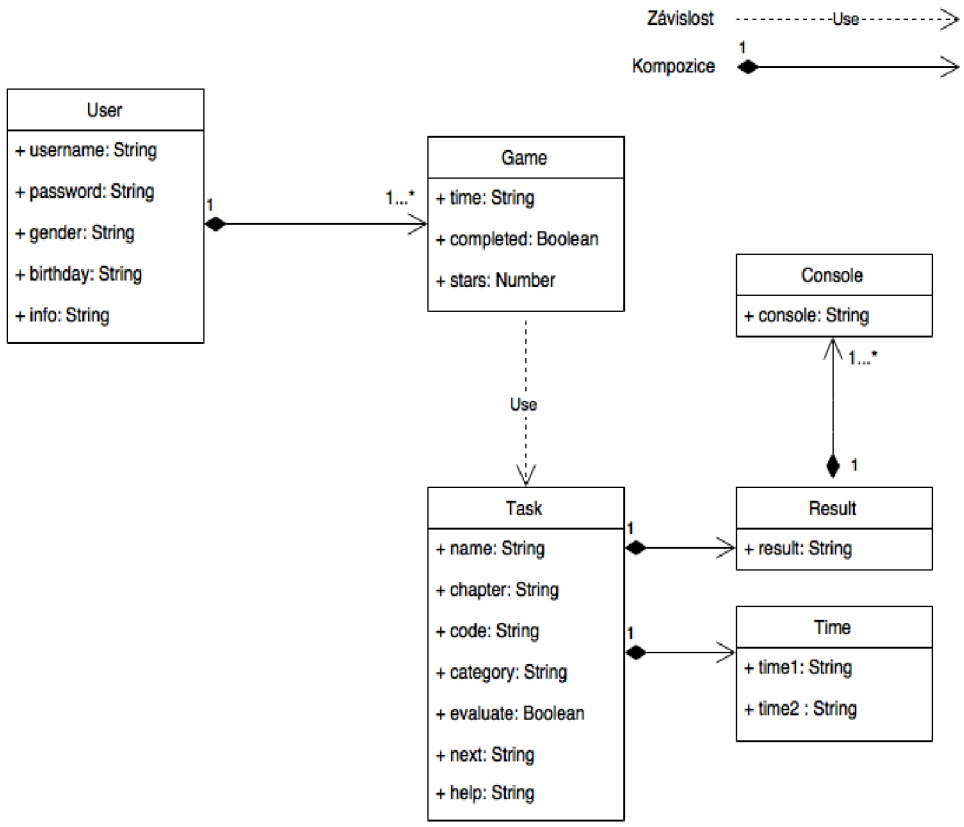
4.4 Datový model

Tento model zobrazuje, jak jsou aplikační data uložena, respektive jakou mají strukturu.

Protože databáze MongoDB je typu *NoSQL*, to znamená, že má tzv. flexibilní schéma (v běžných relačních databází musí vkládaná data vždy odpovídat vytvořenému schématu, to v tomto případě neplatí). Navíc jsou data v databázi MongoDB uložena jako JSON (nebo-li BSON podle MongoDB), tzn. že dokumenty mohou obsahovat reference na jiné dokumenty (podobnost s cizími klíči z relačních databází), vložené dokumenty nebo pole těchto vložených dokumentů nebo jiné kombinace.

Jak tj. potom tato data graficky modelovat, když obsahují tyto odlišnosti? V praxi obecné řešení neexistuje, nicméně se doporučuje použít jazyk UML (angl. unified modeling language), konkrétně diagram tříd (angl. class diagram).

Následující schéma se pokusí pomocí některých prvků z UML definovat aplikační data v databázi MongoDB tohoto projektu. Kromě diagramů tříd, používá toto schéma vztahy pro *Závislost* (angl. Depedency), tento vztah bude reprezentovat referenci jiného dokumentu a vztah *Kompozice* (angl. Composition), který bude značit vložené dokumenty.



Obrázek 4.4: Datový model projektu.

V kapitole o specifikaci aplikace byly představeny některé požadavky, kterých by měla aplikace dosáhnout. Jde hlavně o ukládání dat o uživateli a to zajišťuje dokument *User*.

Dále je to uložení informací o výukových úkolech, jejich zdrojový kód, který bude nahrán do editoru, jak se úkol nazývá a do které kapitoly patří. Důležité je si všimnout vložených dokumentů, jde o *Time*, ve kterém budou časy, jenž jsou řazeny od nejmenšího po nejvyšší, tzn. čas, za který uživatel získá náležitý počet bodů. A poté dokument *Result*, který bude určovat, správný výsledek, jenž se zobrazí, po spuštění zdrojového kódu. Je dobré si uvědomit, že výsledků v konzoli může být několik najednou.

Za zmínku také stojí vlastnost *evaluate* v dokumentu *Task*. Tato vlastnost určuje, zda bude tento výukový úkol bodovaný. Pokud úkol nebude bodovaný, potom budou vložené dokumenty Result a Time prázdné, to flexibilní schéma umožňuje.

Aby bylo možné uchovávat postup uživatele, při řešení jednotlivých výukových úkolů, obsahuje dokument User pole vložených dokumentů *Game*. Tyto dokumenty se nastavují při registraci uživatele. Každý obsahuje referenci na zadaný úkol, vlastnost, jestli už byl úkol

vyřešen, pokud ano, tak čas, který uživateli trval ho vyřešit a počet bodů, který za tento čas získal.

4.5 Uživatelské rozhraní

Důležitým faktorem při tvorbě uživatelského rozhraní bylo dodržet filozofii SPA tak, aby se výsledná aplikace svým stylem podobala desktopové. Základní šablona osobního profilu se snaží vyhovět takovým stylem, že kopíruje jednoduché rozvržení dvou sloupců, vlevo je kontrolní panel, který obsahuje odkazy do další částí aplikace a vpravo je datová část.

