Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Aplikace pro vyhledávání a nabízení prací pro studenty

Adam Hrouda

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Adam Hrouda

Systémové inženýrství a informatika
Informatika

Název práce

Aplikace pro vyhledávání a nabízení prací pro studenty

Název anglicky

Application for searching and offering student jobs

Cíle práce

Bakalářská práce je zaměřena na vývoj aplikace pro vyhledávání a nabízení jednorázových prací pro studenty. Hlavním cílem práce je navrhnout a implementovat webové aplikační rozhraní a ukázkovou aplikaci demonstrující možnosti jeho využití. Dílčím cílem je popsat použité technologie, postup a vytvořit dokumentaci zmíněného webového API.

Metodika

Bakalářská práce sestává ze dvou částí – teoretické a praktické. Metodiky zpracování teoretické části spočívá ve studiu odborných informačních zdrojů. Na základě syntézy zjištěných poznatků budou formulována teoretická východiska práce.

V praktické části bude navržena a implementována serverová část aplikace, která bude vystavovat využitelné aplikační rozhraní. Dále bude implementována ukázková aplikace, která bude toto rozhraní využívat. Výsledné aplikace budou otestovány, budou shrnuty poznatky získané během jejich vývoj a navrženy možnosti případného dalšího rozvoje. Při zpracování praktické části budou využívány standardní nástroje a metody softwarového inženýrství.

Poznatky a výstupy z teoretické i praktické části budou na závěr shrnuty.

Doporučený rozsah práce

35-40 stran

Klíčová slova

API, C#, .NET Core, práce pro studenty, Visual Studio

Doporučené zdroje informací

ČÁPKA, David. ITnetwork [online]. [cit. 2020-03-08]. Dostupné z: https://www.itnetwork.cz/
Mastering ASP.NET Web API. Mastering ASP.NET Web API. 2017. ISBN 9781786463951.
MICROSOFT. Microsoft Docs [online]. [cit. 2020-03-08]. Dostupné z: https://docs.microsoft.com/en-gb/
PRICE, Mark J. C# 8.0 and .NET Core 3.0. Packt 2019. ISBN 978-1788478120

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 12. 02. 2021

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Aplikace pro vyhledávání a nabízení prací pro studenty" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15. 3. 2021 _____

Poděkování

Rád bych touto cestou poděkoval Ing. Jiřímu Brožkovi, Ph.D. za poskytnuté rady a ochotu při konzultacích. Dále děkuji své rodině a blízkým za podporu při studiu.

Aplikace pro vyhledávání a nabízení prací pro studenty

Abstrakt

Tato bakalářská práce je zaměřena na návrh a implementaci aplikace pro vyhledávání a nabízení studentských prací. Práce je rozdělena na teoretickou a praktickou část.

V teoretické části jsou představeny využité technologie a zásady. Z technologií jsou popsány vybrané komponenty architektury .NET a frontendový framework Vue.js. Dále jsou popsány pojmy middleware a jednostránková aplikace. Také jsou zmíněny doporučené zásady při implementaci webového API.

V praktické části jsou nejprve stanoveny požadavky na aplikaci a je vytvořen datový model. Následně je popsána implementace vybraných funkcí aplikace. Tato část je doplněna i ukázkami příslušných zdrojových kódů. Serverová část aplikace je vytvořena pomocí technologie ASP.NET Core a klientská aplikace pomocí frameworku Vue.js.

Vyvinuté řešení je závěrem otestováno a jsou navržena možná rozšíření.

Klíčová slova: webové API, C#, .NET Core, práce pro studenty, ASP.NET Core, Vue.js, server side, client side, NuGet

Application for searching and offering student jobs

Abstract

This bachelor thesis focuses on proposal and implementation of application for searching and offering student jobs. The thesis is divided into theoretical part and practical part.

The theoretical part presents technologies and principles used in the practical part. Selected components of .NET architecture and frontend framework Vue.js are described. Furthermore, it clarifies terms like middleware and single-page application. Recommended principles for web API implementation are mentioned.

In the practical part, the requirements for the application are first defined and then a data model is created. Subsequently, the implementation of selected application functions is described, and examples of relevant source codes are provided. The backend of the application is implemented using ASP.NET Core and the frontend using framework Vue.js.

Finally, the developed solution was tested and suggestions for further improvements were written.

Keywords: web API, C#, .NET Core, student jobs, ASP.NET Core, Vue.js, server side, client side, NuGet

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

API – Application Programming Interface
ASP – Active Server Pages
CLR – Common Language Runtime
DTO – Data Transfer Object
EF – Entity Framework
HTML – Hypertext Markup Language
HTTP – Hypertext Transfer Protocol
IL – Intermediate Language
JSON – JavaScript Object Notation
JWT – JSON Web Token
ORM – Object–Relational Mapping
REST – Representational State Transfer
RPC – Remote Procedure Call
SOAP – Simple Object Access Protocol
SPA – Single-Page Application
SQL – Structured Query Language
URI – Uniform Resource Identifier
WSDL – Web Services Description Language

1 Úvod

Část studentů svůj volný čas věnuje brigádám. Jejich důvodem bývá získání zkušeností pro pozdější zaměstnání nebo zajištění finančních prostředků. Existuje řada webových stránek, na kterých jsou inzerovány pracovní příležitosti. Problém může nastat ve chvíli, kdy se student začne se zaměstnavatelem domlouvat na pracovní době. Někteří studenti věnují studiu více času než ostatní. Jiní neobětují studiu tolik času, ale mohou mít ještě jiné zájmy a povinnosti. Rovněž existují studijní obory, kde není dán pravidelný týdenní rozvrh. Tyto skutečnosti mohou vést k zamítnutí spolupráce, neboť student nemůže odpracovat požadované hodiny nebo nemůže zajistit pravidelnou docházku.

Řešením této situace může být třetí strana, která se postaví mezi poskytovatele práce a studenta. Poskytovatel se stane zákazníkem této třetí strany. Ta jej odstíní od činností jako je výběr vhodných zájemců, koordinace a vyplácení odměn. Navíc bude mít brigádníky pouze tehdy, kdy jich bude potřeba. Student se v takovém případě bude přihlašovat pouze na jednorázové pracovní nabídky v časovém rozsahu několika hodin. Zároveň si vybere nabídky pouze v termínech a v lokacích, které mu vyhovují.

V rámci této práce bude vyvinuto zmíněného řešení v podobě webového API a ukázkové webové aplikace, která bude toto rozhraní konzumovat.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem této práce je návrh a implementace aplikace na sjednávání jednorázových studentských prací. Ta bude tvořena serverovou částí vystavující webové aplikační rozhraní a ukázkovou klientskou aplikací, která zmíněné rozhraní bude využívat. Dílčími cíli práce je představení použitých technologií, popsání postupu a vytvoření dokumentace webového API.

2.2 Metodika

Teoretická část práce spočívá ve studiu odborné literatury a dokumentací týkajících se využitých technologií a konceptů. Získané poznatky budou následně syntetizovány do teoretických východisek práce.

V praktické části budou autorem stanoveny požadavky na aplikaci. Na základě těchto požadavků bude navržena a implementována serverová část aplikace pomocí technologie ASP.NET Core 3. Následně bude implementována i klientská část aplikace, která představí funkce vyvinutého API. Pro její vývoj bude využit framework Vue.js.

Nakonec bude řešení otestováno a na základě získaných zkušeností dojde k formulování závěru a sepsání možných rozšíření aplikace.

3 Teoretická východiska

3.1 REST API

API neboli rozhraní pro programování aplikací zajišťuje možnost komunikace dvou různých systémů. Díky tomu je například umožněno, že více aplikací může využívat logiku jiného systému. Nejčastějšími styly implementace API jsou RPC, WSDL, SOAP a REST. [1, s. 2]

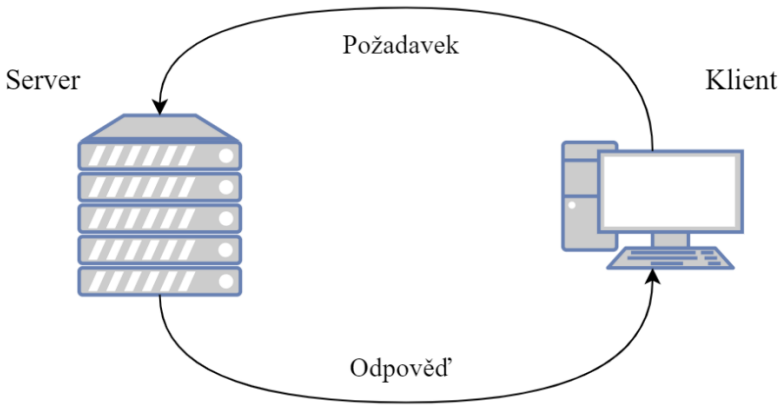
REST (Representational State Transfer) je koncept, který byl navržen v rámci disertační práce Roye Fieldinga v roce 2000. Aby bylo možné službu označit jako RESTful, je nutné, aby splňovala následující pravidla:

- architektura klient-server,
- bezstavovost (stateless),
- jednotné rozhraní (uniform interface),
- cache,
- vrstvený systém (layered system),
- code on demand. [1, s. 5]

3.1.1 Architektura klient-server

Architektura klient-server popisuje koncept, kde jsou server i klient oddělené entity, které mohou být navrženy a vyvíjeny nezávisle na sobě, za předpokladu, že rozhraní mezi nimi zůstane zachované. Dnes je nejsnazší setkat se s touto architekturou ve sféře webových aplikací. Klient zde představuje aplikaci běžící ve webovém prohlížeči starající se především o uživatelské rozhraní. Naproti tomu komponenta server je serverovou aplikací, ve které je definováno, jak jsou data zpracovávána a ukládána. [2, s. 12]

Komunikaci zahajuje klient tím, že na server pošle požadavek. Server tento požadavek zpracuje podle implementované logiky a poté pošle klientovi odpověď. Toto schéma je znázorněno na obrázku 1. [1, s. 6]



Obrázek 1 - Architektura klient-server

Vlastní tvorba dle zdroje: [1, s. 6]

3.1.2 **Bezstavovost**

Bezstavovost komunikace mezi serverem a klientem znamená, že požadavek musí obsahovat veškeré potřebné informace, aby mohl být zpracován. Na straně serveru tedy není uložen stav klienta. Díky této vlastnosti lze službu snáze škálovat. [3, s. 19]

3.1.3 **Jednotné rozhraní**

Princip jednotného rozhraní udává, že serverem vystavěné rozhraní je stejné pro jakéhokoliv klienta. Tomuto omezení je vyhověno při splnění čtyř dílčích podmínek.

