VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Absolvování individuální odborné praxe ve firmě

Individual Professional Practice in the Company

2016

Tomáš Putna

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Zadání bakalářské práce

Student: **Tomáš Putna**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R059 Mobilní technologie

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě: Railsformers s.r.o.
2. Struktura závěrečné zprávy:
a. Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta
b. Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti
c. Zvolený postup řešení zadaných úkolů
d. Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe
e. Znalosti či dovednosti scházející studentovi v průběhu odborné praxe
f. Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vedl odbornou praxi studenta

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Zdeňka Chmelíková, Ph.D.**

Konzultant bakalářské práce: Ing. Lukáš Kamp

Datum zadání: 01.09.2015
Datum odevzdání: 29.04.2016

VYSOKÁ ŠKOLA BÁŇSKÁ - TECHNICKÁ UNIVERZITA OSTRAVA
Fakulta elektrotechniky a informatiky
OSTRAVA

doc. Ing. Miroslav Vozňák, Ph.D.
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

Railsformers s.r.o.
IČ: 24704440 DIČ: CZ24704440
www.railsformers.com
info@railsformers.com

V Ostravě 13. dubna 2016 .....

Rád bych poděkoval mé vedoucí bakalářské práce Ing. Zdeňce Chmelíkové, Ph.D. za odbornou pomoc a konzultaci při tvorbě této bakalářské práce.

Poděkování také patří Ing. Jiřímu Kubicovi, jednateli Railsformers, s. r. o. za možnost absolvovat odbornou praxi a Ing. Lukáši Kampovi, konzultantovi mé bakalářské práce a celému kolektivu zaměstnanců, kteří ochotně poradili při řešení problémů.

Chtěl bych poděkovat také svým rodičům a celé rodině za podporu při studiu.

Abstrakt

Bakalářská práce se zabývá odbornou praxí ve firmě Railsformers, s. r. o. která sídlí v Ostravě. Firma se zabývá tvorbou a implementací webových služeb. Cílem této práce je popis harmonogramu a pracovní pozice, na které jsem pracoval. Zmiňuji se i o firmě jako celku, popisuji moje úkoly týkající se webových aplikací a projektů, při kterých jsem použil technologie, které běží na frameworku Ruby on Rails. Vysvětluje řešené úkoly, které mi byly zadané. Na závěr popisuji a hodnotím průběh celé mé praxe a znalosti, které jsem získal a naopak, které mi chyběly.

Klíčová slova: odborná praxe, Ruby on Rails, Railsformers, s. r. o., webové aplikace, webové služby, MVC, HTML 5

Abstract

This bachelor thesis is about my bachelor practice in the company Railsformers s.r.o, which is residing in Ostrava. Their solutions are creation and implementation of web services. The goal of this bachelor thesis is to describe my working schedule and the job position where I was working. I will include informations about the company as the whole. I will describe my tasks regarding the web applications and the projects, where I used technologies which will work in Ruby on Rails framework. The bachelor thesis is explaining tasks, which was assigned to me. I will describe and assess the progress of my whole bachelor practice and knowledges which I have got and on the other hand which I have received in the conclusion of this bachelor thesis.

Key Words: professional practice, Ruby on Rails, Railsformers, s.r.o., web applications, web services, MVC, HTML 5

Obsah

Seznam použitých zkratek a symbolů

AJAX	–	Asynchronous JavaScript and XML
CMD	–	Command Prompt
CSS	–	Cascading Style Sheets
DRY	–	Don't repeat yourself
HAML	–	HTML Abstraction Markup Language
HTML	–	Hyper Text Markup Language
MVC	–	Model View Controller
MySQL	–	My Structured Query Language
OS	–	Operating system
POP3	–	Post Office Protocol
RVM	–	Ruby Version Manager
SASS	–	Syntactically Awesome Stylesheets
SEO	–	Search Engine Optimization
SSL	–	Secure Sockets Layer
URL	–	Uniform Resource Locator
VCS	–	Version Control System
WWW	–	World Wide Web
XHTML	–	Extensible Hyper Text Markup Language

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Cílem této bakalářské práce je přiblížit a seznámit s průběhem mého působení ve firmě Railsformers, s.r.o. a nastínit mou činnost a práci ve firmě. Hlavní myšlenkou pro absolvování praxe byla nová zkušenost s neznámým programovacím jazykem a uplatnění mých znalostí z informačních a internetových technologií.

Tuto bakalářskou práci jsem si zvolil z důvodu získávání nových zkušeností v reálném pracovním prostředí. Odbornou praxi jsem vykonával ve firmě Railsformers, s.r.o., při které jsem dostával úkoly týkající se vývoje webových aplikací, konkrétně aplikací na způsob inzertní webové služby a importování informací z bank. Popisuji celkový průběh praxe, která se zabývá tvorbou webových aplikací v jazyce Ruby on Rails s různými knihovnami, které zajišťují správný běh aplikace.

Bakalářská práce je rozdělená do kapitol, u kterých popisuji základní informace o firmě Railsformers, s.r.o s jejími produkty. Dále popisuji mé pracovní zařazení, časovou náročnost zadaných úkolů a seznam použitých technologií jakou jsou Git, Ruby on Rails, Bootstrap a další. Řešení zadaných úkolů je kapitola, která se zaměřuje na seznam úkolů u projektu inzertní webové služby a importů z bank s jejími popisy, zadáními, překlady a testováním. Na závěr popisuji znalosti a zkušenosti, které jsem získal a naopak zkušenosti a znalosti, které mi scházely s celkovým hodnocením bakalářské praxe.

2 Seznámení s firmou Railsformers, s. r. o.

Firma Railsformers, s.r.o. [1] byla založena v roce 2010 Ing. Jiřím Kubicou a zabývá se vývojem internetových a intranetových systémů. Sídlí ve Vědecko-technologickém parku v Ostravě a spolupracuje s VŠB-TU Ostrava [2]. Vytváří webové aplikace a informační systémy na profesionální úrovni již několik let. Zaměřují a specializují se na větší projekty, komunitní portály, sociální sítě, řídící a jiné komplexní systémy, mimo jiné také poskytují a zajišťují SEO [3] optimalizace pro vyhledávače a implementační analýzy webových aplikací. Využívají k tomu framework Ruby on Rails [4] a týmové spolupráce. Řídí se moderními trendy v oblasti webdesingu a použitelnosti.

RAILSFORMERS

Obrázek 1: Logo firmy Railsformers, s.r.o.

2.1 Hlavní produkty firmy

Mezi hlavní portfolio patří portál pro širokou veřejnost **sRecepty.cz** [5], což je portál o vaření s recepty, ingrediencemi a dalšími informacemi ohledně světové i domácí kuchyně. Na webu lze nalézt spoustu rad, tipů a nově ke stažení mobilní aplikaci.

Další systém, o který se firma stará a vyvíjí, je **sMoneybox** [6], jedná se o správu osobních financí s několika jazykovými mutacemi. Je to chytrý a elegantní nástroj pro vedení útrat v takzvané online peněžence, tento systém se svou složitostí je nabízen zcela zdarma.

Go4Translate [7], jak už název napovídá, je online nástroj, který slouží k překladu různých aplikací snadno a rychle. Podporuje všechny nejběžnější formáty, jako jsou CSV, JSON, YAML a další. Možno překládat do mnoha jazykových mutací.

Jako poslední hlavní produkt je **Hedurio** [8], slouží pro řízení bezpečnostních agentur a hlídání budov, to znamená, že třetí strana okamžitě vidí aktuální okamžitou událost. Jedná se o unikátní online informační systém svého druhu. Koncovým uživatelům je poskytován na bázi cloudového řešení.

2.2 Pracovní zařazení

Ve firmě jsem byl na pozici externího vývojáře webových aplikací a jako stážista a programátor. Musel jsem analyzovat a ladit daný problém, který mi byl zadán. Mým úkole bylo navrhnout a řešit zadaný problém, zvolit správnou technologii k implementaci ve frameworku Ruby on Rails, který po sléze konzultovat s mými kolegy.

3 Časová náročnost a seznam úkolů

Ještě před samotným prvním projektem jsem se zabýval tvorbou zkušebních aplikací a zaučoval jsem se pracovat ve firemním prostředí. Nultý projekt, na kterém jsem dělal, byl zadán senior programátory, jednalo se o seznam tutoriálů, který seznamoval strukturu a funkce Ruby on Rails. Obsahem těchto návodů byla i samotná instalace frameworku a zároveň i zkušební aplikace, která ověřovala, jestli vše funguje, jak má.

Zadaný první větší projekt zabral zhruba polovinu mé pracovní náplně a doby, jednalo se o projekt Inzertní webové služby, na kterém se podílelo více vývojářů, a i když projekt byl rozsáhlí, tak ve finále díky týmové spolupráci s kolegy byl stihnut za poloviční dobu.

Poslední úkol, na kterém jsem dělal, až do konce mé praxe byl systém na import informací z bank. U této aplikace jsem se zdokonalil a osvojil základní principy v Ruby on Rails a naučil spoustu nových informací v odvětví internetového bankovnictví.

Měsíční časový průběh praxe:

Září, říjen, listopad, prosince: zaučení a práce na projektu inzertní webové službě

Leden, únor, březen, duben: práce na bankovních importech

4 Použité technologie, architektury a služby

4.1 Ruby on Rails

Ruby on Rails [9] je open source framework, což je sada knihoven, které poskytují obecnou funkci, mohou být použity ve více projektech a slouží k vytvoření multifunkční webové aplikace. Je založen na jazyku ruby a pracuje jako server a databáze. Rails je založen na modelu MVC a snaží se, aby tvůrci aplikace se nemuseli starat o mnoho detailů např. kontrola parametrů, koncepty pro autentizaci a autorizaci.

Programovací jazyk Ruby je objektově orientovaný jako Java, C++ a další, nemusí vytvářet pouze webové aplikace, ale i výpočetní sofistikované programy. Podle celosvětového měřítka se Ruby nachází v první desítce hned za PHP a JavaScriptem.

Filozofie Rails využívá jeden z principů vývoje softwaru zaměřený na snižování počtu duplicitních informací z anglického výrazu Don't repeat yourself (DRY, tj. „Neopakuj se“) [10]. Definuje přístup, při kterém jsou data strukturována tak, že se v aplikaci nacházejí pouze jednou. Tento princip zachovává vývojářům čitelnější a zpracovatelný kód a obsahuje méně bugů.

4.2 Zabezpečení v Ruby on Rails

Mezi základní bezpečnost, kterou framework zajišťuje a stará se, jsou relace mezi uživatelem a klientem. WWW server pošle klientovy identifikátor relace pomocí cookie, které jsou uloženy na počítači uživatele, a při každé návštěvě téhož serveru pak prohlížeč posílá zpět na server. Identifikátor relace je MD5 hash z řetězce, který tvoří aktuální čas, náhodné číslo v intervalu (0,1), číslo procesu interpretu jazyka Ruby a konstantní řetězec specifický pro každou aplikaci. Pravděpodobnost útoku a prolomení je malá a díky začlenění aktuálního času, tak je malá i kolize identifikátorů [11].