Šablony pro přihlášení a registraci jsou svým stylem velmi jednoduché a obsahují pouze funkční prvky, které se od nich vyžadují. Při registraci je nutné zadat pouze minimum dat, aplikace cílí na uživatele, kteří nechtějí ztrácet čas vypisováním různých nepodstatných údajů. Po registraci je okamžitě možné aplikaci používat bez nutnosti ověřování registrovaného emailu apod.

Šablona editoru pro vypracování problému je koncipována podobně jako výše uvedené příklady existujících řešení. Součástí je editor, do kterého se bude zadávat řešení problému, tento editor vhodně zvýrazňuje zdrojový kód a má i jiné pokročilé vlastnosti. Dále na pravé straně od editoru je nejprve konzole, ve které se zobrazuje výsledek spouštěného kódu a kde se také případně zobrazují chyby nebo varování. Pod částí s konzolí se nachází sekce s nápovědou, kde se uživatel může inspirovat v případě, že nebude vědět, jak by měl pokračovat. Zdrojový kód se spustí po kliknutí na tlačítko „Run code“, pokud bude výsledek chybný, zobrazí se v konzoli výsledek a chybové hlášení oznamující nesprávný výsledek. Pokud bude výsledek správný objeví se okno s hlášením, které bude oznamovat správně vyřešený úkol a zároveň, pokud se bude jednat o bodovaný úkol, zobrazí se celkový čas a počet získaných bodů.

V rámci profilu si bude moci uživatel aktualizovat údaje nebo zobrazit ostatní uživatele aplikace, mezi nimiž bude moci filtrovat.

Kapitola 5

Implementace a testování

5.1 Konfigurace webového serveru

Předtím, než začne práce na funkcionalitě aplikace, jejím uživatelském prostředí apod., tak je nutné nakonfigurovat a spustit samotný webový server.

Toho lze dosáhnout již použitím pouze samotného Node.js, ale mnohem zajímavější a bezesporu rychlejší je aplikovat framework Express.js, který v tomto případě ještě kromě spuštění serveru naimportuje pomocné moduly a knihovny.

Konfigurační kód bude uložen ve spouštěcím souboru. Tento soubor se většinou pojmenovává jako server.js.

Na prvních řádcích budou nejprve načteny pomocné moduly a framework Express.js samotný pomocí metody „require()“, tj. takto:

```
1 var express = require('express');
```

Zdrojový kód 5.1: Import frameworku Express.js

Obyčejně se importuje několik základních modulů (ale není to nezbytné), jedná se o:

- *path* - modul pro práci s cestami a manipulaci s nimi, užitečné při práci s různými operačními systémy, které mohou stejnou cestu interpretovat jinak
- *cookie-parser* - modul pro rozkódování *cookies* a vytvoření objektů s názvy těchto cookies a uložení do proměnné *req.cookies*
- *body-parser* – modul pro práci a rozkódování příchozích HTTP datových požadavků, jedná se například o data zaslaná pomocí metody POST, vytvořená po odeslání formuláře (např. registrace nového uživatele), data jsou poté přístupná v *req.body*
- *morgan* – modul pro zaznamenávání příchozích požadavků a odpovědí, hodí se zejména pro vývoj aplikace, kdy je nutné mít přehled o probíhajících činnostech aplikace
- *express-session* – vytváření permanentních proměnných na serveru
- *mongoose* – pokud je aplikace závislá na databázi MongoDB, je dobré použít tento modul, který vytváří další pomocnou vrstvu mezi aplikací a databází a zjednodušuje tak práci, např. lze jím vytvořit předlohy objektů *JSON*, které fungují jako schémata skutečných dokumentů v databázi MongoDB

Poté, co se naimportují knihovny, vytvoří se odkaz na aplikaci Express.js, příkazem:

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

Zdrojový kód 5.2: Vytvoření aplikace Express.js

Nyní se musí nastavit globální proměnné a nakonfigurovat načtené knihovny.

```
app.set('views', 'path to files');
```

Zdrojový kód 5.3: Nastavení přístupu k šablonám

Nastavení vykreslovacího mechanismu (pokud aplikace žádný nepoužívá, druhý parametr bude 'html'):

```
app.set('view engine', 'engine name (eg. jade, ejs)');
// if there isn't an engine, second param will be 'html'
```

Zdrojový kód 5.4: Nastavení vykreslovacího mechanismu

U globálních proměnných jde hlavně o dvě věci a to nastavení prostředí, tj. jestli se jedná o produkční nebo vývojové, popř. jiné. Tato proměnná je pouze účelová, určuje, jestli se budou používat knihovny a nástroje vhodné během vývoje (např. modul *morgan*). A další je nastavení portu, na kterém bude webový server naslouchat (odkud bude přístupný).

Tyto proměnné se nastavují podobně jako jakékoliv jiné proměnné v Javascriptu, ale využívají se obvykle systémové proměnné *process.env*.

```
var env = process.env.NODE_ENV =  process.env.NODE_ENV || 'dev'; // use value
    of NODE_ENV or set both env and NODE_ENV to 'dev'
var port = process.env.PORT || 3000; // set port
```

Zdrojový kód 5.5: Nastavení systémových proměnných

Dále se nakonfigurují načtené knihovny, pro tento účel se používá slovo *use*, tj. *app.use(modul)*.

```
app.use(logger(env));
app.use(cookieParser());
app.use(bodyParser.json());
// etc.
```

Zdrojový kód 5.6: Konfigurace knihoven

Po nakonfigurování knihoven se vytvoří aplikační rozhraní (API). A nakonec se spustí webový server příkazem:

```
app.listen(port);
```

Zdrojový kód 5.7: Spuštění webového serveru

5.2 Databáze

Součástí tohoto projektu je databáze MongoDB, proto se pro zjednodušení práce s databází použil modul *mongoose*, aby usnadnil operace nad databází.