- Identifikace zdroje – implementováno pomocí URI.
- Způsob manipulace se zdroji – realizováno využitím protokolu HTTP.
- Zpráva musí obsahovat veškeré informace nutné k tomu, aby ji bylo možné zpracovat.
- Vrácená reprezentace zdroje obsahuje informaci, jaké operace je možné se zdrojem provádět. [2, s. 14] [1, s. 6]

3.1.4 **Cache**

Dalším pravidlem je možnost u každého požadavku specifikovat, zda mají být dotazovaná data uložena do mezipaměti serveru. Pokud je tato funkce aktivována, pak jsou data při prvním požadavku načtena z databáze, ovšem při druhém požadavku na stejný zdroj

budou data načtena z mezipaměti. Toto pravidlo vede ke snížení prodlevy mezi požadavkem a odpovědí. [3, s. 19]

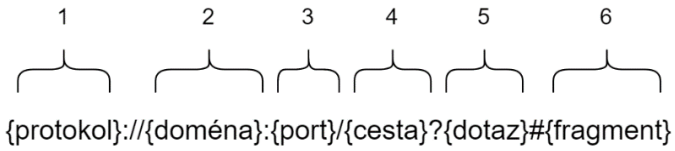
3.1.5 **Vrstvený systém**

Od RESTful služby je očekáváno, že je implementována jako hierarchicky vrstvený systém, kde každá komponenta má povědomí pouze o sousedních komponentách. Díky tomuto principu lze celý projekt snáze udržovat. [2, s. 18]

3.2 **URI**

Primárním předmětem komunikace mezi klientem a serverem je zdroj. Zdrojem se rozumí jakákoliv informace – může se jednat o dokument, fotografii nebo jiný formát. Každý zdroj je jednoznačně identifikován pomocí URI (Uniform Resource Identifier). Struktura URI je znázorněna na obrázku 2 a je rozdělena do těchto částí:

1. Protokol, který je využit.
2. Doména označující server, kterému bude požadavek zaslán.
3. Port může být vynechán v případě, že server využívá standardní porty pro daný protokol.
4. Cesta k danému zdroji.
5. Dotaz je seznam klíčů a hodnot. Může být použit například jako filtr vrácených dat.
6. Fragment specifikuje, že se jedná pouze o část zdroje. [4]



Obrázek 2 - Struktura URI

Vlastní tvorba dle zdroje: [4]

3.3 **HTTP**

Komunikace mezi klientem a serverem je realizována pomocí individuálních zpráv, které definuje přenosový protokol HTTP (Hypertext Transfer Protocol). [3, s. 23]

3.3.1 **Zpráva**

HTTP zpráva se skládá z hlavičky a těla. Tělo zprávy obsahuje předmět komunikace. V hlavičce jsou specifikovány dodatečné informace o zprávě. Může se jednat kupříkladu o údaje nutné k ověření klienta nebo k definování formátu dat v těle.

Zpráva posílána klientem se nazývá požadavek. Naproti tomu zpráva odeslaná ze serveru je odpověď. [5]

3.3.2 **Metody**

U požadavků je nutné specifikovat, jaká operace má být nad daným zdrojem provedena. To je realizováno pomocí následujících metod:

- GET,
- POST,
- PUT,
- DELETE.

Kromě těchto základních metod jsou protokolem dále definovány PATCH, OPTIONS, HEAD, TRACE a CONNECT. [6]

Metoda GET, jak už anglický název napovídá, slouží k získávání informací ze strany serveru. Tělo požadavku je v tomto případě prázdné. Pokud server dotazovaná data nenalezne, pak vrátí odpověď se stavovým kódem 404. V opačném případě jsou dotazovaná data posílána v těle odpovědi s kódem 200. [7, s. 53]

Vytvoření nového záznamu je realizováno pomocí metody POST. V těle požadavku jsou definována data, podle kterých se následně vytvoří nový záznam. Po úspěšném zpracování je serverem zaslána odpověď se stavovým kódem 201. V těle této zprávy jsou informace o vytvořeném zdroji. [7, s. 54]

K úpravě zdrojů je definována metoda PUT. V těle požadavku je zasílán záznam, který je považován za modifikaci již existujícího záznamu. Pokud není nalezen odpovídající záznam k modifikaci, je vytvořen nový záznam a vrácena stejná odpověď jako u metody POST. Pokud dojde k úspěšné úpravě, je vrácen stavový kód 204 bez těla zprávy. [7, s. 55]

Metoda DELETE umožňuje zdroje na straně serveru mazat. Stejně jako u metody GET je tělo požadavku prázdné. Při úspěšném vymazání záznamu odpovídá server se stavem 200 nebo 204. Pokud ovšem daný zdroj nenalezne, vrací kód 404. [7, s. 56]

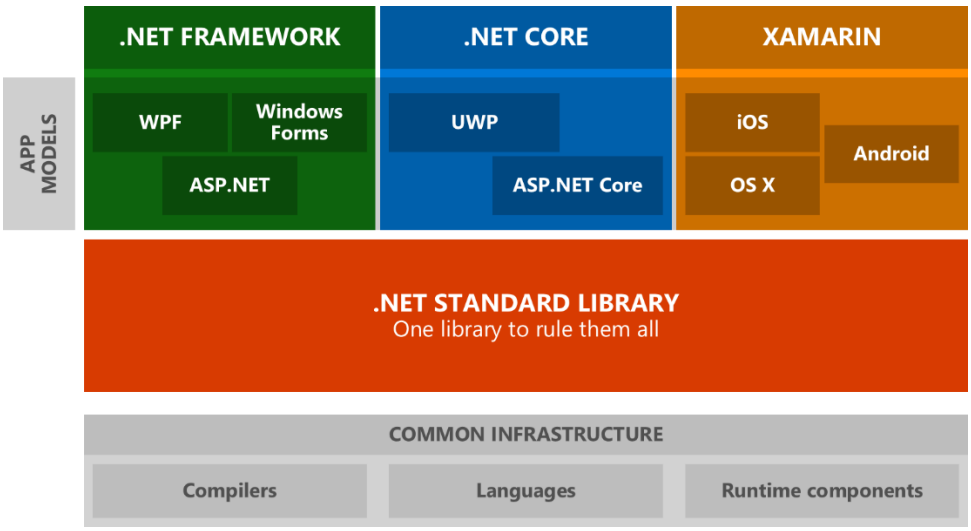
3.3.3 Stavové kódy

Stavový kód je tříciferné číslo sdělující výsledek požadavku. Kódy je možné rozdělit do pěti kategorií dle první cifry.

- 1xx – informační odpověď,
- 2xx – úspěch,
- 3xx – přesměrování,
- 4xx – chyba klienta,
- 5xx – chyba serveru. [7, s. 57]

3.4 .NET

Architektura .NET od firmy Microsoft je tvořena programovacími jazyky, standardní knihovnou a jednotlivými implementacemi této architektury. Standardní knihovna se nazývá .NET Standard a představuje sadu rozhraní API. Tato sada je následně implementována knihovnou tříd vývojářské platformy. Pokud je více vývojářských platforem v souladu se stejnou verzí .NET Standardu, pak je mezi nimi zajištěna přenositelnost kódu. Zmíněnými programovacími jazyky zastřešenými architekturou .NET jsou C#, F# a Visual Basic. [8]



Obrázek 3 - Schéma .NET

Zdroj: [9]

3.4.1 Implementace

Mezi implementace .NETu spadají vývojářské platformy .NET Framework, .NET Core a Mono. Každá z těchto platforem zahrnuje běhové prostředí, knihovnu tříd, případně i aplikační framework a vývojářské nástroje. [8]

.NET Framework je původní implementací konceptu .NET z roku 2002. Je optimalizována k vývoji desktopových aplikací pro operační systémy Windows a zároveň umožnuje tvorbu webových aplikací. Poslední verzí je .NET Framework 4.8. Další verze již vydány nebudou. [10]

V roce 2014 byl zahájen vývoj multiplatformního open-source nástupce .NET Frameworku, který byl pojmenován .NET Core. Současnou verzí je .NET Core 3.1, přičemž v následující verzi dojde ke změně názvu na .NET 5.0. [11, s. 9] V porovnání s jeho předchůdcem nabízí .NET Core zlepšení výkonu a redukci velikosti platformy. [11, s. 11]

Třetími stranami byla vytvořena implementace jménem Mono, která je taktéž multiplatformní. Mono dnes pohání například mobilní aplikace Xamarin. [11, s. 8]

3.4.2 C#

Moderní programovací jazyk C# může připomínat jazyky C, C++ nebo Java, neboť také patří mezi programovací jazyky rodiny C. Mimo jiné je tento jazyk standardizovaný organizacemi ECMA a ISO. [12]

Jedná se o objektově orientovaný jazyk, podporující koncepty zapouzdření, polymorfismu a dědičnosti. [13] C# dále nabízí funkce, které pomáhají s vývojem robustních aplikací. Patří mezi ně například automatické uvolňování paměti a zpracovávání výjimek. Jelikož je C# bezpečně typovaný, je zajištěna kontrola přesáhnutí hranice polí nebo nevalidní operace nad datovým typem. [12]

Zdrojový kód v jazyce C# je nejprve zkompilován do mezijazyka, v originále intermediate language (IL). Při spuštění je IL načten běhovým prostředím CLR, které jej pomocí just-in-time kompilace přeloží do instrukcí pro procesor. [11, s. 13]

3.4.3 ASP.NET Core

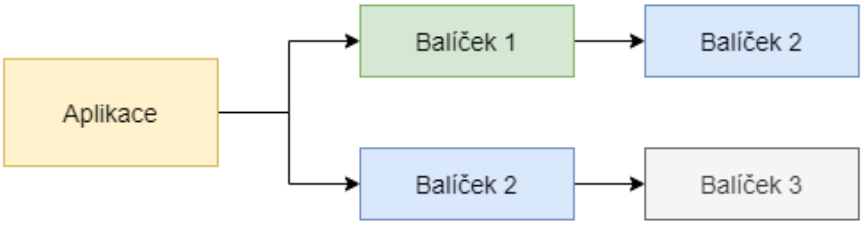
ASP.NET Core je open-source framework pro platformu .NET Core vydaný v roce 2016. Lze jej využít pro vytváření webových aplikací nebo jen jejich serverových částí. Na platformě .NET Framework byly pro tyto účely využívány ASP.NET MVC, ASP.NET Web API a ASP.NET SignalR. Oproti svému předchůdci poskytuje ASP.NET Core místo tří frameworků pouze jedno modulární řešení a stává se tím mnohem flexibilnější. [11, s. 482] [14]

3.4.4 NuGet

Při vývoji aplikací je možné setkat se s balíčkovacím systémem. Platforma .NET využívá balíčkovací systém zvaný NuGet. NuGet umožňuje vývojářům vytvářet, sdílet a využívat balíčky, které obsahují zkompilovaný užitečný kód. Balíček je možné zveřejnit na centrálním úložišti nuget.org, kde je již více než 100 000 balíčků. Ovšem je možné jej ukládat i v lokálním souborovém systému nebo na jiném privátním úložišti. [15]

NuGet dále poskytuje velmi důležitou funkci související s konzumací balíčků. Jedná se o správu závislostí. Sdílené balíčky nemusí nutně využívat jen aplikace, ale i balíčky samotné. Může tedy nastat následující situace znázorněná na obrázku 4.