4.3 Instalace

Ruby on Rails je multiplatformní a může být nainstalovaný na operační systémy Microsoft Windows, Linux, Mac OS X. Doporučované prostředí pro vývoj je Linux a Mac OS X, tedy systémy na bázi Unixu.

Jako první jsem provádět instalaci Ruby a Rails na OS Windows 10 pomocí instalátoru, instalace proběhla v pořádku, ale problém byl v nasazování verzí samotného Ruby a Rails. Instalace obsahovala balíčky RubyGems a funkční databázi SQLite3.

Poté jsem přešel na Linux a jeho distribuci Ubuntu, je to rychlý systém se snadnou instalací. Instalaci jazyka Ruby a frameworku Ruby on Rails jsem prováděl pomocí RVM (Ruby Version Manager), vyvíjeno od roku 2007 [12].

```
curl -ssl https://get.rvm.io | bash -s stable --ruby
curl -ssl https://get.rvm.io | bash -s stable --rails
```

Výpis 1: Posloupnost příkazů pro instalaci Ruby on Rails s RVM

4.4 RubyGems

Jsou balíčky (knihovny) pro Ruby on Rails a ulehčují práci při implementaci složitých řešení. Zabezpečují rozšiřitelnost aplikace a jsou snadnější na implementaci. Mezi základní gemy, které se používají, patří například gem **Simple form** pro zpřehlednění formulářů nebo gem **devise** pro autorizaci uživatelů.

Seznam všech využívaných gemů v Ruby on Rails, které jsou použity v Ruby on Rails aplikaci se nacházejí souboru Gemfile, který se nachází v kořenovém adresáři [13].

```
gem install rubygems-update
Bundle install
```

Výpis 2: Instalace RubyGems a samotných Gemů

4.5 Git

Git je **open source** distribuovaný systém správy verzí, zaznamenává změny souboru nebo sady souborů v průběhu času, a uživatel tak může kdykoli obnovit jeho/jejich konkrétní verzi (tzv. verzování). VCS dokáže vrátit jednotlivé soubory nebo celý projekt do původního nebo předchozího stavu, porovnává změny provedené v průběhu času, dokáže zjistit, kdo naposledy upravil něco, co nyní způsobuje problémy, kdo vložil jakou verzi a kdy. Zároveň s verzováním dochází k zálohování souborů. Mezi základní vlastnosti patří rychlost, jednoduchý design, plná distribuovatelnost a objem dat [14].

```
apt-get install git
git config --global user.name "Tomas"
git clone dev.whatever.org:/git/example.rb
git add --all; $ git commit -m 'my first commit';
git fetch; git pull; git push master
```

Výpis 3: Instalace, konfigurace a základní používání gitu

4.6 MVC

Ruby on Rails obsahují ve svém jádru architekturu **Model**, **View**, **Controller**, obvykle označovanou jako **MVC**. Základní vlastnosti modelu jsou znovupoužitelnost kódu, přehledná struktura usnadňující údržbu a oddělení aplikační logiky od uživatelského rozhraní [15].

4.6.1 Modely (models)

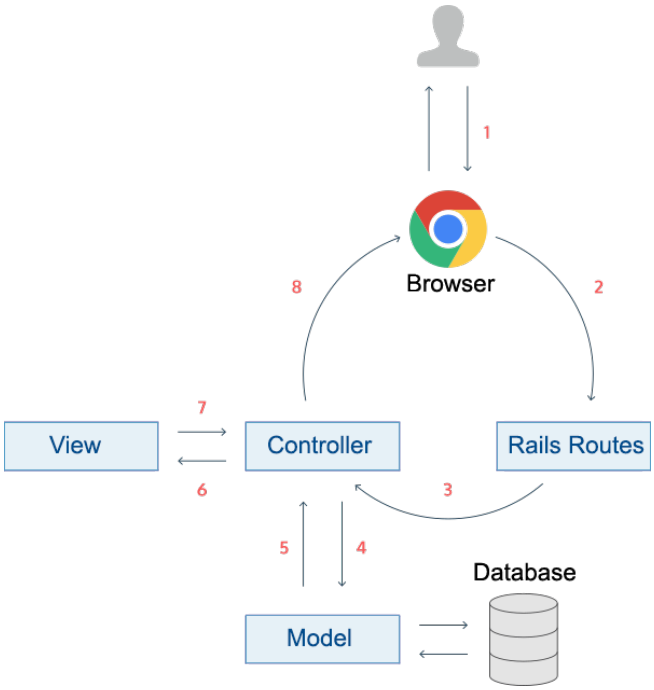
Model reprezentuje informace (data), jedna tabulka v databázi ve většině případů obsahuje jeden model a je zodpovědný za udržování stavu aplikace. Někdy je tento stav přechodný a trvá jen pár interakcí s uživatelem. Jindy je trvalý a ukládá se mimo aplikaci, často do databáze. Propojuje databázi s kontrolerem.

4.6.2 Pohledy (views)

V pohledu se řeší front-end a uživatelské rozhraní, které je obvykle založené na datech z modelu. V Ruby on Rails jsou pohledy převážně HTML soubory s prvky Ruby kódu, který předává do šablony vytvořené data. Pohledy vrací požadavky kontroleru od uživatele a zároveň komunikují a poskytují data webovému prohlížeči nebo jinému nástroji, který posílá požadavky.

4.6.3 Kontrolery (controllers)

Kontrolery dávají dohromady modely a pohledy. V Rails slouží kontrolery k zpracování požadavků které přichází z webového prohlížeče (obvykle vstup uživatele), pracují s modelem a zobrazují příslušný pohled uživatele.



Obrázek 2: Základní model MVC [20]

Uživatel otevře prohlížeč a zadá URL adresu. Rails aplikace se podívá do routovací tabulky a určí, kam se má dotaz přesměrovat na jaký kontroler. Kontroler se zeptá modelu, jestli se mají vytáhnout nějaká data z databáze a pošle odpověď zpět. Kontroler poté pošle do pohledu získána data a vykreslí uživateli HTML stránku.

4.7 Active Record

Active Record tvoří tzv. jádro frameworku a vystihuje konvenci před konfigurací. Active Record je návrhový vzor, který mapuje databázové tabulky na třídy a převádí řádky na objekty (tedy instance tříd) a sloupce na atributy objektů. Z názvu třídy **Advertisement** odhadne název tabulky **advertisements**. Oddaluje vývojáře od základního SQL, a proto místo SQL výběru můžeme použít jednoduší variantu z Rails, která se nachází na vyšší úrovni [18].

```
SELECT * FROM advertisements WHERE id = 1
advertisements.find(1)
```

Výpis 4: Základní dotazy z SQL a Ruby on Rails

4.8 Bootstrap

Bootstrap je sada nástrojů pro tvorbu front-end webových aplikací. Obsahuje šablony založené na HTML a CSS. V dnešní době splňuje bootstrap mnoho aspektů moderního a poutavého vzhledu. Ruby on Rails může implementovat gem bootstrap-sass a s potřebným nastavením lze jednoduše a rychle aplikovat do aplikace [19].

5 Řešení zadaných úkolů

5.1 Inzertní webová aplikace

Cílem bylo vytvořit webovou aplikaci na principu inzerce. Uživatel se mohl registrovat, vkládat inzeráty přidávat k ním popisky, obrázky a základní informace o produktu. Později možnost přidávání komentářů a odpovědí na inzerát.

Scénář uživatele, který není registrován, může pouze prohlížet a vyhledávat inzeráty. Po zaregistrování dojde k přesměrování na úvodní stránku, kde vidí seznam kategorií a výpis všech nabízených inzerátů, které jsou stránkované. Uživatel může upravovat svůj osobní profil, dané inzerce objednávat a nabízet.

5.1.1 Zadání projektu

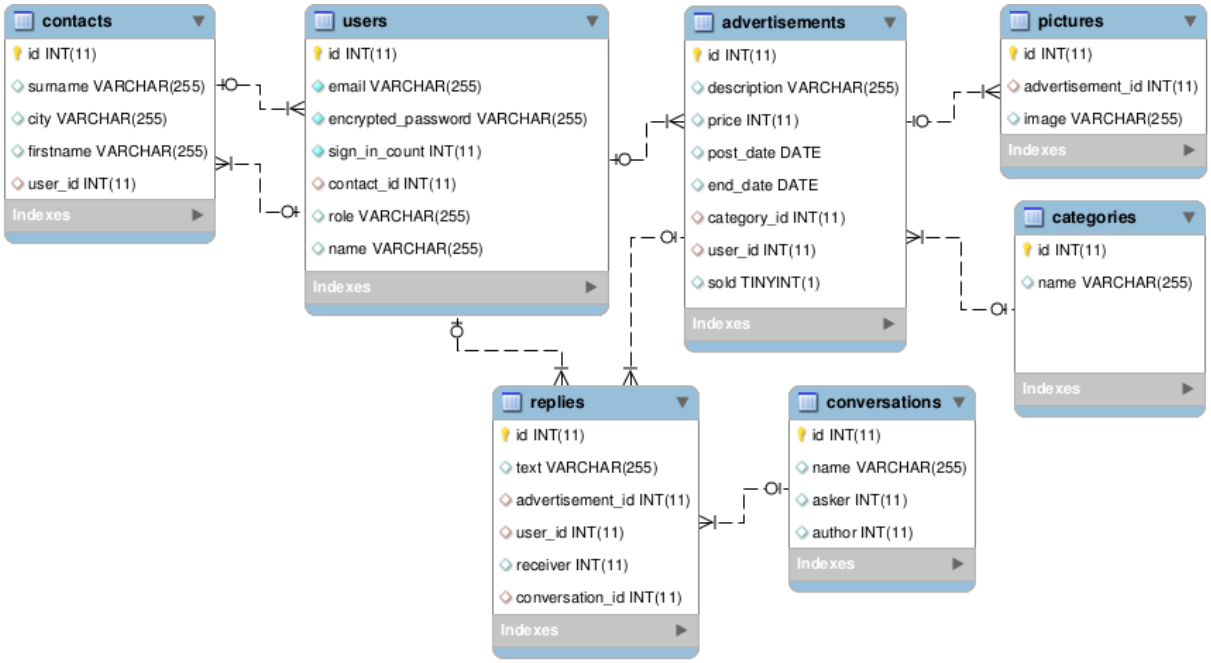
Zadáním bylo vytvořit inzertní webovou službu, kterou jsme dostali od našeho vedoucího. Projekt měl následující požadavky: využívání gitu pro správu kódu, zabezpečený přístup registrace a správy uživatele s rolemi, k inzerátům přidat popis, cenu, do kdy platí, do jaké kategorie spadá, možnost přidat a ořezávat fotky, vytvořit homepage pro výpis posledních vložených inzerátů se sociálními sítěmi, využití technologie bootstrap, tvorba testů, překladů a vyhledávání.