5.2.1 Připojení k databázi

```
var mongoose = require('mongoose');
  var db_name = 'mongodb://127.0.0.1/bc'; //  127.0.0.1 = hostname, bc =
      database name
  mongoose.connect(db_name);
  var db = mongoose.connection;
  db.on('error', console.error.bind(console, 'connection error:'));
  db.once('open', function() { // we're connected! });
```

Zdrojový kód 5.8: Připojení k databázi

V tomto případě se databáze nachází na stejném počítači, jako běží tato aplikace, nicméně v případě, že bude databáze na jiném serveru, stačí jednoduše změnit adresu v proměnné *db_name* (*hostname*), respektive změnit název databáze.

5.2.2 Vytvoření schémat

Výhoda při použití mongoose je také v tom, že není nutné přímo v databázi definovat schémata kolekcí, ale stačí, když se vytvoří přímo ve zdrojovém kódu.

Schéma uživatele vytvořené v mongoose:

```
var userSchema = mongoose.Schema({
    username: String,
    email: String,
    gender: String,
    birthday: Date,
    info: String,
    games: [{
      _id: {type: Number, ref: 'Task'},
      time: String,
      completed: Boolean,
      stars: Number
    }]
  });
  mongoose.model('User', userSchema);
```

Zdrojový kód 5.9: Vytvoření schématu pro uživatele

V tomto úryvku je nejdůležitější zřejmě poslední krok, jde o vytvoření modelu. Model je konstrukt, který je vytvořen právě z výše definovaného schématu. Tento model je takovým prostředníkem mezi aplikací a databází, má definované statické metody, pomocí kterých se poté provádějí operace v databázi, jde např. o vyhledávání metodami (*find()* nebo *findOne()*, kde jejich parametry jsou buď konkrétní identifikační čísla dokumentů, nebo jejich charakteristiky). Kromě základních metod, lze modelu přidat i vlastní. Př. metody pro hašování hesla pomocí modulu *bcrypt* ve schématu uživatele:

```
var bcrypt = require('bcrypt-nodejs');
  userSchema.methods.generateHash = function(password) {
    return bcrypt.hashSync(password, bcrypt.genSaltSync(8), null);
  };
```

Zdrojový kód 5.10: Vlastní metoda modelu User

Její použití je poté při ukládání nového uživatele do databáze, kdy se hašuje heslo uživatele.

I ve tomto schématu je vidět, že se zde používá odkaz na model *Task*, v tomto případě je zde použit jako reference na *_id* konkrétní instanci modelu Task.

Tvorba schématu nabízí podobné možnosti jako vytváření kolekcí v databázi MongoDB. Mongoose také nabízí základní typy *JSON* jako *String*, *Number*, *Array* apod., ale i základní typy z *BSON*, které stačí pro vytváření nekomplexních schémat kolekcí.

5.3 Tvorba API

Z obr.(4.2) v kapitole *Struktura aplikace* je zřejmé, že API bude plnit dvě hlavní činnosti. Bude přistupovat k datům v databázi a vykreslovat šablony vzhledů a pak data z databáze a HTML kód vykreslené šablony bude zasílat klientské části aplikace.

Rozhraní se vytvoří pomocí frameworku Express.js, není nutné instalovat nějaký další doplněk. Před skutečnou tvorbou API je nutné vysvětlit routování v Express.js.

5.3.1 Routování v Express.js

Routování v základu znamená, jak aplikace odpoví na požadavek od uživatele na konkrétní *koncový bod*, který je definován jako *URI* a který je zaslán specifickou HTTP metodou (může jít o GET, POST apod.). Každá *„routa“* může mít jednu či více zpracovávajících funkcí (angl. handler), které jsou spuštěny, když se shoduje URI a cesta v definované routě.

Routa má následující definici v Express.js: *app.METHOD (PATH, HANDLER)*, kde *app* je instance Express.js, *METHOD* je typ HTTP požadavku, *PATH* je cesta na serveru, *HANDLER* je funkce, která je spuštěna, když se cesta v routě shoduje.[13]

Př. routy, která podle předaného parametru id vyhledá odpovídajícího uživatele a zašle ho zpět klientovi

```
1 var app = express();
2   app.get('/api/users/single/:id', function(req,res) {
3     var id = req.params.id;
4     User.findOne({_id:id}, function(req,res) {
5       res.send(user);
6     });
7   });
```

Zdrojový kód 5.11: Routa pro vyhledání uživatele podle ID

V tomto případě je metoda GET, cesta je *'/api/users/single/:id'*, s tím že*:id*, značí, že informace za posledním lomítkem je *id* hledaného uživatele.

Funkce, která se spouští při shodě má dva parametry, *req*, který obsahuje data příchozího požadavku. Pomocí objektu req lze přistupovat, buď k parametrům v URI, jako zde – *req.params.id* nebo k datům poslaných metodou POST. Tato data jsou doplňkem *body-parser* připojena k objektu req a lze k nim přistupovat přes *req.body*. *Res* je objekt, který se použije pro vytvoření odpovědi. Uživatel se vyhledá pomocí statické metody modelu User, která má přístup k této kolekci dokumentů v databázi MongoDB. Výsledek této operace se odešle do klientské části metodou *send*.

5.3.2 Vykreslení šablon

Šablony jsou na serveru uloženy ve složce *views*, kde se nachází soubor *layout.jade*, jenž obsahuje šablonu hlavičky, společně s odkazy na soubory s kaskádovými styly, dále soubor

index.jade, ke kterému je připojen soubor layout.jade a odkazy na soubory s Javascriptem (jde především o aplikaci Angular.js a pomocné knihovny).