Díky správě závislostí se vývojář aplikací musí starat pouze o napojení přímo využívaných balíčků 1 a 2, o zbytek se již postará NuGet. Zároveň sám vyřeší vícenásobnou závislost a nainstaluje daný balíček jen jednou. Problém může jít ještě hlouběji a vícenásobné závislosti mohou požadovat různé verze balíčků. I tehdy NuGet zajistí, že je nainstalována verze balíčku, která splní všechny podmínky. [16]



Obrázek 4 - Správa závislostí NuGet balíčků

Vlastní tvorba dle zdroje: [16]

3.4.5 Entity Framework Core

Entity Framework (EF) je populární implementace objektově-relačního mapování (ORM) pro .NET. Později byla rovněž vytvořena multiplatformní a modernější verze, a tou je právě Entity Framework Core (EF Core). Oproti starší verzi nabízí EF Core podporu i pro nerelační databázové systémy a je rozdělen do více balíčků. [11, s. 360]

ORM je technika, která programátorovi umožnuje pracovat s daty v databázi pouze pomocí objektů v programovacím jazyce, bez nutnosti znalosti SQL příkazů. EF Core pomocí přístupu Code-First umožňuje definovat tabulku v jazyce C# pomocí tříd, vlastností a anotačních atributů. [11, s. 366] Mimo jiné umožňuje přistupovat k různým databázím

pomocí knihoven nazývaných poskytovatelé. Zde je seznam vybraných databází, které jsou podporovány:

- Microsoft SQL,
- SQLite,
- PostgreSQL,
- MySQL,
- Oracle DB. [17]

3.5 Middleware

Příchozí HTTP požadavky se na serveru řadí do fronty, ve které jsou v daném pořadí specifikovány middlewary. Middleware je tedy funkce, která na kontext požadavku ve frontě může aplikovat určitou logiku. Zároveň může předat kontext ke zpracování dalšímu middlewaru, nebo naopak může být kontext navrácen zpět. Pomocí middlewarů může být realizován proces ověření, logování nebo směrování. Výhodou je přehledná struktura projektu a snadná rozšiřitelnost. Middlewary také odpovídají vrstvenému systému, který je jedním z pravidel REST API. [1, s. 49]

3.6 Microsoft SQL

Stejně jako MySQL nebo SQLite, tak i Microsoft SQL (MS SQL) je relačně databázový systém. Data jsou tedy ukládána v tabulkách neboli relacích. Pro sloupce v těchto tabulkách je pevně stanovený datový typ. Nelze tedy do sloupce vložit hodnotu jiného typu, než který byl určen. S databázovým systémem se standardně komunikuje pomocí jazyka SQL. V tomto případě je využit jazyk T-SQL (Transact Structured Query Language), který je rozšířením zmíněného SQL. [18]

3.7 Jednostránkové aplikace

Jednostránkové aplikace, v originále single-page application (SPA), jsou jednou z variant webových aplikací. Klientovi je v tomto případě odeslán jeden HTML dokument a při navigaci ve webové aplikaci se pouze mění obsah dokumentu. Komunikace mezi serverem a klientem je tedy omezena výhradně na výměnu dat, se kterými je pracováno. Tato skutečnost vede k rychlejšímu načítání dat, jelikož není nutné neustále čekat, až server odešle celou stránku včetně grafických doplňků. Zároveň však předpokládá vystavení webového API serverem, které mohou později využít i jiní klienti. [19]

3.8 JSON

Pro přenos komplexnější struktury dat již není dostačující zápis v běžném textu. JSON (JavaScript Object Notation) je tedy řešení pro formátování takových dat. S JavaScriptem má společnou syntaxi zápisu objektů a polí. Struktura JSON dokumentu je velmi dobře čitelná jak člověkem, tak i počítačem. Data jsou zapisována kombinací názvu a hodnoty. Název je vždy text zapsaný v uvozovkách. Hodnotou může být objekt zanořený ve složených závorkách, pole hodnot v hranatých závorkách či elementární datový typ. [20] JSON je standardizovaný organizací ECMA. [21]

3.9 Vue.js

Vue.js je open-source framework pro vytváření uživatelských rozhraní. Verze 1.0.0 byla zveřejněna v roce 2015 a autorem je Evan You. [22] Jádro frameworku umožňuje vytvářet jednoduché klientské aplikace, ovšem pomocí dalších knihoven je možné vyvíjet komplexnější projekty. Framework mimo standardní funkce, kterými jsou například podmínky, cykly a proměnné v HTML dokumentu, nabízí možnost sofistikovaného způsobu tvorby SPA. [23] [24]

3.9.1 Komponentový model

Základním konceptem ve Vue.js je komponentový model. Ten umožňuje vybudovat aplikace pomocí menších znovupoužitelných komponent nazývaných Vue. Ty jsou organizovány do stromové struktury čili komponenty mohou být sestaveny z jiných komponent. Mezi výhody tohoto konceptu patří zamezení duplicity kódu, snadná rozšiřitelnost a větší přehlednost při rozsáhlejších projektech. [24]

Komponenty se během svého životního cyklu nachází v několika stavech. V souvislosti s těmito stavy jsou vystaveny funkce umožňující reagovat na přechody mezi nimi. [24]

U komponent lze definovat data a metody. Data jsou proměnné, které jsou při vytvoření instance komponenty přidány do reaktivního systému. Automatické renderování změn těchto proměnných je zajištěno právě reaktivním systémem. Metody představují logiku, která se vztahuje přímo k dané komponentě. Podřazené komponentě je možné předávat data přes tzv. props, které je nutné v dané komponentě uvést, případně i jejich typ nebo povinnost. V opačném směru lze data předat rodičovské komponentě vyvoláním

vlastní události pomocí metody $emit. Nadřazená komponenta tuto událost zachycuje v atributu v-on. [24]

3.9.2 **Router**

Při vývoji SPA vypomůže knihovna vue-router. Ta umožňuje přistupovat k částem aplikace přes URL adresu stejně jako u tradičních webových aplikací. Tento princip je založen na objektu, ve kterém je k cestě přiřazena určitá Vue komponenta. Následně je tato komponenta renderována do elementu router-view podle současné URL adresy. [25]

4 Vlastní práce

V této kapitole bude popsána analýza a implementace řešení pro sjednávání studentských prací, které je autorem pojmenováno jako *Studentby*. K tomu budou využity teoretické základy popsané v kapitole 3 a ostatní znalosti, kterých autor nabyl během studia.

4.1 Analýza

Prvním krokem v praktické části práce je analýza. V této fázi budou stanoveny požadavky, které se dělí na funkční a nefunkční. Úkony, které aplikace bude umožňovat se řadí mezi požadavky funkční. Ostatní požadavky lze zařadit mezi nefunkční. Těmi mohou být například specifikace implementace. Aplikace je vytvářena pouze v rámci této práce a žádná další strana do vývoje není zapojena. Vzhledem k této skutečnosti budou požadavky určeny pouze autorem. Na základě stanovených požadavků bude následně vytvořen wireframe a datový model.

4.1.1 Funkční požadavky

4.1.1.1 R1.0 – Uživatelské účty

Všechny uživatelské účty v aplikaci mají přiřazené heslo a unikátní emailovou adresu. Celkem jsou definovány čtyři uživatelské role – neaktivovaný student, aktivovaný student, zákazník a operátor.

- Operátor řídí zpracování žádostí o pracovní nabídku a procesy související s realizací pracovní nabídky. Dále spravuje uživatele s ostatními rolemi.
- Neaktivovaný student je výchozí role po registraci studentského účtu. Může provádět operace, které neovlivní ostatní uživatele.
- Po aktivaci studentského účtu je přiřazena role aktivovaný student. S touto rolí je již možné ucházet se o pracovní nabídky.
- Zákazník představuje uživatele, který vytváří pracovní nabídky pod jménem příslušné pracovní skupiny.

4.1.1.1.1 R1.1 – Registrace studenta

Pro veřejnost je možné vytvářet pouze účet s rolí neaktivovaného studenta. Při registraci je nutné uvést následující dodatečné informace: jméno, příjmení, místo bydliště a datum narození.

4.1.1.1.2 R1.2 – Registrace zákazníka

Vytváření účtu zákazníka je v kompetenci operátora. Každý zákaznický účet musí být přiřazen do již existující skupiny.

4.1.1.1.3 R1.3 – Přihlášení

Pro využívání aplikace je nezbytné přihlášení k uživatelskému účtu pomocí emailové adresy a hesla.

4.1.1.1.4 R1.4 – Aktivace a deaktivace studentského účtu

Operátor má možnost měnit roli studentského účtu na aktivovaný a neaktivovaný. S deaktivací musí být provedeno i zrušení nastávajících aktivit příslušících danému účtu.

4.1.1.2 R2.0 – Skupiny

Zákazníci jsou organizováni do skupin. Tato struktura umožňuje více uživatelům veřejně vystupovat pod názvem jedné společnosti. Zároveň to ale znamená, že jeden uživatelský účet odpovídá pouze jedné reálné osobě.

4.1.1.2.1 R2.1 – Správa skupin

Nové skupiny zákazníků smí vytvářet pouze operátor.

4.1.1.3 R3.0 – Pracovní nabídka

Hlavním předmětem aplikace je nabídka studentské práce. Pro nabídku musí být specifikován titulek, podrobný popis, termín, kapacita míst, místo konání a hodinová sazba.

4.1.1.3.1 R3.1 – Vytvoření nabídky

Při vytváření bude k nabídce automaticky přiřazena skupina dle uživatele, který ji vytvořil. Tato funkce je povolená pouze pro zákazníka.

4.1.1.3.2 R3.2 – Prohlížení nabídek

Všem uživatelským rolím je umožněno prohlížet si pracovní nabídky. Pro roli zákazníka je ovšem tato množina zúžena pouze na nabídky, které náleží jeho skupině.

4.1.1.4 R4.0 – Přihláška k pracovní nabídce

Vztah mezi pracovní nabídkou a studentem je reprezentován přihláškou, která nabývá jednoho z pěti stavů. Nezpracováno, zamítnuto, přijato, absence a účast.