5.1.2 Návrh, postup a popis tvorby

Téměř všechny Ruby on Rails aplikace komunikují s databázi. Nastavení, jakou databázi aplikace využije, se dá nastavit v konfiguračním souboru config/database.yml. V základu každé aplikace je nastavena defaultní databáze SQLite, která je nainstalována společně s framevorkem Rails, pro vývoj inzertní webové služby jsem použil databázi MySQL [16], jedná se o multiplatformní relační databázi s jazykem SQL. Podporuje celou řadu programovacích jazyků např. `C`, `C++`, `C#`, `Java`, `Ruby on Rails` a další. V dnešní době převážně využívána na webové aplikace. Kvůli jednoduší implementaci a využití jsem zvolil právě MySQL databázi. Každá Rails aplikace z pohledu databáze je nastavena pro tři různá prostředí, v nichž může být Rails aplikace spuštěna.

- Development- vývojové prostředí, lokální databáze na počítači
- Test- testovací prostředí pro automatizované testy
- Production- produkční prostředí je používáno na serveru

Jako první krok, který jsem musel udělat, byla analýza zadaného projektu a následné vytvoření databáze na které je postavena celá aplikace. Poté jsme si s kolegy rozdělili práci a pracovali tak, aby práce byla rychlá a efektivní, a pracovní postupy, na kterých jsme dělaly, se minimálně opakovali. Pro rychlé pochopení problému popisuji tabulku **advertisements** a **categories**, pro ostatní tabulky byl vývoj podobný.



Obrázek 3: EER diagram inzertní webové služby

EER diagram reprezentuje návrh databáze, kde tabulka **advertisements** uchovává informace o inzerátech. Tabulka **pictures** a **categories** jsou propojené s tabulkou **advertisements** a uchovávají data s obrázky a kategoriemi. Pomocí gemu devise byla vytvořena tabulka **users**, která navazuje spojení s inzeráty a kontakty, které uchovávají informace o kontaktu uživatele. Odpovědi a konverzace mezi uživateli řeší tabulky **replies** a **conversations**.

Po návrhu databáze a konzultaci s mými kolegy jsem se pustil do tvorby samotné aplikace pomocí scaffoldingu jsem vygeneroval hlavní části programu, příkazem ***rails generate scaffold Advertisement description:string price:integer*** jsem vytvořil jednu z funkčních části zdrojového kódu ze šablon, které Rails obsahuje. Generátor vygeneruje funkční MVC s databázovou migraci, základními konfiguračními soubory a testy. Tento proces jsem aplikoval na další části programu, které byly potřeba vygenerovat tj. tvorba další modelů a tabulek.

5.1.3 Migrace databáze

Jedna z operací, která se provede při scaffoldingu je databázová migrace. Migrace jsou třídy, které slouží k vytváření, úpravě, editaci a mazání tabulek z databáze. Samotná migrace umožňuje také naplňovat tabulky daty. Příkazem ***rake db:drop*** se smaže celá databáze a příkazem ***rake db:create*** se vytvoří databáze podle tabulek, které jsou definované v databázovém souboru **schema.rb** v Rails aplikaci. Pro aplikaci databázových tabulek, je zapotřebí zavolat migraci příkazem ***rake db:migrate***.

Velkou výhodu migrace je, že při změně atributu nebo jména tabulky nemusíme mazat celou tabulku a tak nepřijdeme o naše data, ale stačí pouze změnit to, co potřebujeme. Ruby on Rails migrace umožnuje jednoduchý přechod na jiný typ databáze.

Pro jakoukoli změnu v databázi (smazání tabulky, změny atributu) se používá implementovaný generátor pro migrace. Příkazem ***rails generate migrate RemoveNicknameFromUsers*** se vygeneruje soubor do složky **db/migrate** ve kterém se můžou implementovat změny, které chceme pro danou databázi provést.

```
class RemoveNicknameFromUsers < ActiveRecord::Migration
  def change
    remove_column :users, :nickname, :string
  end
end
```

Výpis 5: Vytvoření migrace s odstraněním nickname

Z této vytvořené migrace se z tabulky **User** pomocí metody **change** a **remove_column** smaže atribut **nickname**, který má hodnotu **string**.

```
class CreateAdvertisements < ActiveRecord::Migration
  def change
    create_table :advertisements do |t|
      t.string :description
      t.integer :price
      t.timestamps
    end
  end
end
```

Výpis 6: Migrace k vytvoření tabulky Advertisement s atributy descritption a price

Tato migrace vytváří metodu s názvem **change** volanou když se spustí migrace. Rails aplikace vědí a pamatují si, kdy se jaká akce volá a je možné ji později vrátit. Jakékoli úpravy lze vzít zpět. Při spuštěný migrace se vytvoří dva sloupce jeden formátu **string** a druhý formátu **integer**. Vytvoří se také dva sloupce časové značky, které udržují informaci o času, vytvoření a úpravě.

5.1.4 Přihlášeni a registrace uživatele

Přihlášení a registrace uživatele je jedna ze základních vlastností aplikace a bez této funkcionality se neobejde téměř žádný web. Pro tuto funkci jsem využil gem **devise**, který je postaven na flexibilní přizpůsobení Rails aplikaci a automaticky se implementuje to MVC modelu. Gem řeší práci s databází a využívá šifrování pro zabezpečení hesel, dále pak řeší správu a editaci vytvořených účtů tím, že vygeneruje provizorní pohledy, které se můžou dále upravovat k dokonalosti.

Aby Rails aplikace věděla, že se má použít **devise**, musel jsem přidat do souboru **Gemfile** gem **devise** a nainstalovat dvěma příkazy ***bundle install*** a ***rails generate devise:install***.

```
source 'https://rubygems.org'
gem 'devise'
...
```

Výpis 7: Vložený gem v souboru Gemfile

Pro správné zobrazování notifikací například pro upozornění správného přihlášení uživatele nebo jestli došlo k nějaké chybě, tak se musí přidat do souboru **views/layouts/applications.html.erb** následující kód.

```
<p class="notice"><%= notice %></p>
<p class="alert"><%= alert %></p>
```

Výpis 8: Zobrazení notifikací v devise

Nyní je provedeno základní nastavení a pomocí příkazu ***rails g devise User*** vygeneruji základní migraci a strukturu aplikace pro tvorbu uživatelů. Zároveň se vytvoří i routování, které se nachází v **routes.rb**. Následně jsem vytvořil migraci, pomocí které jsem přidal cizí klíč **user_id** do tabulky **advertisements** a v modelu jsem provedl validaci. Model **advertisements** má vazbu s uživatelem belongs_to a uživatel s **advertisements** has_many. To znamená, uživatel může mít mnoho inzerátů a inzerát jednoho uživatele.

Ostatní nastavení gemu se nachází ve složce **config/initializers/devise.rb**, kde se dá nastavit spousta drobností od délky hesla po různé mailovací konfigurace, potvrzovací služby a další. Pro správné využívání gemu je také zapotřebí nastavit metody, který gem využívá. Metoda ***before_action :authenticate_user!*** zabezpečí, potřebné přihlášení každého uživatele v kontroleru, bez přihlášení a vytvoření nové sessions není možnost zobrazit jakoukoli stránku.

```
<% if user_signed_in? %>
<%= link_to 'Option',edit_user_registration_path(current_user)%>
<%= link_to 'LogOut',destroy_user_session_path,:method => :delete%>
<% else %>
<%= link_to 'LogIn', new_user_session_path %>
<%= link_to 'SignUp', new_user_registration_path %>
<% end %>
...
```

Výpis 9: Využití gemu devise v pohledu application.html.erb

Zde můžeme vidět sílu gemu v podmínce je, když uživatel je přihlášený, tak zobraz linky na nastavení a odhlášení, jestliže není, tak zobrazí linky na přihlášení nebo registraci, Jsou zde i cesty na jednotlivé pohledy, které jsou umístěny ve **views/devise/** a v tomto adresáři jsou uloženy veškeré formuláře pro registraci, editaci, zrušení účtu a přihlášení.

```
= simple_form_for(resource, as:resource_name, url:session_path (resource_name))
    do |f|
= f.input :email, autofocus: true
= f.input :password, autocomplete: "off", label: t('Password')
- if devise_mapping.rememberable?
= f.input :remember_me, :as => :boolean, input_html: {checked:false}, label: t(
   'RememberMe')
= f.submit t('LogIn')
...
```

Výpis 10: Ukázka přihlašovacího pohledu vygenerovaný devisem

Devise nám vygeneroval šablonu pro přihlašování a je jenom na nás, jak si ji nastylizujeme podle zvoleného bootstrapu. Ve výsledku se nám vygeneruje formulář, jelikož je to formulář na přihlášení, tak bude obsahovat **textbox** pro Email s automatickým označením pro psaní (autofocus) a **textbox** pro zadání hesla, dále pak **checkbox** pro zapamatování hesla a tlačítko pro přihlášení.

Abychom mohli přidat k danému inzerátu určitého uživatele je zapotřebí nastavit kontroler tak, aby zobrazoval stávajícího přihlášeného uživatele. V devise existuje metoda **current_user**, která značí, který uživatel a která **session** je právě aktivní.

```
if !params[:user_email].blank?
@user.user_email = params[:user_email]
end
@category = Category.find(params[:category_id])
@advertisement=@category.advertisements.create(advertisement_params)
@advertisement.user_id = current_user.id if user_signed_in?
if @advertisement.save
respond_with(@category)
@user = User.find_by(:id => @advertisement.user_id)
...
```

Výpis 11: Ukázka kontroleru s akcí create a implementovaným uživatelem

Při vytvoření inzerátu se ošetřuje správnost emailu, dále se do proměnné **category** ukládá zvolená kategorie, pod kterou bude uložen inzerát. Proměnná **advertisemnet** vytvoří inzerát podle kategorie a přiřadí se aktuální uživatel. Jestliže je inzerát správně uložený, tak přesměruj stránku na kategorie a do **user** najdi právě to **id** podle inzerátu a user_id.

5.1.5 Správa aplikačních rolí

Pro vytvoření a správu rolí v aplikaci jsem použil gem **CanCan**, což je knihovna, která slouží k definování práv uživatele a admina. Všechna oprávnění se definují na jednom místě, tím vzniká přehledná a unikátní správa rolí, které se neopakují mezi modely a řadiči [17].