Nicméně nejdůležitější částí tohoto souboru je HTML značka *div*, která obsahuje direktivu *ng-view* knihovny Angular.js. Tato direktiva říká, že do tohoto místa se budou načítat *částečné šablony* ze složky *partials*, která obsahuje právě tyto soubory, pro každou routu jiné.

Express.js pro vykreslení používá funkci *render*, která se váže na objekt *res*, podobně jako funkce *send*.

V API jsou dvě routy, jež tuto funkci využívají, je to funkce pro vykreslení *indexu* (index.jade ve složce views), tj. původní stránka, ve které jsou odkazy na všechny podpůrné skripty a kaskádové styly:

```
1  app.get('/', function(req,res) {
2    res.render('index');
3  });
```

Zdrojový kód 5.12: Routa pro vykreslení počáteční stránky

Tato šablona se vykreslí, když se uživatel poprvé dostane na úvodní stranu aplikace.

Druhá routa je pro vykreslování částečných šablon uložených ve složce partials, jejíž cesta bere jako argument jméno té části, která se má v tu chvíli vykreslit.

```
1  app.get('/partials/:name', function(req,res) {
2    var name = req.params.name;
3    res.render('partials/'+name);
4  });
```

Zdrojový kód 5.13: Routa pro vykreslení částečných šablon

Angular.js má také svůj routovací systém, nicméně je nutné podotknout, že tento systém pracuje pouze v klientské části, tj. nemá přístup k HTTP požadavkům jako systém v Express.js.

Tento routovací systém vypadá takto:

```
1  app.config(function($routeProvider, $locationProvider){
2    $routeProvider.
3      when('/login', {
4        templateUrl: 'partials/login',
5        controller: LoginCtrl
6      });
7
8    $locationProvider.html5Mode(true);
9  });
```

Zdrojový kód 5.14: Routovací systém Angular.js

Podobně jako v případě Express.js je i zde *app* jako odkaz na instanci, v tomto případě aplikace Angular.js. Předaná funkce má dva parametry *$routeProvider*, který nastavuje routovací systém a *$locationProvider*, který uvádí, jak se bude chovat odkazování v rámci aplikace, zde je nastaven *html5Mode*, což znamená přístup k historii prohlížeče a tj. že se bude jednat o SPA aplikaci.

Parametr *$routeProvider* vytváří jednotlivé routy. Stejně jako v Express.js i zde je cesta, která když se shoduje, tak spouští další činnosti. Část *templateUrl* odkazuje na šablonu, která se bude vykreslovat, zde je poznat, že když se pošle požadavek GET na adresu *'partials/login'*, tak ji routovací systém Express.js zachytí a vykreslí danou šablonu a HTML

kód pošle nazpět. Poslední část *controller* udává, který obslužný kód spravuje vykreslenou šablonu.

5.4 Klientská část

Klientská část je tvořena frameworkem Angular.js, vykreslenými šablonami, jejichž vzhled je kombinací vlastních CSS stylů a frameworkem *Bootstrap*.

Hlavní část tvoří index.jade, ve kterém jsou uvedeny odkazy na všechny pomocné knihovny, zdrojové kódy frameworku Angular.js a samotná aplikace vytvořená v tomto frameworku. Zároveň je v tomto souboru definován název a začátek Angular.js aplikace, pomocí direktivy ng-app, která je připojena k úvodní HTML značce.

V kapitole *Adresářová struktura*(4.2.1) bylo uvedeno, že hlavní část Angular.js aplikace tvoří soubor *app.js*. V tomto souboru je definována aplikace a všechny její doplňkové moduly (závislosti):

```
1  var app = angular.module('myApp', ['ngRoute', 'ngAnimate', 'ui.bootstrap',
       'ngSanize', 'UserValidation', 'ui.ace', 'ngIdle', '720kb.datepicker']);
```

Zdrojový kód 5.15: Závisloti aplikace Angular.js

Textový řetězec *'myApp'* značí název aplikace, který stejně uveden v direktivě *ng-app*, dále následují všechny závislosti, které je nutné načíst, aby se daly použít dále v aplikaci.

Tento soubor funguje spíše jako konfigurace pro Angular.js aplikaci. Dále se tady definuje systém routování viz předchozí kapitola, popř. další systémy, které bude se budou používat v celé aplikaci.

Další aplikační soubory jsou uloženy v adresářích pojmenovaných podle své funkce (*controllers*, *directives*, *filters*, *services*).

5.4.1 Kontrolér

Kontrolér je řídící mechanismus pro všechny operace, které se odehrávají v šabloně. Do šablony lze buď vložit direktivu *ng-controller='název'* nebo kontrolér definovat již při vytváření rout. Kontrolér není určen k tomu, aby měnil *DOM*, na to jsou direktivy

Kontrolér může být definovát více způsoby např. takto:

```
1 function LoginCtrl($scope, $http, $location){
2     ...
3   }
```

Zdrojový kód 5.16: Definování kontroléru

Jak bylo výše zmíněno, účelem kontroléru je řídit, to co se děje uvnitř šablony, tj. např. jak s ní interaguje uživatel apod. To, jakým způsobem je to možné, zcela závisí na objektu *$scope*. Tento objekt umožňuje nastavovat proměnné, které se dají vypsat v šabloně takto:

`{{ název_proměnné }}`

. Zároveň lze nastavovat funkce, které se dají spustit jako reakce na uživatelovu aktivitu.

```
1 $scope.greetings = 'Hello';
```

Zdrojový kód 5.17: Definování proměnné v šabloně

v šabloně poté:

```
<h1>{{ název_proměnné }}</h1>
```

```
1 $scope.greetings = function() {
2     console.log('Hello');
3   };
```