4.1.1.4.1 R4.1 – Vytvoření přihlášky

Novou přihlášku k pracovní nabídce může podat pouze uživatel s aktivovanou studentskou rolí. Při tomto procesu musí být splněna následující pravidla:

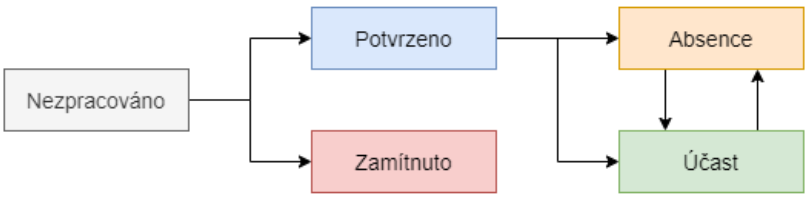
- Kapacita pracovní nabídky není vyčerpána.
- Termín se nepřekrývá s termínem jiné nabídky, na kterou student již podal přihlášku.
- Počáteční termín nabídky nesmí být v minulosti.

4.1.1.4.2 R4.2 – Zrušení přihlášky

Přihláška může být studentem zrušena pouze v případě, že je stále ve stavu nezpracováno.

4.1.1.4.3 R4.3 – Přechod mezi stavy

Přechod mezi stavy přihlášky řídí operátor a pravidla podléhají stanovenému schématu na obrázku 5.



Obrázek 5 - Přechod mezi stavy přihlášky

4.1.2 **Nefunkční požadavky**

4.1.2.1 R5.0 – Implementace

Serverová část aplikace bude implementována jako webové API za pomocí frameworku ASP.NET Core 3.

4.1.2.2 R6.0 – Ukládání dat

Pro ukládání dat bude využit databázový systém Microsoft SQL.

4.1.3 Pojmenování

Při programování celého řešení budou proměnné, třídy i metody pojmenovávány v anglickém jazyce, aby nedocházelo k tak častému míchání českého a anglického jazyka. Dává tedy smysl, aby s touto konvencí bylo počítáno již v další fázi analýzy. Tabulka 1 je slovníkem vybraných pojmů z analýzy požadavků.

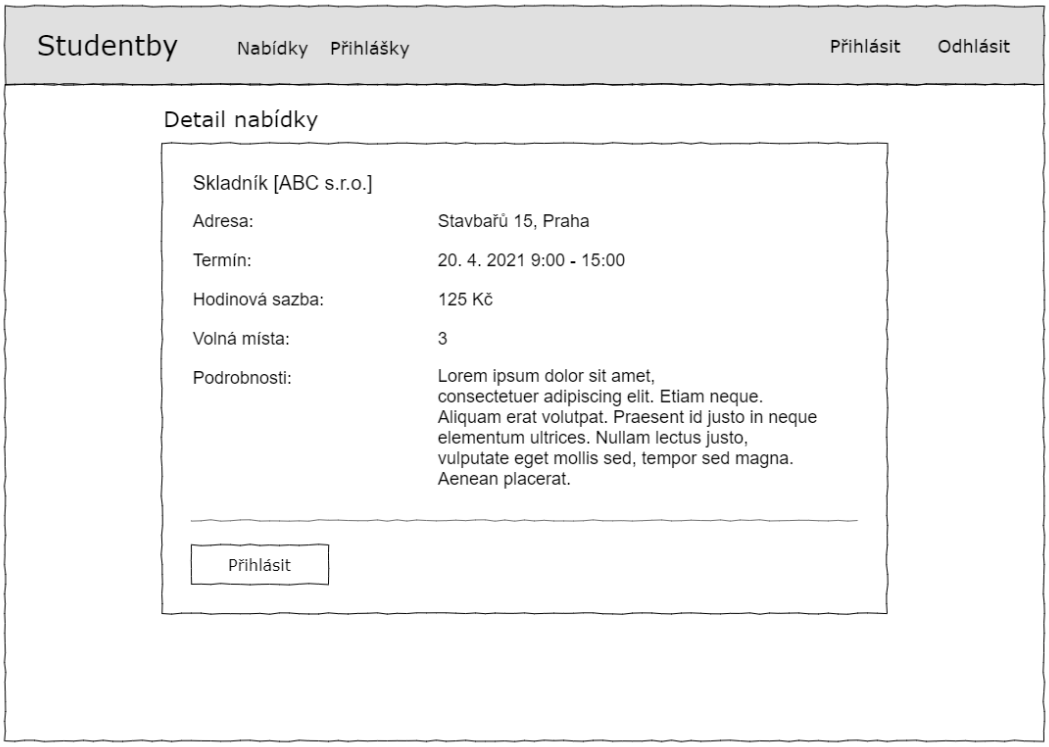
pojem v českém jazyce	**pojem v anglickém jazyce**
zákazník (R1.0**)**	customer
skupina zákazníků (R2.0)	group
pracovní nabídka (R3.0)	job offer
přihláška k pracovní nabídce (R4.0)	job application

Tabulka 1 - Pojmenování

4.1.4 Wireframe

Součástí analýzy je i vytvoření drátových modelů (wireframů). Ty slouží především k definici rozmístění prvků na stránkách aplikace. Vzhledem k množství wireframů bude v této práci uvedeno pouze zobrazení detailu pracovní nabídky z pohledu studenta.

Všechny stránky budou mít společný navigační panel. Zobrazené položky na tomto panelu závisí na roli uživatele, který je přihlášen. Pro studenta jsou k dispozici navigační tlačítka *Nabídky* a *Přihlášky*. Na panelu se dále nachází název aplikace a možnost přihlášení, nebo odhlášení. Stránka s detailem nabídky obsahuje název pracovní nabídky, název skupiny, termín, hodinovou sazbu, lokaci, počet volných míst a podrobný popis. Dalším důležitým prvkem na této stránce je tlačítko přihlášení k pracovní nabídce, které vytvoří přihlášku, pokud nebyla porušena žádná omezení. Popsaný wireframe je na obrázku 6.



Obrázek 6 - Wireframe detailu pracovní nabídky z pohledu studenta

4.1.5 **Datový model**

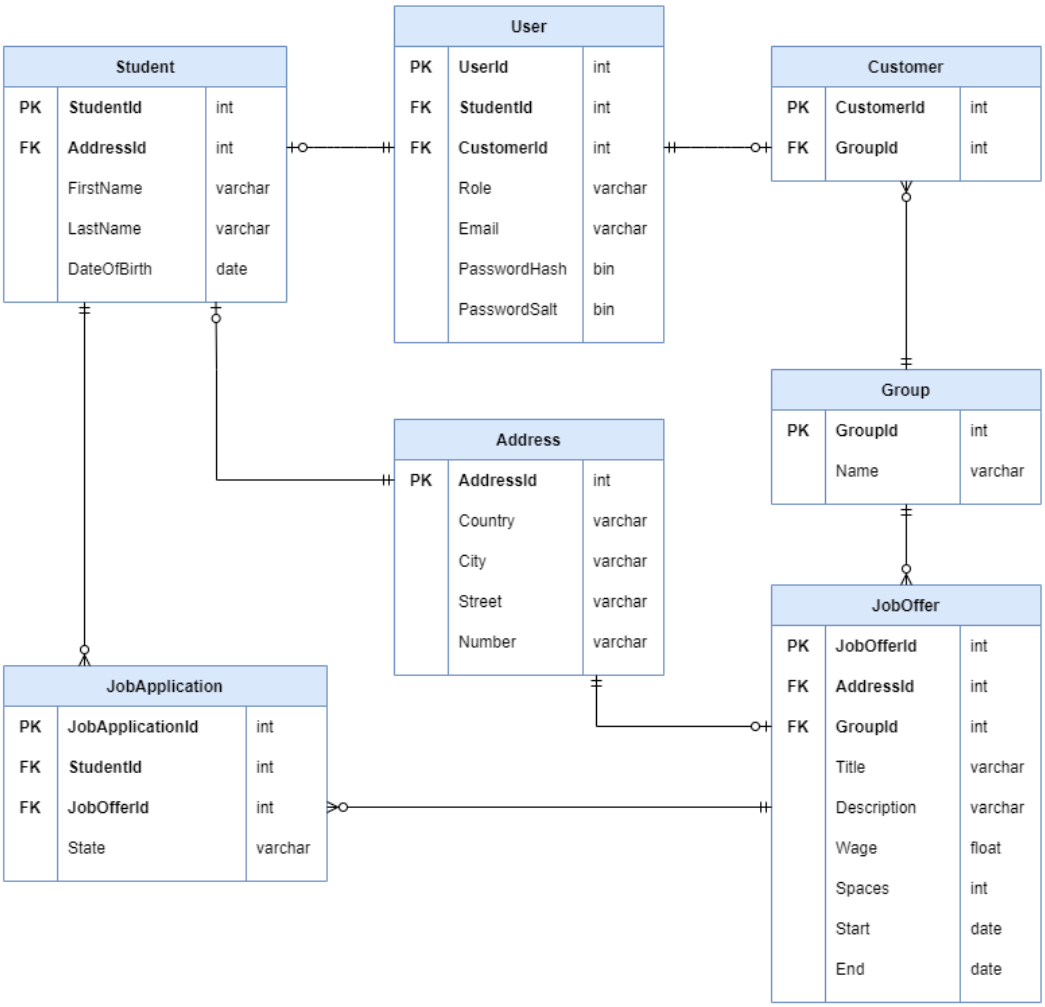
Dalším úkonem před zahájením vývoje aplikace je návrh datového modelu na základě přechozích kroků analýzy. Datový návrh určí strukturu tabulek a vazby mezi nimi. Této fázi je vhodné věnovat dostatečnou pozornost, neboť změny datového modelu v průběhu vývoje mohou znamenat značné úpravy zdrojového kódu, v horším případě i kompletní přeprogramování některých funkcí.

4.1.5.1 Popis modelu

V datovém modelu na obrázku 7 je stanovena struktura jednotlivých tabulek a jsou zvýrazněny primární a cizí klíče. Dále je graficky označena kardinalita a povinnost vazeb mezi nimi. Zde je nutné podotknout, že na datový model nebyla aplikována datová normalizace v plné míře. Především role uživatele a stav přihlášky by správně měly vytvořit další dvě tabulky. Pro snazší manipulaci s daty byla ovšem zvolena struktura zobrazena na obrázku 7.

Pro realizaci uživatelských rolí byly vytvořeny dvě tabulky *Student* a *Customer*. Vazby uživatele na tyto tabulky nejsou povinné. Pouze při nastavení uživatelské role zákazníka nebo studenta vznikne vazba na příslušnou tabulku. Vztah byl vymodelován tímto

způsobem, aby bylo zřejmé, že vazba na tabulku přihlášek je relevantní pouze pro studenta, nikoli pro zákazníka.