Vložením gemu do souboru Gemfile a následném nainstalováním jsem vytvořil role pro admina a uživatele. V modelu **ability.rb** a ve třídě **Ability** jsem nastavil globální role pro admina (administrátor webu) a usera (uživatel využívající služby).

```
class Ability
  def initialize(user)
    ...
    can :read, :all if user.role == "user"
    can :manage, :all if user.role == "admin"
    ...
  end
end
```

Výpis 12: Nastavení rolí v souboru ability.rb

Výpis nám říká a gem CanCan zajišťuje, když uživatel je v roli user, tak může pouze číst data a nemůže je řídit, ale když se role otočí a uživatel je v roli admin, tak může provádět veškeré operace v aplikaci.

Uživatel, využívající inzertní služby, může pomocí rolí vytvářet nové inzeráty, upravovat, mazat nebo vést zprávové konverzace s ostatními uživateli. Při registraci nového uživatele se mu automaticky přiřadí role user, který je definován v modelu. Pokud chceme vytvořit nového uživatele s rolí admin, tak se to musí provést přes změnu ve zdrojovém kódu nebo ručním přidáním přes příkazový řádek. Administrátor může dělat vše co registrovaný uživatel, ale navíc může také spravovat inzeráty ostatních registrovaných uživatelů a má správu nad celým systémem.

```
class User < ActiveRecord::Base
...
  def set_default_role
    if self.role.blank?
      self.role = 'user'
    end
  end
end
```

Výpis 13: Vytvoření uživatele s rolí user v models/user.rb

Zdrojový kód nastaví jako defaultní roli user, při podmínce, když role je prázdná, což při registraci nového uživatele platí.

S gemem **CanCan** je úzce spjat gem **devise**, který s jednoduchými podmínky udává podle rolí přístup k funkcím na které má právo. Například pro zobrazení odkazu nebo obsahu na který uživatel nemá právo.

```
<% if user_signed_in? %>
  <li><%= link_to t('HomeLink'), root_path %></li>
  ...
  <% if current_user.role == 'admin' %>
    <li><%= link_to t('Category'), categories_admin_path %></li>
    ...
  <% end %>
<% end %>
```

Výpis 14: Použití rolí ve views/layouts/application.html.erb

Ukázka kontroluje dvě podmínky a to když je uživatel přihlášený, tak uvidí odkaz domů, ale když je uživatel přihlášení a má roli admin, tak navíc uvidí odkaz na kategorii. Když uživatel má defaultní roli user, tak uvidím pouze linky s rolí user nebo s žádnou.

5.1.6 Tvorba kontroleru inzerát

Při tvorbě kontroleru inzerát, bylo zapotřebí propojit funkcionalitu s kontrolerem kategorie, kterou jsem vygeneroval pomocí Scaffoldingu. Aby vše správně fungovalo podle zadání, tj. uživatel si vybere kategorii a pod tuto kategorii vloží inzerát, musel jsem nastavit v souboru **controllers/advertisements_controller.rb** propojení s kategorií.

```
def create
@category = Category.find(params[:category_id])
@advertisement = @category.advertisements.create(params)
if @advertisement.save
respond_with(@category)
else
respond_with(@category,:location => new_category_advertisement)
end
end
...
```

Výpis 15: Akce create z kontroleru advertisements

Do proměnné **category** se uloží kategorie podle zvolené id z databáze pomocí metody **find**. Proměnná **advertisement** slouží pro vytvoření inzerátu pod zvolenou kategorii a **params** slouží k ošetření správnosti uložení dat do databáze. V podmínce se ošetřuje správnost uložení inzerátu,

jestli inzerát není uložen, tk ho přesměruj na nový inzerát ale, jestli je přesměruj ho na přehled inzerátů.

5.1.7 Tvorba pohledu šablony

Jakmile byla hotová koncepce kontroleru, začal jsem vytvářet samotný vzhled stránky. V architektuře MVC je to právě pohled. Základní struktura vzhledu se provádí v souboru **views/layouts/aplication.html.erb**. Zde se nastavují hlavičky, patičky a samotný obsah celé aplikace. Pomocí klíčového slova **yield** se zobrazují právě zmíněné ostatní šablony vzniklé pomocí kontrolerů a pohledů. Je to uděláno proto, abychom do každé šablony nemuseli pořád dokola přidávat ten samý kód, který byl už jednou napsán.

Funkce **yield** vyrenderuje a vykreslí šablonu podle aktuálního **controller/action**, která se zjistí podle zadaného URL a nastavené routovací cestě. Funguje to tak, že **yield** zjistí volání z kontroleru například podle **advertisements/show**, tak se vyrendruje pohled, který je umístěn ve **views/advertisements/show.html.erb**. Veškeré pohledy se nachází ve složce **views/advertisements/**. Totéž bude fungovat i u dalších kontrolerů a pohledů.

Pro zjednodušení celé struktury pohledů existují v rails aplikacích parciální šablony které se značí podtržítkem před jejím jménem a v jiné šabloně se zavolají pomocí metody **render**, tyto parciální šablony musí být uložené ve stejném adresáři jako je soubor ze kterého je volána. Ukazují se zde základní principy DRY, tj. „Neopakuj se“.

```
= simple_form_for([@category, @advertisement]) do |f|
.field
= f.input :description, label: t('Description')
.field
= f.input :price, label: t('Price')
= f.submit t('SubmitButton')
...
```

Výpis 16: Ukázka parciální šablony z _form.html.haml

Využívají se zde dva gemy, jeden pro přehlednost formuláře **simple_form** a druhý **haml** pro jednoduší čitelnost kódu. Metoda **simple_form_for** vygeneruje formulář s parametry objektů a metoda **input** vygeneruje textbox pro vložení popisu a ceny.

Mezí základní faktory zdrojového kódu patří i čitelnost, Rails podporuje několik formátu pro zobrazování webových šablon mezi základní patří **html.erb** a **html.haml**. V tomto případě jsem použil HAML a od HTML se liší lepší čitelností, poskytuje flexibilitu, urychluje a zjednodušuje tvorbu šablon. Jsou to jedny ze základních značkovacích jazyků používané pro tvorbu webových stránek. Gem který jsem použil se jmenuje haml a velmi jednoduše se implementuje do prostředí, které se zaměřuje na čistotu, čitelnost a rychlost výroby [21].

Mezi základní faktory zdrojového kódu patří i čitelnost, Rails podporuje několik formátů pro zobrazování webových šablon, mezi základní patří html.erb a html.haml. V tomto případě

jsem použil HAML a od HTML se liší lepší čitelností, poskytuje flexibilitu, urychluje a zjednodušuje tvorbu šablon. Jsou to jedny ze základních značkovacích jazyků používané pro tvorbu webových stránek. Gem, který jsem použil, se jmenuje haml a velmi jednoduše se implementuje do prostředí, které se zaměřuje na čistotu, čitelnost a rychlost výroby [20].

```
%h1= t('NewAdvertisement')
= render 'form'
= link_to t('Edit'), category_advertisements_path(@category)
```

Výpis 17: Ukázka použíti vyrenderovaného formu

HAML a Rails vytvoří nadpis, nová inzerce a podle překladu vyrenderuje definovaný překlad. Metoda **render** zavolá parciální šablonu form, tedy formulář a vytvoří odkaz na editaci. Uživateli se pak zobrazí základní HTML s formulářem.

5.1.8 Modely a vazby mezi inzeráty a kategoriemi

Vztahy a validace mezi objekty nám zajišťuje jedna z vlastností Active Recordu. Pro správnou funkčnost celé aplikace je také za potřební správně navrhnout vazby, mezi tabulkami k tomu slouží v MVC architektuře modely. V modelech se provádí i samotné validace a ošetření vstupů. Mezi jednotlivými tabulkami existují vazby typu 1:M, 1:1 a M:N. Základní vazby které zapisuje Ruby on Rails jsou belongs_to (patří), has_many (mnoho), has_one a další. Active Record s použitím těchto vazeb v modelech bude považovat za propojené a přidá přístupové metody k propojeným objektům.

Celkově má projekt 5 modelů, které jsou mezi sebou propojené. Model **account** je propojený vazbami belongs_to :bank, belongs_to :user, has_many :transactions a has_many :attachments. Model **bank** je definován jako has_many :accounts oproti atachment modelu, který má vazbu belongs_to :account. **Profile** a **transaction** model mají vazby belongs_to :user a belongs_to :account. Poslední model user má vazby has_one :profile a has_many :accounts

```
class Advertisement < ActiveRecord::Base
  belongs_to :category
  belongs_to :user
  validates :description, presence: true, lenght: {maximum: 200}
  validates :price, numericality {only_integer: true}
end
```

Výpis 18: Ukázka modelu Advertisements

```
class Category < ActiveRecord::Base
  has_many :advertisement
  validates :name, presence: true
end
```

Výpis 19: Ukázka modelu Category

Model **advertisement** se dá přeložit, jako pro každý jeden inzerát patří jedna kategorie a jeden uživatel, Kdežto oproti modelu **category**, kde jedna kategorie může mít více inzerátů. V modelu se rovněž provádí validace a určuje, zda-li daný atribut musí obsahovat pouze číslice nebo holý řetězec stringů dále pak maximální délku a jestli musí nebo nemusí být prázdný.

5.1.9 Směrování

Scaffolding nám mimo jiné vygeneruje směrování pro inzeráty a pro správnou funkčnost přidá záznam do souboru **config/routes.rb**. Směrování musí být přidáno pro každý použitý model, aby Rails věděly, jaké URL mají použít pro další funkčnost.

```
resources :users do
  resources :advertisements
  ...
end
```

Výpis 20: Směrování pro Advertisements a Users

Tato deklarace vytvoří tzv. vnořenou entitu do entity **users**, která byla vytvořená pomocí gemu devise. Pravidla směrování tvoří hierarchii uživatelů a inzerátů to znamená, každý uživatel vždy může mít nějakou inzerci. **Resources** mapuje podle kontrolerů URL adresy a přijímá pomocí metody HTTP požadavky jako GET, POST, DELETE a další, podle kterých se právě zmíněný resources přizpůsobí.Pro zobrazení směrovací tabulky se používá příkaz ***rake routes***.

```
user_advertisements GET /users/:user_id/advertisements(.:format)
...
```

Výpis 21: Ukázka směrování po příkazu rake routes

Na výpisu můžeme vidět jak entita **advertisements** je vnořená do entity **users** s požadavkem GET a s následujícím formátem pro další použití.

5.1.10 Vyhledávání mezi inzeráty se stránkováním

Dalším úkolem, na kterém jsem dělal, bylo vytvořit vyhledávání a zároveň i stránkování pro lepší čitelnost a přehlednost. K vyhledávání jsem použil gem **Ransack** a pro stránkování gem **Kaminari**. První část, kterou jsem zajistil, byla instalace zmíněných gemů. Vložením do souboru

Gemfile jsem upozornil Rails, že chci nainstalovat tyto gemy a příkazem ***bundle install*** jsem nainstaloval gemy do aplikace.