Zdrojový kód 5.18: Definování funkce v šabloně

v šabloně:

```
<button ng-click="greetings()">Print hello</button>
```

Kontroléry v projektu

- *EditorCtrl* – kontrolér pro okno editoru, obsahuje metody pro zobrazení dialogového okna při správném vyřešení úkolu, pro znovu načtení okna editoru, přesměrování na profil uživatele, načtení dat úkolu z databáze, zaslání dat z editoru na server, kde se spouští v sandboxu.
- *EditorModalCtrl* – kontrolér pro dialogové okno, obsahuje metody pro znovu načtení okna editoru, přesměrování na profil uživatele a přesměrování na další lekci
- *LoginCtrl* – kontrolér pro přihlašování, obsahuje metodu pro odeslání přihlašovacích údajů na server
- *SignupCtrl* – kontrolér pro registrování uživatele, obsahuje metodu pro odeslání registračních údajů na server
- *ProfileCtrl* – kontrolér pro profil uživatele, obsahuje metody pro odhlášení uživatele, vytvoření proměnných obsahující postup v plnění úkolů a statistik
- *ProfileCheckCtrl* – kontrolér pro zobrazení cizího profilu uživatele, obsahuje proměnné s daty o uživateli
- *ProfileUpdateCtrl* – kontrolér pro aktualizaci profilu, obsahuje metodu pro odeslání aktualizačních dat na server
- *TopCtrl* – kontrolér pro zobrazení nejlepších hráčů, obsahuje metody pro stránkování, nastavuje proměnné s daty o uživatelích

5.4.2 Vlastní direktivy

Direktivy jsou atributy na HTML značkách, které říkají překladači Angular.js, aby k této značce připojil další kód.

V tomto projektu existují tři vlastní direktivy, které slouží k validaci zadaných údajů při registraci, jde o validaci uživatelského jména, hesla a potvrzení zadaného hesla.

Direktiva se definuje podobně jako se může definovat kontrolér:

```
1 angular.module('UserValidation',[]).directive('validUsername', function() {
2     ...
3   })
```

Zdrojový kód 5.19: Definice vlastní direktivy

Tento modul *UserValidation* se poté připojí do konfigurace aplikace jako jedna ze závislostí. Sama direktiva se nazývá *validUsername*.

Pomocí direktiv lze vytvářet různé mechanismy jak manipulovat s DOM, validovat data apod.

Následující zjednodušený příklad ukáže pouze jednu z možných situací, kdy je vhodné vytvořit vlastní direktivu, v tomto případě tj. validaci dat.

```
1  .directive('validUsername', function() {
2    return {
3      require: 'ngModel', // we need to access data in directive
4      link: function(scope, elm, attr, ctrl) { // this fuction will be linked
           with this directive
5        // every time that value inside this directive change, this function
           will be triggered, viewValue is current value in model of
           directive
6        ctrl.$parsers.unshift(function (viewValue) {
7          var isBlank = viewValue === ''; // test if value is blank
8          ctrl.$setValidity('isBlank', !isBlank); // set error if so
9          return viewValue; // set data value in directive
10       }
11     }
12   }
13 });
```

Zdrojový kód 5.20: Direktiva pro validaci uživatelského jména

Funkce této direktivy je celkem prostá, jedná se o kontrolu toho, že případné textové pole není prázdné, tj. u formulářů HTML je to příznak *required*:

```
<input type="text" name="username" required>
```

Tato direktiva má jméno *validUsername*, které se poté v HTML používá jako *valid-username*.

Tato direktiva potřebuje mít přístup k hodnotě uvnitř textového pole, tj. potřebuje mít přístup k jeho modelu, proto je zde řádek *require: 'ngModel'*.

Dále při každé změně hodnoty uvnitř textového pole dojde ke spuštění funkce *ctrl.$parsers.unshift*, v této funkci se testuje, jestli je hodnota prázdná a dále se nastavuje validační chyba pomocí *ctrl.$setValidity*. Důležité je upozornit, že se v tomto případě nastavuje validita, to znamená, jestli hodnota uvnitř textového pole je správná. V případě že je proměnná *isBlank* nastavena na *logickou* hodnotu *true*, tzn. pravda, tak se validita nastaví na *false* (*!isBlank*). Pokud je hodnota textového pole nevalidní, nastaví se chyba, ke které lze přistoupit v šabloně a to takto: *$error.isBlank* je nastaveno na *true*.

Nakonec direktiva zpátky nastaví hodnotu uvnitř modelu textového pole.

5.5 Autentifikace

Pro přihlašování a registraci nových uživatelů tento projekt kombinuje možnosti frameworku Express.js i Angular.js. Express.js validuje data přes databázi (např. pokud je už uživatelské jméno registrováno), ukládá data do databáze a vytváří permanentní proměnné na serveru, které se zničí při odhlášení, tzv. *sezení* (*session*). Angular.js validuje data při zadávání uživatelem viz vlastní direktivy a odesílá data na server.

5.5.1 Autentifikace na serveru

Pro autentifikaci se na serveru používá doplněk *passport.js*. Tento doplněk pracuje s tzv. strategiemi, každá *strategie* nastavuje parametry pro jiný typ autentifikace, v tomto projektu je použita pouze jedna a to *localStrategy*, nicméně tento doplněk nabízí i autentifikaci pomocí *Google+* nebo *Facebooku*.

Funkce je v zásadě prostá, po zachycení HTTP požadavku typu POST na routu *'/login'* (přihlášení), respektive *'/signup'* (registrace) se spustí obslužná funkce s patřičným zdrojovým kódem.

Tento kód má dvě části. V první části se spouští (v případě přihlašování) akce, které ověřují uživatelské jméno a heslo vůči databázi, pokud nastane chyba, ukončuje se tato část a vrací se chybové hlášení a status chyby. Pokud vše proběhne v pořádku, přejde se na druhou část, kdy se použije metoda doplňku passport.js a to *logIn()*. Tato metoda, pokud proběhne v pořádku, tak vytvoří permanentní proměnnou sezení (session) na serveru, která znamená přihlášení uživatele a pak odešle na klientskou část odpověď serveru na přihlášení ve formě statusu úspěšného přihlášení a identifikačního čísla přihlášeného uživatele.