Obrázek 7 - Datový model

4.2 Implementace

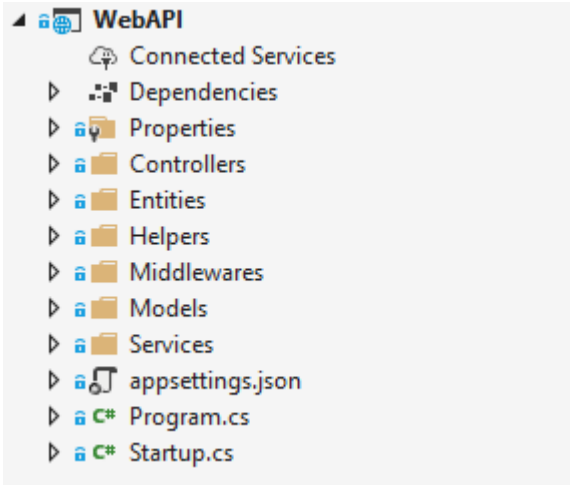
Implementace celého řešení je rozdělena na dvě části. První část je zaměřena na vývoj serverové aplikace (backendu), která bude vystavovat webové API. Ve druhé části bude popsána tvorba ukázkové klientské aplikace (frontendu), která bude zmíněné webové API využívat.

4.2.1 Backend

K vývoji backendu poslouží robustní vývojové prostředí Visual Studio 2019 od firmy Microsoft v edici Community. Výběrem vestavěné šablony ASP.NET Core Web API při

zakládání nového projektu je vytvořena výchozí adresářová hierarchie. Rozšířením této struktury o další složky je dosaženo přehlednější organizace souborů potřebných pro tvorbu aplikace. Vzniklá adresářová struktura je na obrázku 8.

Složka Controllers byla vytvořena dle výchozího nastavení šablony. Mimo odstranění jejího ukázkového obsahu není nutné tuto složku nijak měnit. Dále budou zmíněny autorem vytvořené adresáře. Složka Entities bude obsahovat veškerý kód související s definicí datového modelu pro Entity Framework Core. Podobný obsah bude mít i složka Models. Pod ní se sdružují definice objektů pro přenos dat. Dále je zde složka Services pro aplikační logiku. Veškeré middleware funkce jsou zařazeny pod stejnojmenný adresář. K organizaci ostatních souborů existuje adresář Helpers.



Obrázek 8 - Adresářová struktura backendu

4.2.1.1 Datová vrstva

Datový model navrhnutý v kapitole 4.1.5 je nyní nutné implementovat. K tomu bude využit EF Core a přístup Code-First. Každé definované tabulce bude odpovídat jedna třída.

Příkladem takové třídy je JobOffer ve zdrojovém kódu 1. První vlastností této třídy je primární klíč JobOfferId. EF Core nevyžaduje žádný atribut označující, že se opravdu jedná o primární klíč, rozpozná to automaticky podle výskytu slova Id v názvu. Následuje definice dalších sloupců. Titulek a popis jsou povinné hodnoty. Je nutné využít atribut Required, jelikož string je nullovatelný datový typ. Vazba na jinou tabulku je zde realizována pomocí dvou vlastností – AddressId a Address. V prvním případě se jedná o cizí klíč do tabulky s adresami. Z této strany je vazba povinná, jelikož integer nemůže nabývat hodnoty null. Address je navigační vlastností, která udržuje referenci na navázanou entitu. Navigační vlastností je rovněž JobApplications. Skutečnost, že se zároveň jedná i o kolekci, znamená

kardinalitu vztahu 1:N. Navigační vlastnosti budou velmi užitečné při pozdější manipulaci s objekty.

```
[Table("JobOffer")]
public class JobOffer
{
    public int JobOfferId { get; set; }

    [Required]
    public string Title { get; set; }
    [Required]
    public string Description { get; set; }
    public double Wage { get; set; }
    public int Spaces { get; set; }
    public DateTime Start { get; set; }
    public DateTime End { get; set; }

    public int AddressId { get; set; }
    public int GroupId { get; set; }

    public Address Address { get; set; }
    public Group Group { get; set; }
    public ICollection<JobApplication> JobApplications { get; set; }
}
```

Zdrojový kód 1

Dalším krokem je sjednocení definicí pro EF Core. K tomu poslouží třída StudentbyContext, která dědí ze třídy DbContext. V této chvíli je nutné nainstalovat balíček EntityFrameworkCore pomocí systému NuGet popsaném v kapitole 3.4.4.

Pro každou tabulku, která má být vytvořena, musí být v kontextové třídě uveden DbSet. Pomocí této vlastnosti bude později k příslušné tabulce přistupováno. Ve zdrojovém kódu 2 je uvedení třídy User ve třídě StudentbyContext.

```
public DbSet<User> Users { get; set; }
```

Zdrojový kód 2

Zdrojový kód 3 ve třídě Startup zajistí připojení k databázi dle připojovacího řetězce, který je specifikován v konfiguračním souboru appsettings.json. Dle požadavku R6.0 je

využit Microsoft SQL Server. Pro připojení je nepostradatelný příslušný poskytovatel v podobě NuGet balíčku. Po spuštěním příkazů *Add-Migration* a *Update-Database* se v databázi vytvoří tabulky dle definice ve třídě StudentbyContext.

```
services.AddDbContext<StudentbyContext>(options =>
{
    string connectionString = Configuration.GetConnectionString("Default");
    options.UseSqlServer(connectionString);
});
```

Zdrojový kód 3

4.2.1.2 Přenos dat

Složka Models je obsahem velmi podobná složce Entities z předchozí kapitoly. Nejedná se ovšem o duplicitu, jelikož jsou v ní seskupeny objekty pro přenos dat, v originále data transfer object (DTO). DTO je v projektu prostá třída a je pro vystavené webové API velmi důležitá, neboť přesně definuje strukturu dat, které server bude zasílat, nebo naopak přijímat.

Pochopitelně by se v některých případech dalo namítat, proč je DTO vůbec nutný. Pokud má být v odpovědi vrácena pracovní nabídka, tak může být jednoduše vrácena instance třídy Entities.JobOffer, která definuje příslušnou tabulku v databázi. Může ovšem nastat situace, že nebude chtěné zveřejnit klientovi veškeré vlastnosti objektu. Jako příklad lze vzít entitu uživatele. Rozhodně není žádoucí zasílat v odpovědi hash hesla uživatele, které třída Entities.User definuje. Tento argument pro využití DTO stále nemusí být dostatečný, jelikož vlastnost třídy může být označena tak, aby nebyla při serializaci brána v potaz.

Avšak využití DTO nabízí více výhod než jen skrytí některých vlastností. Umožňuje rozšířit strukturu objektu o další související informace. V případě pracovní nabídky může být kupříkladu požadováno, aby byl do odpovědi zahrnut i počet aktivních přihlášek. Dále DTO nabízí možnost pro více koncových bodů poskytnout stejnou entitu, ale jinak strukturovanou. Objekt pro přenos dat by ale mohl být využit i ve chvíli, kdy požadovaná struktura pro přenos odpovídá třídě v adresáři Entities. V tomto případě má DTO výhodu, že při pozdější možné změně definice tabulky není webové rozhraní nijak narušeno.

Z výše uvedených důvodů budou pro definici využity pouze objekty pro přenos dat. Třídy v adresáři Entities budou sloužit výhradně pro práci s databází. Vzhledem k tomu, že při dalším vývoji aplikace může počet objektů pro přenos dat značně narůst, je nutné zajistit přehlednost. Autorem byly tedy stanoveny následující konvence:

- Všechny třídy související se stejnou entitou budou v jednom souboru.
- Název třídy napovídá o struktuře objektu pomocí zkratek.
- V názvu třídy je uvedeno, zda se jedná o definici odpovědi či požadavku.
- Třídu lze využít i pro více koncových bodů.

Zdrojový kód 4 představuje DTO související s přihláškou k pracovní nabídce. Kromě standardního identifikátoru a stavu, budou do odpovědi zahrnuty údaje o studentovi. Vlastnost Student je také objekt pro přenos dat, který je uveden ve zdrojovém kódu 5.

```
public class JobApplicationMinWithStudRes
{
    public int JobApplicationId { get; set; }
    public string State { get; set; }
    public StudentMinRes Student { get; set; }
}
```

Zdrojový kód 4

```
public class StudentMinRes
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Zdrojový kód 5

4.2.1.3 Spuštění aplikace

Vstupním bodem do aplikace je standardně metoda Main ve třídě Program. Dochází zde k vytvoření HTTP serveru a k jeho spuštění. S vytvářením serveru souvisí i třída Startup, ve které se nachází metody ConfigureServices a Configure. První ze zmíněných metod se stará o registraci služeb využívaných v aplikaci. Právě zde bylo specifikováno připojení k databázi. Metoda Configure řeší přidávání middlewarů pro zpracování požadavků.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseAuthorization();
}
```

Zdrojový kód 6

4.2.1.4 Koncové body

Aby byla aplikace realizována jako webové API je nezbytné vystavit koncové body (endpointy), které představují určitou operaci nad entitou. Koncové body jsou v ASP.NET

Core definovány pomocí metod. Ty jsou pro lepší přehlednost organizovány v adresáři Controllers do tříd, které jsou potomky třídy ControllerBase.

Ve zdrojovém kódu 7 je vystaven endpoint GET /api/groups pomocí metody GetAll. Metoda je označena atributem HttpGet, který určuje, že se jedná právě o HTTP metodu GET. Návratový typ metody je právě zmíněný DTO, jenž je zanořen v generických typech. IEnumerable značí, že se jedná o kolekci, ActionResult je návratovým typem metod pro koncové body a jelikož se jedná o asynchronní metodu, tak je tento celek zanořen v typu Task.

```
[Route("api/groups")]
[ApiController]
public class GroupController : ControllerBase
{
    private readonly IGroupService _groupService;

    public GroupController(IGroupService groupService)
    {
        _groupService = groupService;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<GroupRes>>> GetAll()
    {
        var response = await _groupService.GetListAsync();
        return StatusCode(200, response);
    }
}
```

Zdrojový kód 7

4.2.1.5 Aplikační logika

Kvůli přehlednosti zdrojového kódu neřeší aplikační logiku žádná z metod ve třídě kontroleru. Pro zpracování dat dle aplikační logiky jsou využívány tzv. služby ve složce Services. Stejně jako kontrolery, tak i služby jsou rozděleny do tříd především podle entit, ke kterým se vztahují. Po zpracování vrací služba kontroleru data, dle kterých je vygenerována HTTP odpověď.