Vyhledávácí formulář jsem vytvořil pomocí gemu **Ransack**, který jsem nastavoval v kontroleru **categories_controller.rb** protože inzeráty jsou vloženy pod určitou každou kategorii. Poté jsem musel do **views** konkrétně **index.html.erb** vložit formulář pro vyhledávání. Existuje zde spousta nastavení od různých seřazování až po různé jednotlivé vyhledávání. Pro tento projekt je nejvýhodnější hledání podle popisu a podle rozmezí dané ceny.

```
def index
  @q = Advertisement.search(params[:q])
  @advertisement = @q.result
end
```

Výpis 22: Instalace vyhledávání v kontroleru

V akci index je proměnná **q** do které se ukládají inzeráty podle parametru a metoda **search** zajišťuje právě to zmíněné hledání, výsledek hledání se ukládá do samotného inzerátu, aby bylo možné zobrazit vyhledané údaje.

```
= search_form_for @q, url: search_categories_path, html: { method: :post } do |
    f|
= f.text_field :description_cont
= f.text_field :price_gteq
= f.text_field :price_lteq
= f.submit t('Search')
...
```

Výpis 23: Vložení vyhledávání do pohledu

Seach form najde v kontroleru proměnou **q** a po úspěšném vyhledávání buď podle popisu, nebo v rozmezí podle určité ceny vyhodí výsledek a zobrazí ho na adresu která je v URL. Provádět se to bude metodou POST.

Větší čistotu a velké sofistikované přizpůsobení webové stránky nám zajišťuje gem **Kaminari**. Kdybych nepoužil tento gem, tak výsledná stránka s inzeráty by mohla být nekonečné dlouhá a tím pádem nevhodná pro další použití, ale s použitým gemem si můžu nastavit kolik produktů se mi zobrazí na stránku a pomocí stránkování můžu listovat mezi těmito produkty.

Podobně jako u vyhledávání je zapotřebí říct kontroleru aby použil stránkování. Je zapotřebí přidat do kontroleru propojení se stránkováním, tak aby zobrazoval určený počet inzerátu na stránku. Poté jednoduchým příkazem ***=paginate @advertisements*** vypsat stránkování. Toto stránkování se automaticky přizpůsobí nastavení podle bootstrapu a přidá tlačítka na listování.

```
@advertisements = @advertisement.page(params[:page]).per(9)
...
```

Výpis 24: Využití a použití stránkování v pohledu

Pomocí metody **page**, která je umístěna v gemu **Kaminari**, se v tomto případě nastaví to, že na jednu stránku připadne 9 inzerátu a jakmile se vytvoří 10-tý inzerát, tak se vytvoří nová stránka a přidají se tlačítka pro listování mezi inzeráty.

5.1.11 Implementace obrázků a tvorba sociálních tlačítek

K inzerátu podle zadání jsem přidal funkcionalitu na přidávání obrázku, ke každému inzerátu je možnost přidat obrázek, který se zobrazí na hlavní stránce společně s cenou a popisem. Obrázek s inzerátem lze editovat, mazat a prohlížet.

Obrázky jsou uloženy do databáze **pictures** s atributem **image** a jsou propojené navzájem s tabulkou **advertisements** pomocí cizího klíče se ukládají do společné složky public v Rails aplikaci. V modelu je to implementováno tak, že inzerát může mít více jako jeden obrázek a jeden obrázek je přiřazen jednomu inzerátu.

Pro souborový systém a možnost nahrávání obrázku do aplikace jsem použil gem **carrierwave**, který umožní nahrát jakýkoli soubor a pomocí metody **image_tag** vykreslit obrázek podle cesty kde je daný obrázek uložený. Bez této metody se vypíše pouze cesta k danému obrázku. Základní logika vytváření, úpravy a mazání se provádí v kontroleru **pictures** a dále se sním pracuje i v kontroleru **advertisements** jako s cizím klíčem.

Pomocí renderu a parciálních šablon jsou pak vykreslovány obrázky do samotného pohledu. V pohledu se provádí stylizování pomocí CSS a SCSS. Pro lepší efekt jsem využil gem **fancybox**, který umožní po kliknutí na obrázek efektivní zvětšení.

```
- for x in @advertisement.pictures
  %div{:style => "margin-top: 20px;"}
    = link_to image_tag(x.image.url(:thumb)), x.image.url, class: "fancybox"
...
```

Výpis 25: Parcialní šablona z advertisements _images.html.haml

Výpis znázorňuje použití stylu, cyklu a metody **image_tag** pro vygenerování a zobrazení obrázku s využitím gemu **fancybox**. Pro zobrazení této šablony ve **show.html.haml** se použije příkaz ***render 'images'***.

```
<% for x in @advertisements %>
  <% if iteration_count == 1 && x.pictures.size == 1 && counter== 0 %>
  <%= link_to image_tag("/no-img.jpg"), category_advertisement_path(x.category,
      x) %>
...
```

Výpis 26: Parcialní šablona z categories _showPictures.html.erb

Cyklus prochází kategorie pod kterýma jsou umístěné obrázky a kontroluje podmínkou, jestli existují nějaké zadané obrázky nebo jestli se má použit defaultní obrázek. Poté pro vygenerováni parciální šablony se použije příkaz ***render 'showPictures'*** v jakékoli podsložce, kde se nachází tato šablona.

```
<%= render_shareable :buttons=>['twitter','facebook'],
    :button_caption=>['Twitter','Facebook']%>
```

Výpis 27: Generování sociálních tlačítek

Sociální tlačítka se generují přes gem **shareable** a pomocí výše zmíněného příkazu, který je umístěn do záhlaví webové stránky, vygeneruje sociální tlačítka pro sdílení na sociálních sítích. Sdílet se bude právě aktuální stránka, na které se nacházíme pomocí definovaných pravidel v routes.

5.2 Importy z bank

Cílem bylo vytvořit webovou službu, aplikaci, která se bude starat o zákazníky a jejich bankovní účty. Pomocí emailu aplikace stahuje data a ukládá do vlastní databáze a zobrazuje přehledy ze všech bank, který si klient uloží a navolí. V aplikaci lze vidět přehledné pohyby na účtech, a pokud je to uvedeno ve feedu, tak i celkové množství financí na účtu.

Veškeré informace se berou z emailu, které posílají banky, buď ve formě výpisů nebo pohybů na účtu. Email musí být v aplikaci totožný s emailem, na který se posílají finanční údaje. Po obdržení emailu aplikace rozparsuje email podle klíčových slov a vytáhne data, která jsou potřebná pro další zpracování. Emailová komunikace je pouze simulační a funguje ve vývojářském módu. Ostrá verze bude napojena na konkrétní emaily. Toto řešení je zvoleno, abychom mohli otestovat funkčnost aplikace.

Scénář uživatele, který při využívání služeb aplikace se musí nejprve zaregistrovat a spárovat účet se svým emailem a bankovním účtem. Každý uživatel může sledovat více bankovních účtu naráz, tím se stává aplikace téměř jedinečná. Ověření, jestli je to právě ten uživatel, probíhá přes email a čísel bankovních účtů. Když další uživatel zadá stejné údaje jako předchozí nebo jakýkoli jiný, tak se vyhodí chybová hláška.

5.2.1 Zadání projektu

Zadání dalšího projektu, na kterém jsem se podílel, byla webová služba aplikace na téma importy z bank. Jednalo se o projekt, ve kterém si klienti budou přidávat pomocí webové služby své bankovní účty a dostanou plnou kontrolu nad jejími pohyby v elektronickém bankovnictví. Zadání mělo následující požadavky: Využívání gitu pro správu verzí zdrojového kódu, registrace a správa uživatele, na vstupu možnost parsování emailu, statistiky o pohybech, vytvoření homepage pro neregistrované a poté registrované zákazníky, použít technologii Bootstrap a nakonec tvorba testů, lokalizací, překladů a vyhledávání.

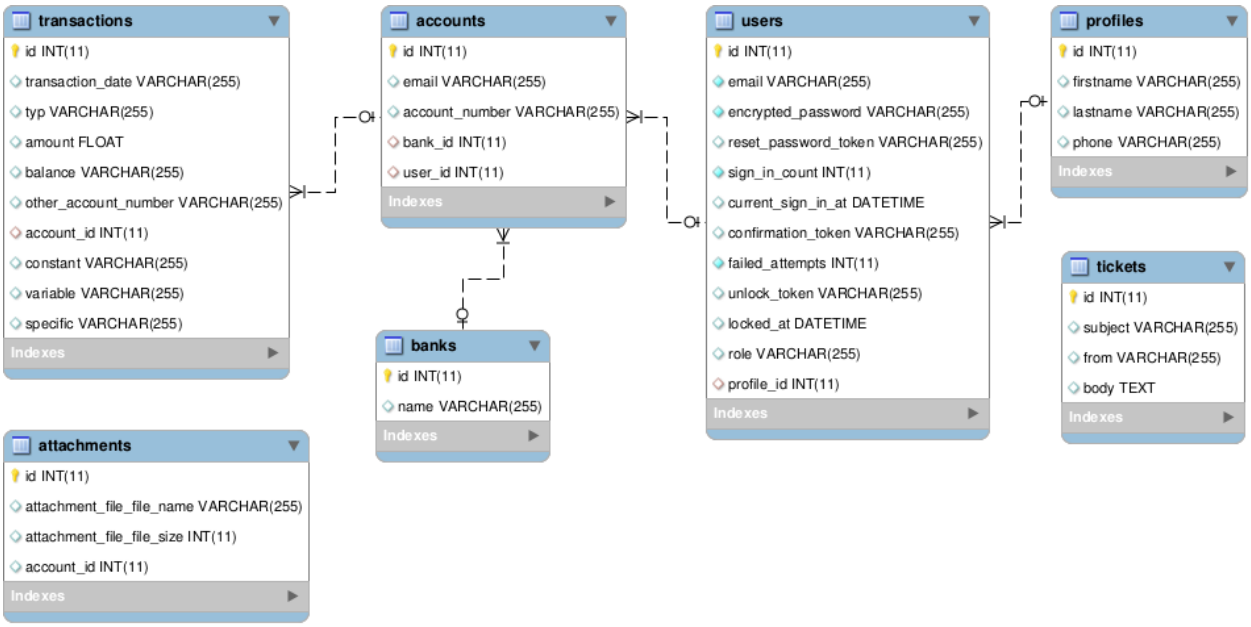
5.2.2 Návrh databáze a tvorba MVC

Pro tuto aplikaci byla použita databáze MySQL místo defaultně integrovaného SQLite. Nastavení proběhlo podobně jako u předchozího projektu, který běžel také na MySQL. Nastavil jsem jména databází pro tři vývojové prostředí a to **bidev** pro development, **bitest** pro testování a **production** pro ostrou verzi.

```
development: database: db/bidev.mysql2
test: database: db/bitest.mysql2
production: database: db/production.mysql2
```

Výpis 28: Základní nastavení databáze

Výše je výpis znázorňující nastavení jmen databází pro development, test a production založené na MySQL.