Při odhlášení se použije metoda doplňku passport.js *logout()*, která odstraní sezení a odešle odpověď do klientské části.

Registrování uživatele funguje podobně jako přihlášení, jen se navíc ukládá do databáze nový uživatel.

5.5.2 Autentifikace na straně klienta

Pro validaci dat jsou vytvořeny direktivy viz výše. Chybová hlášení jsou zobrazována, pokud jsou nastaveny chybové proměnné typu *$error.název_chyby*.

Pro autentifikaci jsou vytvořeny kontroléry *LoginCtrl* a *SignupCtrl*, jež fungují podobně, mají jednu metodu a to *login()*, respektive *signup()*, jejichž funkcí je odesílat data, které do nich vložil uživatel. Tato data jsou uložena ve formátu JSON, jež jsou pomocí služby *$http.post()* odeslána na server ke zpracování.

Pokud vše proběhne v pořádku a ze serveru přijde potvrzovací status a identifikační číslo uživatele, tak se toto číslo uloží do úložného prostoru HTML5 API, tzv. localStorage, takto: *localStorage.setItem('user', user);* A uživatel je poté přesměrován na svůj profil.

Pokud dojde k chybě, tj. v odpovědi ze serveru přijde chybový status, tak se v kontroléru vytvoří proměnná, která bude obsahovat chybovou zprávu ze serveru, jež bude poté zobrazena uživateli v šabloně.

5.6 Testování

Testování této aplikace nebylo příliš rozsáhlé. V zásadě lze rozdělit na testování funkčnosti a testování použitelnosti.

Testování funkčnosti

Toto testování bylo soustředěno na splnění specifikací požadavků z kapitoly Návrh(4). Dále šlo o testování jejich funkčnosti. Toto testování probíhalo již během vývoje aplikace a také bylo v zásadě provedeno při testování použitelnosti.

Testování použitelnosti

Testování použitelnosti znamená zkoušení produktu reprezentativní skupinou uživatelů. Během testování účastníci plní předem zadané úkoly podle daného scénáře, zatímco operátoři testu pozorují účastníky a dělají si poznámky o jejich práci. Účelem tohoto typu testování není nalézt chyby nebo nesrovnalosti v aplikaci vůči specifikaci požadavků, ale jde o identifikování problémů při užívání aplikace, tj. jakousi spokojenost uživatele během jeho práce.

Vytvoření účelného testovacího scénáře je poměrně složité. Je nutné se zaměřit nejenom na hlavní cíle aplikace, které se mají otestovat, ale také na logické okolnosti, které souvisí s určitým úkolem, tj. že zadané úkoly by měly určitým způsobem odrážet reálné požadavky uživatele.[11]

Scénář testování v tomto projektu byl poměrně hodně zjednodušen a jednotlivé úkoly jsou:

1. Pokus se zaregistrovat pomocí registračního formuláře
2. Vyhledej uživatele „sarah“ a zobraz si její profil
3. Dokonči úkol Basic operators v kapitole Introduction

Výsledky testování jsou následující:

1. V tomto úkole nenastaly větší obtíže, pravděpodobně proto, že je to velmi jednoduchá registrace s málo poli na vyplnění. Pouze jednou se stalo, že si účastník nevšiml značky pro nepovinné pole a zbytečně ho vyplnil.
2. Zde účastníci nejprve většinou nevěděli, jak se dostat k vyhledávání. Po nasměrování na odkaz Top players, bylo již samotné vyhledávání jednoduché. Někteří ocenili rychlost a flexibilitu vyhledávání. Jako vylepšení pro tuto část bylo navrženo, aby po vyhledání uživatele bylo v tabulce vidět, na které pozici v celkovém seznam je.
3. Účastníci byli povětšinou spokojeni s rozložením na stránce a rozdělení okna editoru, výsledků, respektive pomoci. Někteří měli problém s pochopením, jaký úkol mají vlastně řešit. Navíc testování objevilo funkční problém, kdy se chyby syntaxe nezobrazovaly na výstupu. Po splnění úkolu, účastníci ocenili zobrazení dialogového okna a snadnou orientaci cesty zpátky na profil nebo znovu načtení úkolu. Jako vylepšení se nabízí lepší využití pomoci. Když se uživatel zasekne a neví jak má pokračovat, tak by se mohla po nějakém časovém intervalu objevit nápověda.

Závěr

Na základě testování bylo zjištěno, že klíčové části pracují správně. Orientace v aplikaci je, až na výjimky, snadná. I když se objevily připomínky na přílišnou strohost profilu v aplikaci, tím se upevňuje původní návrh zaměřený na jednoduchost prostředí. Připomínky na vylepšení byly brány v potaz a v případě dalšího rozšiřování aplikace by se určitě využily.

Kapitola 6

Rozdíly při použití relační databáze

Nejprve je důležité si uvědomit, jaké rozdíly nastanou, pokud se pro projekt použije tradiční platforma s relační databází, např. LAMP (Linux, Apace, MySQL, PHP). Jak již bylo zmíněno v kapitole MEAN vs. LAMP(2.5) a v kapitole o MongoDB(2.4), největší rozdíly se nachází v přístupu k práci s daty, tj. např. jejich vytváření, čtení apod. Vytvářená webová aplikace bude vždy derivací typu klient-server, a proto to, jakým způsobem se vytvoří klientská, respektive serverová část aplikace, bude víceméně podobné. Určité rozdíly budou maximálně v odlišnostech programovacích jazyků, nicméně pokud by se použily technologie z balíku MEAN.js a pouze by se nahradil typ databáze z MongoDB na MySQL, pak by se odstranily i tyto rozdíly.