V případě nastání chyby v aplikační logice je vyvolána výjimka typu AppLogicException, která je ošetřena v middlewaru ExceptionHandler. Ten přesměruje požadavek na /api/error. Příslušný koncový bod na základě typu výjimky odešle odpověď. Ve zdrojovém kódu 8 je uvedena definice zmíněného koncového bodu.

```
[Route("error")]
public ErrorRes Error()
{
    var exception = HttpContext.Features.Get<IExceptionHandlerFeature>().Error;

    if (exception is AppLogicException)
    {
        Response.StatusCode = 400;
        string message = exception.Message;
        string detail = ((AppLogicException)exception).Detail;
        return new ErrorRes(message, detail);
    }
    else
    {
        Response.StatusCode = 500;
        string message = "Objevila se chyba na serveru";
        return new ErrorRes(message);
    }
}
```

Zdrojový kód 8

4.2.1.6 Validace dat

Při zpracování požadavků, které zasílají data v těle zprávy, může nastat chyba spočívající v předání nevalidních dat. Samotnou strukturu zasílaného objektu framework ošetří automaticky, jelikož se jej snaží převést na DTO uvedený v hlavičce metody, která definuje koncový bod. Jsou ale případy, kdy tato validace nestačí. Kupříkladu může být chtěné ověřit, zda textový řetězec odpovídá platnému formátu emailové adresy, nebo zda číslo náleží určitému intervalu. Zdrojový kód, který ověřuje tyto podmínky, by neměl být součástí služeb, jelikož by aplikační logiku učinil nepřehlednou.

Opět je možné využít DTO v kombinaci s validačními atributy vztahujícími se k jednotlivým vlastnostem objektu. Zde je výčet několika takových atributů, které jsou v ASP.NET Core připraveny k použití:

- Required – hodnota nesmí být null,
- Range – hodnota náleží danému intervalu,
- RegularExpression – hodnota je instancí uvedenému regulárního výrazu,
- EmailAddress – hodnota má formát validní emailové adresy.

Jestliže neexistuje atribut, který by požadovanou validaci zajistil, pak je možné vytvořit vlastní. Důležité je, aby třída reprezentující nový atribut byla potomkem třídy ValidationAttribute. Samotná logika validace je poté zapsána do metody IsValid.

V ukázce zdrojového kódu 9 je validace, která zajišťuje minimální počet uplynulých let od určitého data. V aplikaci je použita například pro ověření, že registrovaný student je

již plnoletý. Pokud nebude podmínka splněna je serverem navrácena odpověď s informacemi o nedostatcích. Avšak je žádoucí, aby struktura chybových odpovědí byla konzistentní, jelikož je tím usnadněno zpracování zpráv na straně klienta. Po přidání zdrojového kódu 10 do metody ConfigureServices ve třídě Startup je tento požadavek splněn. Veškeré validační chyby jsou zde sjednoceny do textového řetězce, který je následně předán konstruktoru rozšířené výjimky. Tato výjimka je následně zachycena příslušným middlewarem, zmíněným v části 4.2.1.5.

```
protected override ValidationResult IsValid(object value,
   ValidationContext validationContext)
{
    if (value != null)
    {
        DateTime birthDate = (DateTime)value;
        if (birthDate.AddYears(_minimumAge) < DateTime.Now)
        {
            return ValidationResult.Success;
        }
    }
    return new ValidationResult($"Věk musí být nejméně {_minimumAge}");
}
```

Zdrojový kód 9

```
services.AddControllers()
    .ConfigureApiBehaviorOptions(options =>
    {
        options.InvalidModelStateResponseFactory = context =>
        {
            var errors = string.Join(" ", context.ModelState.Values
               .Where(v => v.Errors.Count > 0)
               .SelectMany(v => v.Errors)
               .Select(v => v.ErrorMessage));

            throw new AppLogicException("Chyba při validaci požadavku", errors);
        };

    });
```

Zdrojový kód 10

4.2.1.7 Zabezpečení

Aby mohla být splněna většina funkčních požadavků, musí aplikace implementovat autentifikaci a autorizaci. Ověření uživatele bude řešeno využitím přístupového tokenu, konkrétně JSON Web Token (JWT). Celý proces lze stručně popsat následujícími kroky.

- Klient pošle serveru přihlašovací údaje uživatele.
- Server tyto údaje ověří proti databázi a v případě shody vrátí přístupový token.
- Klient tento token posílá v hlavičce požadavku na zabezpečený zdroj.

- Server na základě tokenu vyhodnotí, zda má uživatel na daný zdroj oprávnění.

Prvním krokem je napsání metody, která bude řešit vytvoření nového uživatele v databázi – tedy uložení jeho emailu, hesla a role. V databázi nemohou být hesla v prosté čitelné formě, neboť by je mohl znát kdokoliv s přístupem do databáze. Pokud by byl uložen pouze hash hesla, bylo by toto riziko eliminováno, ovšem pro obvyklá hesla může být hash již předpočítán. I toto riziko lze obejít, pokud před použitím hashovací funkce přidáme k heslu sůl (v originále salt). Tím pravděpodobnost, že byl hash takového hesla již přepočítán, značně klesá. Metoda by rovněž měla zajistit, že daný email není v databázi už registrován. Výsledek lze vidět ve zdrojovém kódu 11.

```
public async Task<User> CreateAsync(string email, string password, string role)
{
    bool userExists = await _context.Users.AnyAsync(x => x.Email == email);
    if (userExists)
    {
        throw new StudentbyException("Uživatel již existuje");
    }

    byte[] hash, salt;
    HashPassword(password, out hash, out salt);

    var newUser = new User
    {
        Email = email,
        PasswordHash = hash,
        PasswordSalt = salt,
        Role = role
    };
    return newUser;
}
private void HashPassword(string password, out byte[] hash, out byte[] salt)
{
    using (var hmac = new System.Security.Cryptography.HMACSHA256())
    {
        salt = hmac.Key;
        hash = hmac.ComputeHash(Encoding.UTF8.GetBytes(password));
    }
}
```

Zdrojový kód 11

Dále je nutné vystavit koncové body, které výše uvedenou metodu budou volat. Registrace studenta a zákazníka, ovšem nemůže být řešena stejným koncovým bodem. Důvodem je především skutečnost, že registrace studentského účtu je veřejná, kdežto zákazníka smí dle požadavku R1.2 registrovat pouze operátor. Navíc jsou ke každé roli navázána jiná data, která musí být při registraci rovněž uložena. U studenta se jedná například o osobní údaje a u zákazníka o příslušnost ke skupině.

U definice koncového bodu je vhodné zmínit především parametr DTO, který je uvozen atributem FromBody. Ten značí, že objekt bude získán z těla HTTP požadavku. Volaná metoda aplikační logiky na základě přijatého objektu vytvoří všechny potřebné entity. K tomu využije například i metodu pro vytvoření uživatele.

```
[AllowAnonymous]
[HttpPost]
public async Task<ActionResult<StudentRes>> Post([FromBody] StudentReq request)
{
    var response = await _studentService.CreateAsync(request);
    return StatusCode(201, response);

}
```

Zdrojový kód 12

Třetím krokem je vystavení koncového bodu POST /api/login, jehož logika ověří přihlašovací údaje v těle požadavku vůči databázi a v případě shody vygeneruje a vrátí nový JWT. V databázi bude nejprve hledán uživatel, kterému přísluší zaslaný email. V případě, že takový uživatel existuje, je nutné ověřit i heslo. To je docíleno vypočítáním hashe ze zaslaného hesla a uložené soli. Následně je porovnán uložený a vypočítaný hash, v případě shody lze vygenerovat token.

Token je generován pomocí NuGet balíčku System.IdentityModel.Tokens.Jwt. Díky uložení identifikace uživatele a jeho role v token může aplikace při ověření rozhodnout, zda má klient k danému zdroji povolen přístup. Životnost tokenu je nastavena na jednu hodinu pomocí vlastnosti Expires. Celá metoda pro generování JWT je ve zdrojovém kódu 13.

```
private string GenerateJwt(User user)
{
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.ASCII.GetBytes(_appSettings.Secret);
    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new Claim[]
        {
            new Claim(ClaimTypes.Name, user.UserId.ToString()),
            new Claim(ClaimTypes.Role, user.Role)
        }),
        Expires = DateTime.UtcNow.AddHours(1),
        SigningCredentials = new SigningCredentials(new
           SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
    };
    var token = tokenHandler.CreateToken(tokenDescriptor);
    return tokenHandler.WriteToken(token);
}
```

Zdrojový kód 13

V poslední řadě musí být celý proces ověření na serveru nakonfigurován. Je nutné nainstalovat NuGet balíček Microsoft.AspNetCore.Authentication.JwtBearer. Zdrojový kód 14 ve třídě Startup určuje, že autentifikace bude probíhat pomocí JWT. Důležité je, aby byl podepisovací klíč pro konfiguraci a pro vytváření tokenu shodný.

```
services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
    .AddJwtBearer(x =>
    {
        x.RequireHttpsMetadata = false;
        x.SaveToken = true;
        x.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(key),
            ValidateIssuer = false,
            ValidateAudience = false,
            ClockSkew = TimeSpan.Zero
        };
    });
```

Zdrojový kód 14

Po přidání middlewarů UseAuthentication a UseAuthorization je možné zabezpečovat koncové body pomocí atributu Authorize podobně jako ve zdrojovém kódu 15.

```
[Authorize(Roles = UserRoles.Operator)]
```

Zdrojový kód 15

4.2.1.8 Logování

Ověřeným postupem při vývoji aplikací je vést záznamy o událostech. Takovými událostmi nemusí být nutně jen chyby. V případě webového API se může jednat také o zaznamenávání přijatých požadavků a navrácených odpovědí. Tyto záznamy mohou být užitečné například pro diagnostikování chyb. Pro realizaci byl zvolen projekt Serilog, který je dostupný jako NuGet balíček. Konfigurace logování je dle dokumentace Serilogu provedena ve třídě Program.

Ve zdrojovém kódu 16 je nastaveno zaznamenávání do souborů. Parametrem rollingInterval je určeno, že pro každý den vznikne nový soubor. Zároveň lze definovat formát zaznamenané zprávy pomocí outputTemplate.

```
.WriteTo.File("./logs/log.txt", rollingInterval: RollingInterval.Day,
   retainedFileCountLimit: null,
   outputTemplate: "[{Level:u4} {Timestamp:HH:mm:ss} ] {Message:lj}{NewLine}")
```

Zdrojový kód 16

Logování jednotlivých HTTP požadavků lze zajistit přidáním dalšího middlewaru, který zaznamená každý požadavek a stavový kód, který server vrátil. Metoda ve zdrojovém kódu 17 vytvoří záznam s úrovní závažností „informační“. V případě logování chyby by byla využita její obdoba LogError.