Obrázek 4: EER diagram importy z bank

Do tabulky **transactions** se ukládají data, která jsou rozparsována metodou z tabulky **tickets** a do tabulky **tickets** jsou data vkládaná prostřednictvím emailu, soubory které jsou přikládané k emailu jsou uloženy v tabulce **attachments**. V tabulce **users** jsou jednotlivý uživatelé, kde každý má svůj profil. Tabulka **banks** slouží k vytvoření jmen banek a prostřednictví tabulky **accounts** jsou zobrazována a propojena.

Místo generování aplikace pomocí Scaffoldingu jako u projektu inzertní webové služby jsem využil možnost generovat samotný model pomocí příkazu ***rails generate model Account***. Důležitá syntaxe je psaní modelů v jednotném čísle a musí začínat velkým písmenem, jelikož jsou to Ruby třídy. Ruby on Rails je natolik chytrý, že spoustu věcí udělá za nás, to znamená, že tabulky které se vytvoří z modelu, převede do množného čísla. Generátor také vytvoří k samotnému modelu migraci a vytvoří tabulku Accounts v databázi.

Po vygenerování modelu jsem musel ještě vygenerovat samotný kontroler pomocí příkazu ***rails generate controller Accounts***. Vytvoří se nám soubory (kontroler a views), které jsou potřebné pro komunikaci s MVC architekturou a uživatelem.

Totéž jsem opakoval pro další tabulky a pomocí migrace příkazem ***rails g migration AddColumm*** jsem přidával nebo odebíral atributy z tabulek databáze. Vygenerované soubory z výše uvedeného příkazu musí probíhat za sebou jak jsou vytvořené z důvodu funkčnosti celé aplikace. Mohlo by se stát, že další kolegové si stáhnou vývojovou verzi ve které budou chybět migrace a tím by si museli vymazat celou databázi a přišli tak o svoje data.

```
class AddSymbolToTransactions < ActiveRecord::Migration
  def change
    add_column :transactions, :constant, :string
    add_column :transactions, :variable, :string
    add_column :transactions, :specific, :string
  end
end
```

Výpis 29: Migrace pro přidání sloupců do tabulky

Pomocí této vygenerované migrace se přidají sloupce **constant**, **variable** a **specific** do tabulky **Transactions**, ve formátu string.

5.2.3 Modely, vazby a směrování

V modelech jsou vazby propojeny vztahy tak, že **Account** má nastavené belongs_to :bank, belongs_to :user a has_many :transactions. **Bank** má pouze hasy_many :accounts, **User** hasy_one :profile a hasy_many :accounts, **Transaction** belongs_to :accounts a **Profile** belongs_to :user. Tato propojení musí být u každé aplikace aby se zajistili správné propojení.

```
class Account < ActiveRecord::Base
  belongs_to :bank
  belongs_to :user
  has_many :transactions, dependent: :destroy
...
end
```

Výpis 30: Vazby v modelu models/account.rb

Ze třídy **Account**, která dědí ze základní třídy ActiveRecord, lze vyčíst, že může mít jednu banku a uživatele. Také může mít mnoho transakcí.

V modelech se mimo jiné také provádí validace a u modelu **Account** se ověřují správně zadané emaily a čísla účtu. Také se ověřuje jejich délka pomocí metody **length** a metoda **presence** zajišťuje důležitost atributu tím, že musí být za každou cenu zadána alespoň nějaká hodnota.

```
class Account < ActiveRecord::Base
  validates :email, presence: true, length: {minimum:2, maximum:50}
  validates :account_number, presence: true
end
```

Výpis 31: Validace v modelu models/account.rb

Směrování se provádí v souboru **config/routes.rb** a pomocí root metody se navolí úvodní zobrazovací stránka. S metodami **resources** se implementují všechny dostupné routy. Pro překlad cesty v URL se použije standart **i18n.t('routes.banks')**, který přeloží cestu z angličtiny na definovaný jiný název v tomhle případě na češtinu. Překlady jsou definované v souboru **config/locales**.

```
resources :tickets
resources :transactions
resources :banks, path: I18n.t('routes.banks')
root 'welcome#index'
...
```

Výpis 32: Routování v souboru config/routes.rb

5.2.4 Přihlášení a registrace uživatele

Podobně jako u předchozí aplikace, se registrace a přihlášení řešilo přes gem devise, který v sobě uchovává spoustu nastavení a konfigurací. Jedna z konfigurací je nastavení emailové služby, které se stará například o potvrzování registrace nového uživatele, resetování zapomenutého hesla nebo zablokování účtu po několika špatných pokusech při zadávání hesla. V této aplikaci pro větší

bezpečnost je zavedená právě tato služba, to znamená, že při registraci je zapotřebí ověřit na uvedeném registračním emailu token, který je vygenerován devisem.

Po zadání příkazu ***rails generate devise User*** se vygeneruje model user a sním i potřebné routování. V modelu user je potřeba povolit vazby **:confirmable** a **:lockable** pro potvrzování emailu a popřípadě zablokování účtu. Dále se musí nastavit v souboru **development.rb** v **config/environments**, action_mailer pro příjem lokálních emailu a otevíráni emailu v novém okně prohlížeče, všechny tyto operace se provádí pouze ve vývoji, ostrá verze bude napojena na oficiální emaily zákazníků. Nastavení délek hesel, potvrzování emailu, zablokování účtu se nastavuje v souboru **devise.rb**, který se nachází v **config/initializers**.

```
config.lock_strategy = :failed_attempts
config.maximum_attempts = 3
...
```

Výpis 33: Nastavení povolení v config/initializers/devise.rb

Výpis zobrazuje nastavení zablokování účtu. Uživatel, který třikrát zadá špatné údaje, tak se mu zablokuje účet, a pro obnovení svého účtu musí potvrdit vygenerovaný token na emailu.

5.2.5 Tvorba kontroleru a pohledů

Kontrolery mají tu vlastnost, že komunikují s pohledy a modely. Zajišťují funkční logiku a posílají informace do pohledu, kde se požadavky renderují uživateli. Každý kontroler může mít akce pro vytváření, editaci, mazaní, například pro různé příspěvky či uživatele, které se ukládají do databáze, nebo se s nimi dále pracuje.

Jelikož se pracuje s citlivými daty, tak aby se předcházelo vytváření stejného účtu a nedocházelo k bankovním podvodům v aplikaci, musí se zajistit v kontroleru **accounts_controller.rb** duplicita přidávání bankovního účtu u každého uživatele. Také se v kontroleru řeší zobrazení obsahu právě danému uživateli podle params a id.

```
def create
...
@account.save!
if Account.find_by(:account_number => @account.account_number, :email =>
   @account.email).nil?
else flash[:notice] = 'Chyba'
...
end
```

Výpis 34: Podmínka v kontroleru ověřující duplicitu

Akce **create** v kontroleru **accounts** zajišťuje ověření duplicity v čísle bankovního účtu a v emailu. Při zadání stejného účtu nebo emailu jiným uživatelem dojde k vyhození chybové hlášky.

Šablony, zobrazující se uživateli vychází z pohledů, které můžou dostávat informace a data z kontrolerů a ty pak z modelů. V projektu využívám k zobrazení důležitých informací uživateli pohledy a využil jeden z pohledů pro tvorbu dashboardu, který zobrazuje informace o přehledu účtu. Tyto informace se zobrazují ve **views/welcome/index.html.erb** a pomocí routování je tato šablona nastavena na root, tedy zobrazí se hned po přihlášení uživatele. Pohled udává informace o čísle účtu, emailu, o celkovém příjmu, výdeji a o možnosti tato data exportovat.

```
<% transactions.each do |t| %>
<% income\_sum = income\_sum + t.amount %>
<li> Celkovy prijem za mesic: <%= income\_sum %></li>
...
```

Výpis 35: Ukázka šablony z dashboardu

Zdrojový kód výše znázorňuje průchod cyklem všechny transakce a do proměnné **income_sum** ukládá všechna čísla, tedy částky uživatele na účtu a s čítává je do jednoho výsledného součtu. Uživatel pak uvidí svůj celkový příjem.

5.2.6 Posílání a parsování dat z emailu

Posílání emailu zajišťuje gem **Mailman**, který nám vytvoří virtuální server pro posílání emailu. Přidáním gemu do souboru Gemfile v rootu aplikace a příkazem ***gem install mailman*** se nainstaluje virtuální server s POP3 clientem, který je kompatibilní téměř s každou verzí Ruby on Rails.

Ve složce script se definuje a konfiguruje mailman server. Je potřeba nastavit uživatelské jméno a heslo na daný server, dále pak v samotné run metodě zajistit vytvoření objektu reprezentující daný email, který je následně uložen do databáze. Testovací data, feedy, které jsem dostal od banky (jedná se o výpisy, přehledy nebo pohyby na účtech) je uložen do souboru **mailman_test.erb**, ze kterého se vytvoří objekt do tabulky tickets s atributy subject, body, from a pokaždé kdy je poslán virtuální email příkazem ***cat mailman_test.eml | script/-mailman_server.rb***, tak se email uloží do databáze.

```
Mailman.config.pop3 = {
  server: 'pop.gmail.com', port: 995, ssl: true,
  username: "bankovniImporty@gmail.com",
  password: ""
}
Ticket.new(subject: message.subject, body: the_message_text, from: message.from
    .first)
...
```

Výpis 36: Základní nastavení mailman serveru v souboru mailman_server.rb

Ukázka výše je konfigurace POP3 serveru, která slouží pro stahování emailových zpráv ze vzdáleného serveru na klienta. Nastavuje se zde server, port, ssl, username a password. Následně se vytváří objekt Ticket s parametry subject, body a from.

Pro vytáhnutí dat z emailu a uložení potřebných informací do proměnných je zapotřebí data rozparsovat. K tomu slouží tabulka **tickets** do, které jsou ukládány všechny emaily. Na začátku parsování je zapotřebí rozeznat email o který se jedná, protože každá banka má jinou strukturu emailu a pro každou banku musí být vytvořený jedinečný parser.

Abychom mohli vytvořit parser pro danou banku, musíme znát vstup, který je uložen do tabulky **tickets**. Výstupem by pak měl být email bankovního účtu, zůstatek na účtu, datum transakce, částka, číslo protiúčtu, konstantní, variabilní a specifický symbol. Tyto informace, jestli jsou dostupné, se zobrazí uživateli při přehledech bankovních účtů.