Relační databáze	**Mongo**
Databáze	Databáze
Tabulka	Kolekce
Řádek tabulky	Dokument
Sloupec tabulky	Pole v dokumentu
Operace JOIN	Vložený dokument
Primární klíč	Pole ObjectID

Tabulka 6.1: Rozdíly mezi relační a MongoDB databází.

Zároveň MongoDB obsahuje tzv. reference, tj. odkazy na jiné dokumenty v databázi. Tyto reference nelze úplně přirovnat k cizím klíčům v relačních databázích, protože ony nevytvářejí žádné omezení při manipulaci s daty.

Před tím, než se definuje databáze a její části, je nutné vytvořit její schéma.

U relačních databází se ve velké míře využívá tzv. ER (entity relationship) diagram, pomocí kterého se naznačí struktura dat, jejich logické celky (tabulky), zde představeny jako entity a vztahy mezi nimi.

Tento projekt nevyužívá relační databázi, ale tzv. NoSQL dokumentově orientovanou databázi nazvanou MongoDB. Jak bylo řečeno v kapitole o Datovém modelu(4.4), problémem takových typů databází je v tom, že k popisu jejích funkčních celků neexistuje zavedená technika. A spíše se využívá diagram tříd jazyka UML, který jak je z názvu patrné, nebyl přímo vytvořen pro tento účel.

Při modelování dat je dobré znát rozdíl mezi oběma typy databází. Zatímco u relačních

databází se modelování řídí normalizačními pravidly, kdy jde o odstranění redundantních dat, zjednodušení schématu apod., tak u MongoDB jde v mnoha případech o opak.

U relačních databázích se musí pečlivě vytvořit schéma pro každou tabulku, je nutné mít úplnou představu, jakým způsobem budou data strukturovaná, protože struktura dat v relačních databázích je fixní.

Naopak je tomu u NoSQL. Tento typ má flexibilní schéma dat, tzn. že během manipulace s daty se nemusí struktura dodržovat. Proto při modelování NoSQL databáze se volí přístup podle toho, jakým způsobem bude výsledná aplikace s daty pracovat a ne naopak.

Další důležitá část u obou typů databází je modelování vztahů. Hlavní tři typy jsou 1:1 (one to one), 1:M (one to many) a M:N (many to many).

Vztah 1:1 se v relačních databázích řeší tak, že se přidá entitě další atributy, respektive se přidají další sloupce do tabulky. V MongoDB se to řeší obdobně, tj. přidají se další pole do dokumentu, popř. se přidá vložený dokument.

Vztah 1:M v relačních databázích znamená vytvořit novou tabulku a její hodnoty odkazovat, pomocí cizích klíčů (cizí klíče odkazují na indexy primárního klíče odkazované tabulky), tam kam je potřeba. V MongoDB se naopak vytvoří pole vložených dokumentů, které představují v relačních databázích onu nově vytvořenou tabulku. V tomto vloženém dokumentu jsou všechny informace a jsou přímo přístupné z původního dokumentu. Existuje ještě druhá možnost, která velmi připomíná řešení z relačních databází, a to vytvoření tzv. pole referencí na jiné dokumenty, které obsahují potřebná data, stejně jako nová tabulka v relační databázi.

Vztah M:N znamená v relační databázi vytvoření speciální tabulky, která bude uchovávat identifikační klíče obou tabulek a unikátní klíč (primární klíč této tabulky) bude složen z obou dvou klíčů. V MongoDB lze tento vztah modelovat podobně jako vztah 1:M pro všechny dokumenty patřící do N.

Pro vytvoření databáze, tabulek a obecně práci s relační databází se využívá deklarativní dotazovací jazyk SQL a jeho mutace pro různá řešení relačních databází (Oracle, Microsoft SQL Sever).

Dotazovací jazyk nad MongoDB byl již částečně představen v kapitole MongoDB(2.4).

Pokud se tento jazyk používá pro operace s kolekcí, tak všechny jeho metody mají tento zápis: *db.collection.method()*, kdy objekt db, je odkaz na databázi, se kterou se právě pracuje a collection je kolekce se kterou se má manipulovat. Tyto metody přebírají parametry ve formátu JSON.

Příklad vložení v SQL a NoSQL:

```
INSERT INTO book (
    `ISBN`, `title`, `author`
)
VALUES (
    '9780992461256',
    '1984',
    'George Orwell'
);
```

Zdrojový kód 6.1: Vložení pomocí SQL

```
db.book.insert({
  ISBN: "9780992461256",
  title: "1984",
  author: "George Orwell"
});
```

Zdrojový kód 6.2: Vložení pomocí NoSQL

Kapitola 7

Závěr

Cílem bakalářské práce bylo vytvoření webové aplikace na platformě Node.js, s využitím technologického balíku MEAN.js, který kromě platformy Node.js, obsahuje frameworky Express.js, Angular.js a NoSQL databázi MongoDB. Konkrétně šlo o aplikaci pro výuku programovacího jazyka ve formě doplňování zdrojových kódů, respektive řešení hádanek.

Důležitou částí projektu bylo vytvoření mechanismu, který by bezpečným způsobem dokázal interpretovat kód tak, aby nebylo možné (nebo aby se alespoň snížila pravděpodobnost), že bude narušena integrita aplikace. Tzn. že případný útočník po vložení škodlivého kódu jako odpovědi na některou z hádanek, nezíská přístup k citlivým datům, které jsou uloženy na serveru aplikace. Tento problém byl vyřešen kombinací spouštění nedůvěryhodného kódu v odděleném rámci na serveru s využitím pokročilých technik jazyku Javascript.