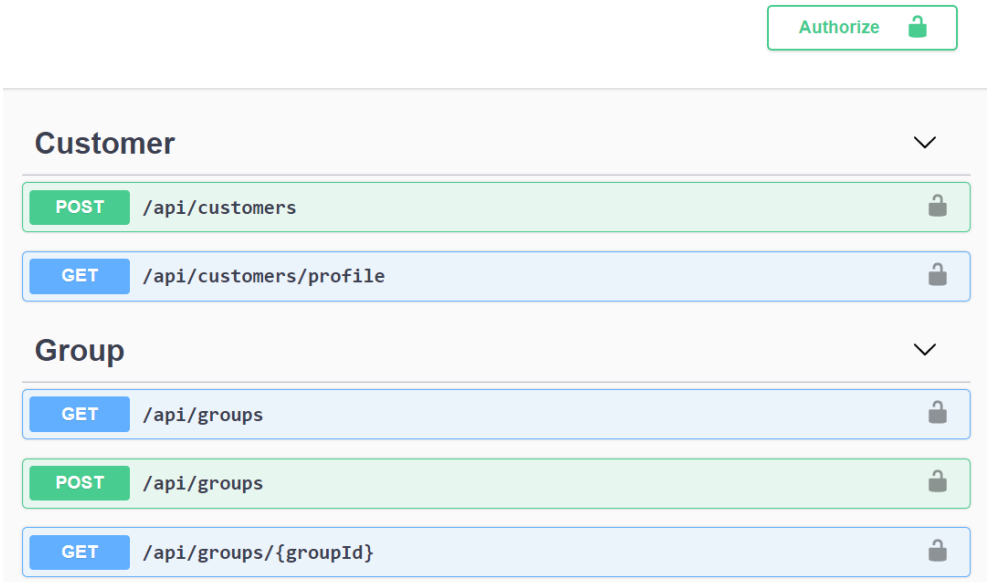
```
_logger.LogInformation(message);
```

Zdrojový kód 17

4.2.1.9 Swagger

Jedním z cílů této práce je vytvoření dokumentace vystaveného webového API. Ta může být užitečná v případě, že by mělo být API využíváno třetí stranou. Pro splnění tohoto cíle je využit NuGet balíček Swashbuckle.AspNetCore, který zajišťuje vytvoření dokumentace v podobě interaktivního rozhraní Swagger UI. Vytvoření dokumentace probíhá automaticky na základě zdrojového kódu koncových bodů a objektů pro přenos dat.

Ve třídě Startup je provedeno nastavení nástroje Swashbuckle. Kromě základních údajů je pro tento projekt specifikováno, že API využívá zabezpečení Bearer tokenem. Na obrázku 9 je ukázka vygenerované dokumentace.



Obrázek 9 – Dokumentace webového API

4.2.1.10 Zrušení přihlášky k pracovní nabídce

V této části je popsána implementace zrušení přihlášky k pracovní nabídce dle funkčního požadavku R4.2. Nezbytná je definice koncového bodu ve třídě JobApplicationController. Ve zdrojovém kódu 18 je uvedena příslušná metoda, označená atributem HttpDelete, jelikož bude zdroj odstraněn.

```
[HttpDelete("{jobApplicationId}")]
public async Task<ActionResult> Delete([FromRoute] int jobApplicationId)
{
    bool found = await _jobApplicationService.DeleteAsync(jobApplicationId);
    if (!found)
    {
        return StatusCode(404);
    }
    return StatusCode(204);
}
```

Zdrojový kód 18

Jak již bylo zmíněno v části 4.2.1.5, tak aplikační logiku obstarává služba, v tomto případě metoda DeleteAsync. Návratová hodnota je typu boolean. Pokud byl zdroj službou nalezen a smazán, je vrácena hodnota true. Na základě této hodnoty je pak metodou kontroleru vrácena odpověď klientovi. Podle požadavku musí být ověřen stav přihlášky a smazat ji lze pouze, pokud ještě nebyla zpracována. V opačném případě je vyvolána výjimka, kterou zachycuje middleware ExceptionHandler. Po samotném vymazání musí být provedené změny uloženy metodou SaveChangesAsync. Ve zdrojovém kódu 19 je celé tělo popsané služby.

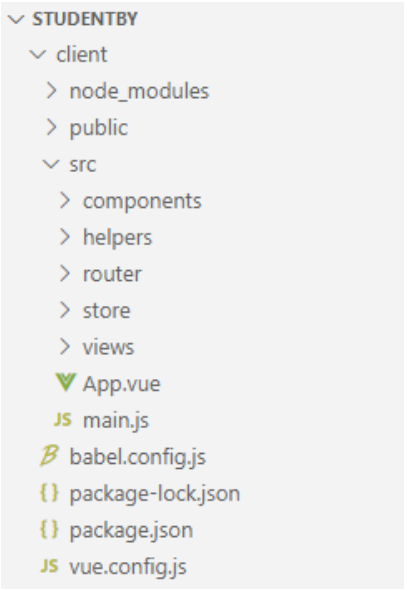
```
public async Task<bool> DeleteAsync(int jobApplicationId)
{
    var jobApplication = await _context.JobApplications
        .Include(ja => ja.JobOffer)
        .FirstOrDefaultAsync(ja => ja.JobApplicationId == jobApplicationId);

    if (jobApplication == null)
    {
        return false;
    }

    if (jobApplication.State != JobApplicationStates.Pending)
    {
        throw new AppLogicException("Přihláška je již zpracovaná");
    }

    _context.JobApplications.Remove(jobApplication);
    await _context.SaveChangesAsync();
    return true;
}
```

Zdrojový kód 19

Výše byla objasněna implementace jednoho koncového bodu aplikace. Jelikož implementace ostatních je strukturou velmi podobná a liší hlavně ve způsobu zpracování dat, tak v této práci již nebude dále popsána. Zdrojové kódy jsou ovšem k dispozici v příloze.

4.2.2 **Frontend**

Klientská část aplikace je vyvíjena pomocí frameworku Vue.js. Jako editor zdrojového kódu postačí software Visual Studio Code s rozšířením Vetur, které umožňuje snazší práci s Vue.js. O vytvoření struktury projektu se postará příkaz *vue create* z nástroje Vue CLI. Tento příkaz zároveň vygeneruje základní zdrojový kód aplikace. Následně je ještě nezbytné nainstalovat plugin BootstrapVue a vytvořit adresář helpers. Výsledkem je adresářová struktura na obrázku 10.



Obrázek 10 - Adresářová struktura frontendu

Jak již bylo v kapitole 3.9.1 zmíněno, Vue.js aplikace jsou tvořeny pomocí komponent. Ty jsou zanořeny v kořenové komponentě App.vue. Komponenty, jejichž zobrazení závisí na navigaci v aplikaci, se nachází v adresáři views. Zbylé jsou ve složce components.

4.2.2.1 Bootstrap

Místo definování vlastního stylopisu je v aplikaci využit plugin BootstrapVue, který poskytuje populární CSS knihovnu Bootstrap v podobě Vue komponent. Bootstrap nabízí řadu kvalitních prvků uživatelského prostředí, které případně lze do jisté míry modifikovat. Ve zdrojovém kódu 20 je příklad využití komponenty z BootstrapVue, která vykreslí navigační panel aplikace.

```
<b-navbar toggleable="md" type="dark" variant="primary"></b-navbar>
```

Zdrojový kód 20

4.2.2.2 Router

Rozšíření vue-router, které umožňuje vykreslovat komponenty dle adresy URL, je nastaveno ve složce router. Nejdůležitější částí konfigurace je pole s názvem routes. Objekty v tomto poli přiřazují Vue komponentu ke specifikované cestě.

V ukázce zdrojového kódu 21 je zároveň uvedeno jméno přiřazení ve vlastnosti name. Využití jména má výhodu, že pokud bude cesta v definici změněna, tak funkčnost směrování nebude narušena. Toto ovšem platí za předpokladu, že komponenty volají router rovněž podle jmen jako ve zdrojovém kódu 21.

```
{
  path: '/login',
  name: 'Login',
  component: Login
}
```

Zdrojový kód 21

```
<b-nav-item :to="{ name: 'Login' }">
```

Zdrojový kód 22

V aplikaci je v několika případech nutné pomocí URL předávat komponentám data. V případě zobrazení detailu studenta při navigaci na /student/1, je především nutné v nastavení routeru uvést, že část cesty je tvořena proměnou. To je učiněno přidáním dvojtečky před název proměnné. Vlastnost props s hodnotou true znamená, že obsah proměnné bude předán do props příslušné komponenty. Volání routeru je obdobné jako v předchozí ukázce. Jedinou změnou je předávání objektu params, který obsahuje klíč studenta. Ve zdrojových kódech 23 a 24 je tento případ ukázán.

```
path: '/operator/students/:studentId',
props: true
```

Zdrojový kód 23

```
:to="{name: 'OperatorStudentDetail', params: { studentId: st.studentId }}"
```

Zdrojový kód 24

4.2.2.3 Komunikace se serverem

Pro získání dat, která by klientská část aplikace mohla zobrazovat, je nezbytné vyvinout funkci pro volání vystaveného webového API. Tato funkce bude volána na více místech. Je tedy vhodné navrhnout ji tak, aby množství opakovaného kódu bylo co nejmenší.

Pokud je funkci předáno tělo požadavku, pak musí být převedeno na textový řetězec obsahující JSON pomocí funkce JSON.stringify. Následně je volána funkce fetch, která má za následek odeslání HTTP požadavku. Adresa je sestavena z proměnné části předané v parametru funkce a ze základní neměnné části, která je definována ve stavovém kontejneru aplikace. Ošetření chyb je komplikovanější. Funkce fetch vyvolá výjimku, pokud nastane chyba při odesílání požadavku, nikoliv při odpovědi s chybovým kódem. Výjimka je tedy ošetřena v konstrukci catch. Pokud je volání serveru úspěšné, pak je odpověď zpracována podle jejího stavového kódu. V případě kódu 200 nebo 201 je obsah odpovědi převeden na objekt. Tento objekt je následně vrácen komponentě. V případě chybového kódu je opět volána funkce raiseError, která uloží chybovou hlášku do stavového kontejneru a vyvolá výjimku. Zdrojový kód 25 se vztahuje k výše zmíněné funkci.

```
async function httpRequest(method, specUrl, body=null)
{
    var stringBody = null;
    if (body) {
        stringBody = JSON.stringify(body);
    }

    var response = await fetch(store.state.baseApiUrl + specUrl, {
        method: method,
        mode: "cors",
        headers: {
            "Content-Type": "application/json",
            "Authorization": "Bearer " + store.state.accessToken
        },
        body: stringBody,
    })
    .catch(() => {
        raiseError({error: 'Vyskytla se chyba', detail: null});
    });

    switch (response.status) {
        case 200: case 201:
            return response.json();
        case 204:
            return null;
```

```
        case 401:
            raiseError({error: 'Neautorizovaný požadavek', detail: null});
            break;
        case 403:
            raiseError({error: 'Nedostatečná oprávnění', detail: null});
            break
        default:
            error = await response.json();
            raiseError(error);
            break;
    }
}
```