K samotnému parsování musíme hned na začátku zjistit, od jaké banky pochází feedy a to zjistím podle emailu dané banky. Abychom data mohly dále parsovat, musíme zjistit a najít v emailu klíčová slova, podle kterých si vyhledáme index daného prvku, s kterým se dále pracuje. Po nalezení všech údajů z emailu vytvořím nový objekt **Transaction** pomocí metody **create**, který se uloží do databáze.

```
def self.methode(myTicket)
  datum = getParseItem(myTicket.body, "Date", 6, "\n")
  def self.getParseItem(body, what, letters, ending)
    index = body.index(what)
    text = body[index+letters..-1]
    endIndex = text.index(ending)
    return text[0..endIndex-1]
  end
end
```

Výpis 37: Používání metody pro parsování

Do metody **methode** se pošlou data z emailu, který řeší mailman server. Proměnná **datum** parsuje data pomocí metody **getParseItem**, kde se zjišťuje index zadaného slova, poté celý text od daného indexu, kde se přičítají mezery plus nadbytečné písmena a končí na konci celého emailu, abychom zjistili přesný konec, tak k tomu slouží proměnná **endIndex**, která nám zajistí přesný konec datumu a ne konec celého emailu. Celá metoda poté vrací pouze text, který potřebujeme a uloží ho do tabulky **transactions**.

Pro snažší a čitelnější kód, jsem vytvořil services v **lib/services/imports.rb** a zde vytvořil metodu **parse** pro parsováni dat. Tato metoda se volá při vytváření samotného tiketu ve **script/mailman_server.rb**. Původně veškeré parsování bylo v modelu **ticket.rb** ale pro přehlednost a znovupoužitelnost kódu jsem to udělal tímhle způsobem.

```
import = Services::Imports.new
import.parse(myTicket)
...
```

Výpis 38: Volání services

5.2.7 Exportování dat

Jako téměř každá aplikace, která pracuje s daty, se neobejde bez exportu dat do různých formátů. Proto jeden bod ze zadání bylo vytvořit exporty dat do různých Existujících formátů jako XML, PDF, CSV a další. Vytvářel jsem formát pro stahování dat do .csv formátu, který se stáhne po kliknutí na odkaz ve výpisech transakcí.

Při vytváření exportu jsem musel jako první věc přidat novou cestu do souboru **config/routes.rb** a to ***get :export_all_to_csv***. Abychom mohly správně odkazovat, pak bylo zapotřebí vytvořit šablonu ve **views/accounts/** se stejným jménem jako je v **routes.rb**. Metoda pro vytváření a stahování dat je umístěna do souboru **accounts_controller.rb** a aby se stahovaly uživatelovi transakce, tak další metoda, která definuje jaký formát a co se má stahovat je umístěna do modelu **transactions.rb**. Poté je přidán odkaz do **accounts/show.htlm.haml** pro stahování transakcí.

```
def self.to_csv
  attributes = %w{id transaction_date typ amount balance other_acc}
  CSV.generate(headers: true) do |csv|
    csv << attributes
    all.each do |transactions|
      csv << attributes.map { |attr| transactions.send(attr)}
    end
  end
end
```

Výpis 39: Metoda pro export z modelu transactions.rb

V metodě **self.to_csv** se definují do proměnné **attributes** hodnoty, které se vytahují z databáze a udávají, v jakém formátu se data budou zapisovat do souboru. Poté se generuje soubor, který prochází dané transakce z databáze a mapuje podle atributů.

```
def export_all_to_csv
@transactions = Transaction.where(account_id: current_user.accounts)
  respond_to do |format|
    format.html
    format.csv {send_data @transactions.to_csv}
  end
end
```

Výpis 40: Metoda pro export z kontroleru accounts_controller.rb

V samotném kontroleru se kontroluje, aby se vypisovaly pouze transakce, které patří přihlášenému uživateli. Pomocí metody **send_data** se vytváří soubor, který se formátuje podle definovaného formátu a volá metodu **to_csv** z modelu transactions.

5.2.8 Vyhledávání a stránkování

V kontrolerech se nastavují informace, které souvisí s vyhledáváním a stránkováním. U každého kontroleru u kterého chceme mít vyhledávání, musíme přidat metodu **search**, kterou nám zajišťuje gem Ransack. Totéž platí u stránkování, kde stačí přidat k danému objektu, u kterého chceme mít stránkování metodu **page**, kterou nám zajistí gem Kaminari a poté zavolat v daném pohledu metodou paginate.

```
@transactions = Transaction.all.where(:account_id =>
@account).order(transaction_date: :DESC).search(params[:search]).page(params[:
    page]).per(10)
...
```

Výpis 41: Vytvoření v kontroleru stránkování s vyhledáváním

Objekt **transactions** vyhledává všechny transakce podle účtu uživatele a jeho id a zároveň vrací seřazenou transakci. Metody search a page říkají a zavádějí informace danému objektu o tom, že někde v pohledu budou volány pro splnění svého účelu, tedy pro vyhledávání a stránkování v transakcích.

```
= form_tag user_account_path, :method => 'get' do
  zadej typ platby
  = text_field_tag :search, params[:search]
  = submit_tag "Search", :typ => nil
...
```

Výpis 42: Vytvoření formuláře v pohledu pro vyhledávání a stránkování

Zdrojový kód výše znázorňuje vytvoření formuláře, který je přepsán do haml formátu. Tento případ ukazuje vyhledávání podle typu dané platby, tedy pro příjem nebo výdej.

5.3 Překlady a lokalizace aplikace

Překlady a lokalizace, vytváří externí databázi obsahující překlady jednotlivých části aplikace, například chybové hlášky, formáty data a času, měny a podobně. Dále pak jsou důležitým faktorem úspěchu každé aplikace na světové úrovni. Jako standart pro internacionalizaci jsem použil i18n, kde 18 je počet písmen v anglickém (internationalization). Standart je rychle použitelný a snadno rozšiřitelný pro překlad každé aplikace do více než jednoho jazyka. Poskytuje multijazyčnou podporu a abychom mohli využívat překlady se základními nastaveními musí se nastavit v souboru **config/application.rb** standart a popřípadě časová zóna společně s různými konfiguračními a nastavovacími knihovnami [22].

```
config.time_zone = 'Prague'
config.i18n.available_locales = %w(cs en)
config.i18n.default_locale = :cs
...
```

Výpis 43: Nastavení překladu a lokalizace v souboru config/application.rb

Mutaci neboli překlady jsem převáděl do anglického jazyka, jelikož je celosvětově nejvíce rozšířený. V aplikaci se překlady píši do adresáře **config/locales**, zde probíhá veškerá konfigurace mutací. Rails od samého začátku ví, kde se tyto lokalizace a překlady nachází, tak není potřebné nastavovat další cesty pro nalezení ale je zapotřebí správně dodržovat strukturu psaní překladů.

```
cs:
  banks:
new:
  title: "Vytvorit banku"
index:
  newBank: "Pridat banku"
  name: "Nazev"
  title: "Seznam bank"
...
```

Výpis 44: Překlad z adresáře config/locales/cs/cs.yml

Ve zmíněném adresáři se provádí překlady pro jednotlivé pohledy v tomto výpisu konkrétně pro banks a její akce new a index. Struktura překladu simuluje adresářovou strukturu jednotlivých pohledů a podle této adresářové struktury se používají a aplikují překlady a lokalizace.

```
%h1= t('.title')
%th= t('.name')
= link_to t('.newBank'), new_bank_path
...
```

Výpis 45: Použití vytvořeného překladu ve views/banks/index.html.haml

Překlady, které jsem napsal v souboru **config/locales** se aplikují v pohledu přidáním metody **t()** a názvu, který je definován jako překládající. V tomto případě se nám přeloží tittle na Seznam bank, name za Název a newBank za Přidat banku. Obdobně fungují překlady u ostatních pohledů.

5.4 Testování aplikace

Testování a psaní testů pro aplikace jsem prováděl jak pro inzertní webovou službu, tak i pro importy z bank. Testy lze psát a možno otestovat přímo za běhu aplikace. Máme zde 3 testovací databáze Test, Productin a Development. Testy se převážně provádí na Testovací databázi, aby při vydání ostré verze nebyli žádné problémy. Testy lze provádět jak na databázi SQLite tak i na MySQL. Oba projekty běžely na databázi MySQL a při psaní testů byli procesy téměř totožné.

K testování a samotnému prověření, jestli jsou data a metody skutečně správně napsané a neobsahují žádné chyby, jsem použil gemy **rspec-rails** a **factory_girl_rails**, které jsem navolil, aby byli použité pouze ve vývojové a testovací verzi, do produkčního systému musí být vše správně otestované.

```
group :development, :test do
  gem "rspec-rails"
  gem "factory_girl_rails"
end
...
```

Výpis 46: Označení skupiny v Gemfile pro development a test

Příkazem ***rails generate rspec:install*** se nám vytvoří soubor **spec/** ve kterém se provádí a píší testy, aby fungoval i gem factory, tak je potřeba jej nainstalovat příkazem ***gem install factory_girl_rails***. Pomocí příkazu ***rspec spec/models/nazev.rb*** se spouští testy a v cmd lze vidět výsledek zda-li test prošel nebo neprošel.

Pro testování aplikace je zapotřebí vytvořit si vlastní databázi a naplnit ji daty, protože při testování se vždy smaže testovací databáze a vytvoří se nová z předefinovaných dat. Abychom mohli aplikaci testovat, slouží k tomu gem **FactoryGirl**, který vytvoří složku **spec/factories**.

```
FactoryGirl.define do
  factory :advertisement do
    description 'Nejaky popis'
    price 1234
    post_date { Time.now }
    association :user
...
```

Výpis 47: Vytvoření testovacích dat ze spec/factories/advertisement.rb

Následující zdrojový kód nám vygeneruje testovací data tj. objekty podle atributů v databázi. Testovací data neberou data z databáze, ale právě z těchto vytvořených. Jako příklad uvádím testování inzerátů, u ostatních tabulek je to obdobné.

U inzerátu jsem tedy testoval vztahy mezi modely čili cizí klíče, vazby a relace mezi nimi pomocí metody **belongs_to**, která ošetřuje relace mezi modely. Dále tento kód jsem opakoval pro další relace, které se nacházejí v inzerátech. Když očekáváme, že se vymaže objekt, který je v relací s inzeráty, tak se přidává metoda **dependent(:destroy)** za předchozí metodu. Při testování modelu jsem také testoval validaci atributu pomocí **validate_presence_of**. Poté jsem testoval i správnou délku celého stringu pomocí metody **ensure_lenght_of** a také celé kladné číslo pomocí metody **validate_numericality_of**.

```
describe "relation" do
  it { should belongs_to :category }
  it { should have_many(:pictures).dependent(:destroy) }
end
describe "validation" do
  it {should validate_presence_of :description}
end
describe "is valid: price is integer" do
  it { should validate_numericality_of(:price).only_integer }
end
...
```

Výpis 48: Tvorba tesů u inzerátů

Na ukázku jsem vypsal některé testy, které jsem testoval, můžeme vidět testy z testování relací poté validací a nakonec test na ověření celočíselného čísla.