Uživatelské prostředí bylo vytvořeno kombinací frameworku Bootstrap a Angular.js. Účelem bylo vytvořit velmi jednoduchý návrh, přesto s pokročilými možnostmi jednostránkové aplikace, který by případného uživatele neobtěžoval při práci. Tento bod byl podpořen ve výsledcích testování použitelnosti.

Výsledná aplikace splňuje předem specifikované požadavky. V případě pokračování v práci na aplikaci je zde několik rozšíření, některé byly zmíněny při testování použitelnosti. Jde o rozšíření funkčnosti profilu, účastníkům připadal příliš strohý. Co se týče funkčnosti aplikace, tak by bylo vhodné vytvořit administrační sekci, pomocí které by bylo možné spravovat uživatelské účty nebo, a to je důležitější, spravovat úkoly v jednotlivých kategoriích, případně nahrávat další, nové úkoly. Nyní to funguje tak, že se všechny úkoly nahrají do databáze při prvním spuštění aplikace. V neposlední řadě by bylo také na zvážení další, pokročilejší, bezpečností opatření pro spouštění nedůvěryhodného kódu.

Literatura

[1] Bartosz Krajka: *The difference between Virtual DOM and DOM*. 2015, [Online; navštíveno 12.5.2016].
URL http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/

[2] Douglas Crockford: *Code Conventions for the JavaScript Programming Language* . [Online; navštíveno 12.5.2016].
URL http://javascript.crockford.com/code.html

[3] Google: *Data Binding*. 2016, [Online; navštíveno 12.5.2016].
URL https://docs.angularjs.org/img/Two_Way_Data_Binding.png

[4] Google, Hyperakt, Vizzuality: *The Evolution of the Web*. 2012, [Online; navštíveno 12.5.2016].
URL http://www.evolutionoftheweb.com/#/evolution/day

[5] Hage Yaapa: *The MongoDB Tutorial*. 2011, [Online; navštíveno 12.5.2016].
URL http://www.hacksparrow.com/the-mongodb-tutorial.html

[6] Indeed: *react.js, backbone.js, angular.js, ember.js Job Trends*. 2016, [Online; navštíveno 12.5.2016].
URL http://www.indeed.com/jobtrends/react.js%2Cbackbone.js%2Cangular.js%2Cember.js.html

[7] Jakub Mrozek: *Začínáme s AngularJS*. 2012, [Online; navštíveno 12.5.2016].
URL https://www.zdrojak.cz/clanky/zaciname-s-angularjs/

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

[9] Mike Wasson: *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET*. 2013, [Online; navštíveno 12.5.2016].
URL https://msdn.microsoft.com/dynimg/IC690875.png

[10] Mozilla Developer Network and individual contributors: *Functions and classes available to Web Workers*. 2016, [Online; navštíveno 12.5.2016].
URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Functions_and_classes_available_to_workers

[11] NIELSEN NORMAN GROUP: *Turn User Goals into Task Scenarios for Usability Testing*. 2014, [Online; navštíveno 12.5.2016].
URL https://www.nngroup.com/articles/task-scenarios-usability-testing/

[12] Node.js Foundation: *Node.js v6.1.0 Documentation*. [Online; navštíveno 12.5.2016].
URL https://nodejs.org/api/vm.html

[13] Node.js Foundation: *Basic routing*. 2016, [Online; navštíveno 12.5.2016].
URL http://expressjs.com/en/starter/basic-routing.html

[14] Node.js Foundation: *Using middleware*. 2016, [Online; navštíveno 12.5.2016].
URL http://expressjs.com/uz/guide/using-middleware.html

[15] Osmani, A.: *Developing Backbone.js Applications* . O'Reilly Media, 2013, ISBN 1449328253.

[16] The OWASP Foundation: *Cross-site Scripting (XSS)*. 2016, [Online; navštíveno 12.5.2016].
URL https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[17] TILDE INC.: *Core Concepts*. 2016, [Online; navštíveno 12.5.2016].
URL https://guides.emberjs.com/v2.5.0/getting-started/core-concepts/

[18] Zakas, N. Z.: *JavaScript pro webové vývojáře* . Computer Press, 2009, ISBN 978-80-251-2509-0.

Přílohy

Seznam příloh

Příloha A

API

Cesta	Příchozí data	Odchozí data
GET /api/users/single/:id	Nic	User
GET /api/users/single/:username	Nic	User
GET /api/users/all	Nic	[User]
GET /api/tasks/single	Nic	Task
GET /api/tasks/all	Nic	[Task]

POST /api/users/games/update	{
	_id: Number,
	time: String,
	stars: Number
	}
		Chyba: {
		success: false,
		error: String
		}
		Správně: {
		success: true
		}

POST /api/users/profiles/update	{
	_id: Number,
	gender: String,
	info: String,
	birthday: String
	}
		Chyba: {
		success: false,
		error: String
		}
		Správně: {
		success: true}

POST /api/users/login	{
	username: String,
	password: String
	}
		Chyba: {
		success: false,
		error: String
		}
		Správně: {
		success: true,
		user: User._id
		}

POST /api/users/signup	{
	username: String,
	password: String,
	gender: String,
	info: String,
	birthday: String
	}
		Chyba: {
		success: false,
		error: String
		}
		Správně: {
		success: true,
		user: User._id
		}

GET /api/users/logout	Nic	Nic

POST /api/games/run_code	{
	code: String
	}
		{
		output: String
		}

Model User:

```
{
   username: String,
   password: String,
   email: String,
   gender: String,
   birthday: String,
   info: String,
   games: [{
      _id : { type: Number, ref:
'Task' },
      time: String,
      completed: Boolean,
      stars: Number
   }]
}
```

Model Task:

```
{
_id: Number,
name: String,
chapter: String,
code: String,
result: {console: [String], result: String},
category: String,
times: {
   time1: String,
   time2: String
},
evaluate: Boolean,
next: String,
help: String
}
```

Obrázek A.1: API JSON data.