Testoval jsem také základní metody v kontroleru. Vytvořil jsem si metodu, která se spouští při každém testu daného modelu.

```
before(:each) do
  @advertisement = FactoryGirl.create(:advertisement)
end
```

Výpis 49: Testování metod před spuštěním testu

Tato metoda vytváří do proměnné **advertisement** objekt inzerát a ověřuje, jestli se vytvoří nebo ne. Tento objekt je předem definován ve **FactoryGirl**.

6 Znalosti a zkušenosti

6.1 Teoretické a praktické znalosti získané v průběhu studia

Znalosti, které jsem získal v průběhu studia, jsou velmi bohaté, naučily mě být samostatný pracovat rychle a efektivně a dodržovat striktně data konečných odevzdávání projektů. V praxi využiji jak teoretické tak i praktické myšlenky, principy a zkušenosti, které jsem získal při studiu.

Při tvorbě zadaných úkolů jsem využil znalosti především z předmětů (SPJA) a (URO), kde jsem se naučil správného rozvržení uživatelského rozhraní a dodržení různých typografických a vizuálních pravidel. Ze skriptovacích programovacích jazyků jsem využil základní principy jazyka Python a mohl jej tak využít právě na praxi v jazyku Ruby a frameworku Ruby on Rails postaven na architektuře MVC.

Další velmi cenné a důležité informace jsem získal v předmětech tvorby aplikací pro mobilní zařízení a programovacích jazyků. Vyzkoušel jsem si funkčnost a principy objektového programování a základní funkčnost webové aplikace, jejich stav a chování při menším a větším rozlišení jak monitoru, tak i mobilního telefonu. Tyto znalosti jsem využil při tvorbě webových aplikací a vytvářel jsem aplikace multiplatformní.

Celkově si myslím, že praxe spojená se studiem byla časově náročná, ale na druhou stranu mi dala velmi mnoho užitečných firemních zkušeností a myslím si, že při psaní jiné bakalářské práce bych tyto zkušenosti nezískal.

6.2 Chybějící znalosti a dovednosti

Mezi mé chybějící znalosti a dovednosti patřily převážně znalosti databázových systémů a návrhových vzorů, jelikož jsem na studijním oboru mobilních technologií, tak daná problematika se tam neprobírá. Dále mi chyběla hlubší znalost programování a znalost celkového frameworku Ruby on Rails. Musel jsem se doučit a prohloubit znalost HTML, CSS a Model, View, Controller architekturu. Pro správnou funkčnost aplikace se musely provádět testy, což je velmi důležité a mnohdy podceňovaná záležitost, proto jsem se musel naučit, jak správně a efektivně tyto testy psát.

Velkou nevýhodou a chybějící znalosti při programování v Ruby on Rails je rychlé zapomenutí určitých postupů při vytváření aplikace, proto jsem si udělal vlastní systém, při kterém si rychle a jednoduše naleznu tu část, kterou právě potřebuji. V začátcích byl velký problém v orientaci datové a složkové struktury celé architektury, ale s použitím vhodného textového editoru, který obsahoval i složkový systém, jsem vědomostní nedostatky odstranil.

Poslední neznámou byla pro mě velmi využívaná technologie Bootstrap a systém Git pro správu verzi svého zdrojového kódu. Tato neznalost mi ze začátku přinesla mnoho neúspěchů a sebrala spoustu času, ale s pomocí kolegů, kteří mi doporučili studijní materiály, jsem se v tom začal orientovat a v následujícím období vše s úspěchem vyřešil.

6.3 Dosažené výsledky a hodnocení v průběhu odborné praxe

S nástupem na praxi jsem uměl pouze některé základy z několika oblastí, které jsem se naučil při studiu, ale firemní požadavky mě donutily věnovat se danému problému několik dní v týdnu a já se vše celkem jednoduše naučil a zapamatoval.

Mezi základní dosažené výsledky bych zařadil funkční webové systémy a stránky, na kterých jsem pracoval. Praxe mě naučila efektivnímu přístupu na danou problematiku a určení vhodného nástroje popřípadě gemu či technologie nebo programovacích jazyků jako je HTML nebo HAML. V frameworku Ruby on Rails jsem využil mnoho programovacích jazyků od vzhledu webových stránek jako je HTML, HAML a CSS, SCSS, tak k vnitřní struktuře s jazykem Ruby a frameworkem Ruby on Rails.

Vyvíjení z počátku probíhalo na operačním systému Windows, později jsem však zjistil, že lepší vychytávky a vymoženosti má operační systém Linux s distribucí Ubuntu a naučil jsem se tak lépe ovládat a orientovat v tomto prostředí.

Inzertní webová služba prohloubila mé zkušenosti v oblasti prodeje a v celkové funkčnosti vnitřní struktury komunikace se zákazníkem. V druhém projektu, na kterém jsem dělal, jsem se dozvěděl funkčnost a formát emailu téměř všech bank. Dále obsah a formu komunikace se zákazníkem ze strany banky a formát, který využívá při posílání emailu o pohybech na účtech. Naučil jsem se parsovat a formátovat data z emailu od různých bank.

Ve výsledku jsem se naučil mnoho nových věcí, jako je práce v týmu, to bylo pro mě velmi důležité a obohacující, v podstatě za výsledkem mé práce stojí celý tým, který spolupracoval na projektu a ne jednotlivec, který by při rozsáhlých projektech na jedné části strávil mnohem více času.

7 Závěr

Při nástupu do firmy Railsformers, s.r.o. na začátku září jsem nevěděl, co mě čeká, byla to moje první životní zkušenost ve firmě, která se zabývá tvorbou webových aplikací. Musím říct, že jsem byl mile překvapen nejen z kolektivu, který byl velmi ochotný a vstřícně mi vždy pomohl, ale i z čerstvě načerpaných praktických a teoretických zkušeností. Proto jsem velmi rád, že jsem mohl tuto praxi absolvovat.

Další velmi cennou a důležitou zkušenost, kterou jsem získal, byla aktivita ze strany firmy a její celkový chod, organizace a průběh fungování. Dozvěděl jsem se, jak probíhá proces při přijímání nových zaměstnanců. Musel jsem absolvovat různá školení ohledně bezpečnosti, přístupu do budovy a pracovní náplně.

V první polovině praxe jsme se ještě s dalšími kolegy zaučovali a připravovali na náš první projekt, který jsme dělali v týmu a společně spolupracovali, což mě velmi povzbudilo a nadchlo. Naučil jsem se větší týmové spolupráci, flexibilnímu řešení různých naléhavých problémů v časové a napěťové situaci. V další části mé bakalářské praxe jsem s mými kolegy pracoval na projektu importy z bank.

Celkově náplň praxe byla pro mě velký přínosný skok do reality, silně mě obohatila a ukázala mi, že se člověk musí stále zdokonalovat a učit se něčemu novému. Doufám, že tyto cenné zkušenosti využiji dále ve svém vzdělávání a později mi budou prospěšné v mém profesním životě.

Literatura

[1] Railsformers, s.r.o. Railsformers [online]. Ostrava, 2016 [cit. 2016-04-06]. Dostupné z: http://railsformers.com/cs

[2] VŠB. VŠB-TU Ostrava [online]. Ostrava, 2016 [cit. 2016-04-06]. Dostupné z: http://www.vsb.cz/cs/

[3] What is SEO. Search engine land [online]. 2016 [cit. 2016-04-06]. Dostupné z: http://searchengineland.com/guide/what-is-seo

[4] Rails. Ruby on Rails [online]. 2016 [cit. 2016-04-06]. Dostupné z: http://rubyonrails.org/

[5] Recepty online. Srecepty [online]. Ostrava: General recipe, s.r.o., 2014 [cit. 2016-04-06]. Dostupné z: http://www.srecepty.cz/

[6] Domácí účetnictví. SMoneyBox [online]. 2007-2016 [cit. 2016-04-06]. Dostupné z: http://www.smoneybox.com/cs

[7] Překlady aplikace. Go4Translate [online]. Ostrava [cit. 2016-04-06]. Dostupné z: http://www.go4translate.com/cs

[8] Online systém. Hedurio [online]. Ostrava: Railsformers, s.r.o., 2014 [cit. 2016-04-06]. Dostupné z: https://hedurio.com/

[9] GitHub repository Rails. Rails [online]. GitHub, 2008 [cit. 2016-04-06]. Dostupné z: https://github.com/rails/rails

[10] Getting Started with Rails. Ruby on Rails [online]. David Heinemeier Hansson [cit. 2016-04-06]. Dostupné z: http://guides.rubyonrails.org

[11] Ruby on Rails Security. Ruby on Rails [online]. David Heinemeier Hansson [cit. 2016-04-06]. Dostupné z: http://guides.rubyonrails.org/security.html

[12] Ruby Version Manager. RVM [online]. [cit. 2016-04-06]. Dostupné z: https://rvm.io/

[13] RubyGem. RubyGems [online]. Nick Quaranto, 2009 [cit. 2016-04-06]. Dostupné z: https://rubygems.org/

[14] Git Book. Git [online]. GitHub [cit. 2016-04-06]. Dostupné z: https://git-scm.com/book/cs/v1

[15] Ruby on Rails MVC. Tutorials point [online]. 2016 [cit. 2016-04-06]. Dostupné z: http://www.tutorialspoint.com/ruby-on-rails/rails-framework.htm

[16] MySQL. Dev Mysql [online]. Oracle, 2016 [cit. 2016-04-07]. Dostupné z: https://dev.mysql.com/doc/

[17] GitHub repository CanCan. Git Hub [online]. 2013 [cit. 2016-04-07]. Dostupné z: https://github.com/ryanb/cancan

[18] Ruby on Rails. Ruby on Rails Active Record [online]. 2016 [cit. 2016-04-07]. Dostupné z: http://guides.rubyonrails.org/active_record_basics.html

[19] GitHub repositroy Bootstrap. Git Hub [online]. 2016 [cit. 2016-04-06]. Dostupné z: https://github.com/twbs/bootstrap-sass

[20] Request response cycle [online]. New York: Codecademy, 2016 [cit. 2016-04-07]. Dostupné z: https://www.codecademy.com/articles/request-response-cycle-dynamic

[21] Haml. Haml info [online]. Hampton Catlin, 2015 [cit. 2016-04-06]. Dostupné z: http://haml.info/

[22] Rails Internationalization (I18n) API. Ruby on Rails i18n [online]. 2016 [cit. 2016-04-07]. Dostupné z: http://guides.rubyonrails.org/i18n.html