Zdrojový kód 25

Obsah proměnné pro chybovou hlášku sleduje kořenová komponenta pomocí watcheru. V případě změny hodnoty sledované proměnné je zavolána přiřazená funkce, která v tomto případě zobrazí varování o chybě.

```
watch: {
  errorMsg: function (errorMsg) {
    this.$bvToast.toast(errorMsg.error, {
      title: "Chyba",
      variant: "danger",
      solid: true,
    });
  },
}
```

Zdrojový kód 26

4.2.2.4 Seznam pracovních nabídek z pohledu studenta

Stejně jako byl v kapitole o implementaci serverové části popsán vývoj vybraného koncového bodu, tak i zde je objasněn vývoj vybrané komponenty. V logice komponenty (viz zdrojový kód 27) je uvedeno, že komponenta pracuje s prázdným polem jobOffers. Toto pole má být naplněno daty z vystaveného API. Dále je tedy nutné specifikovat metodu pro získání dat ze serveru a po připojení komponenty tuto metodu zavolat.

```
data() {
    return {
        jobOffers: []
    };
},
methods: {
    getJobOffers() {
        this.jobOffers = [];
```

```
        apiService
        .get("/job-offers")
        .then((response) => {
            this.jobOffers = response;
        })
        .catch(() => {});
    }
},
mounted() {
    this.getJobOffers();
}
```

Zdrojový kód 27

Samotné zobrazení dat je pro lepší přehlednost a použitelnost rozděleno do dalších komponent. Komponenta PageHeader zobrazuje nadpis aktuální stránky a EmptyList vykresluje informaci, že je pole prázdné. Důležitá je komponenta JobListItem, která zobrazuje jednu položku v seznamu nabídek a je vykreslována pro každý objekt v poli pomocí atributu v-for. Použití těchto dílčích komponent je ve zdrojovém kódu 28 a výsledné zobrazení lze vidět na obrázku 11.

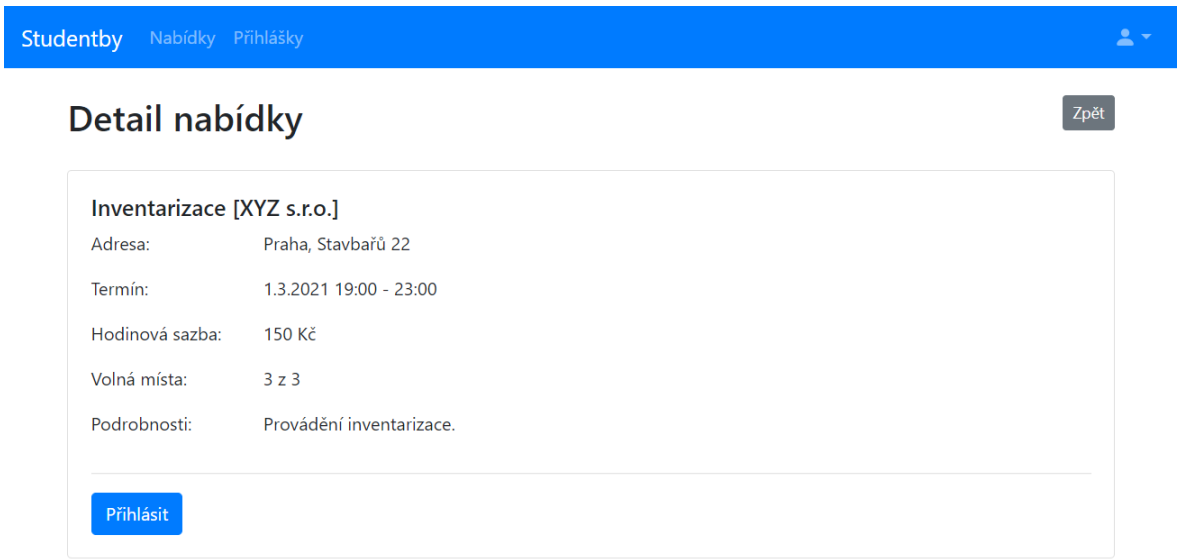
```
<template>
  <div>
    <PageHeader v-bind:title="'Nabídky'"></PageHeader>
    <div class="mt-2">
      <b-list-group v-if="jobOffers && jobOffers.length > 0">
        <JobListItem
          v-bind:key="jobOffer.id"
          v-for="jobOffer in jobOffers"
          v-bind:job="jobOffer"
          v-bind:onClickLink="{
            name: 'StudentJobOfferDetail',
            params: { jobOfferId: jobOffer.jobOfferId },
          }"
        ></JobListItem>
      </b-list-group>
      <EmptyList v-else></EmptyList>
    </div>
  </div>
</template>
```

Zdrojový kód 28

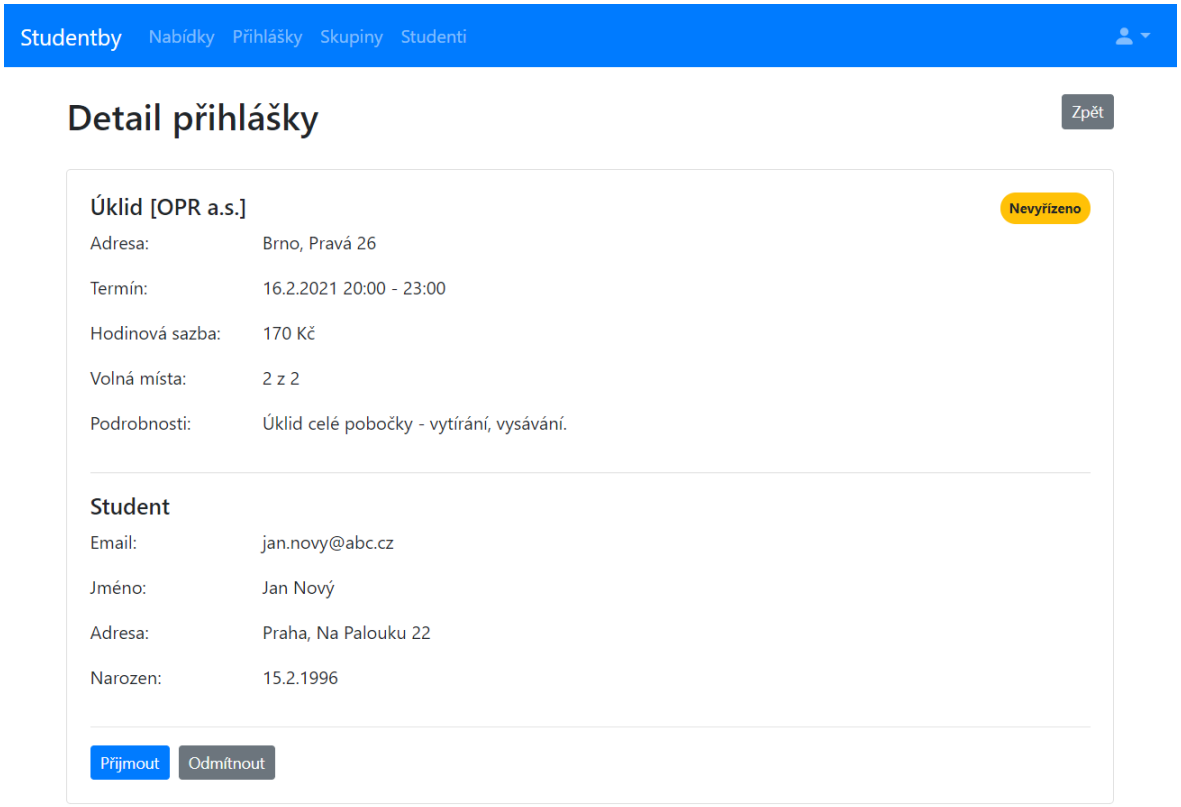
Obrázek 11 – Seznam pracovních nabídek

4.2.2.5 Další stránky

Pro porovnání s wireframem vytvořeným v kapitole 4.1.4 je na obrázku 12 předveden výsledek stejné stránky. Jediným rozdílem oproti návrhu je seskupení možnosti odhlášení uživatele a zobrazení profilu do rozevírací nabídky. Na obrázku 13 je dále předvedena stránka, na které operátor rozhoduje o přijetí přihlášky k pracovní nabídce.



Obrázek 12 - Detail nabídky z pohledu studenta



Obrázek 13 - Detail přihlášky z pohledu operátora

5 Výsledky a diskuse

5.1 Testování

Obě části aplikace byly v průběhu implementace testovány autorem a objevené chyby byly opraveny.

Dále proběhlo otestování čtyřmi subjekty, kteří aplikaci testovali pouze přes klientskou část aplikace. Dva z nich testovali aplikaci z pohledu studenta, jeden z pohledu zaměstnavatele a jeden z pohledu operátora i zaměstnavatele.

Pro roli studenta byly otestovány následující funkce:

- vytvoření účtu,
- přihlášení,
- ověření aktivace účtu,
- výběr pracovní nabídky,
- vytvoření přihlášky,
- zrušení přihlášky.

Registrace, přihlášení a ověření aktivace proběhlo v obou případech bez problémů. U výběru pracovní nabídky naopak oba subjekty poznamenaly absenci filtrování a řazení nabídek. V dalším kroku se jeden ze subjektů pokusil o vytvoření přihlášky k již plné nabídce, ale informace o chybné operaci byla podle něj dostatečně srozumitelná. Zrušení přihlášky opět nečinilo žádný problém. Oba subjekty ocenily snadné sjednání brigády.

Při testování pohledu zaměstnavatele byl subjektům přidělen vytvořený účet a dostaly za úkol provést tyto operace:

- přihlášení,
- zjištění k jaké skupině účet přísluší,
- vytvoření nové pracovní nabídky,
- smazání vytvořené pracovní nabídky.

Problémovým úkolem pro oba subjekty bylo zjištění příslušnosti ke skupině, jelikož neviděly možnost zobrazení vlastního profilu, která je v rozevírací nabídce. Ostatní operace proběhly bez komplikací. Jeden ze subjektů poznamenal, že grafické rozhraní působí prázdné a že by bylo vhodné přidat přehledy a statistiky o vytvořených nabídkách.

Z pohledu operátora byly určeny k otestování následující operace:

- přihlášení,

- vytvoření nové skupiny,
- vytvoření účtu zákazníka pro danou skupinu,
- aktivace studentského účtu,
- schválení přihlášky k pracovní nabídce,
- zadání docházky studenta.

Testovací subjekt poznamenal, že při vytváření skupiny by mělo být možné zadat další informace, například kontakt. Dále měl subjekt problém nalézt způsob, kam zadat docházku studenta. Tento proces se provádí v detailu příslušné pracovní nabídky.

Závěrem ode všech subjektů vzešla připomínka ke grafickému rozhraní, které by bylo vhodné pozměnit na uživatelsky příjemnější.

5.2 Možnosti rozšíření

Vyvinuté řešení v této práci není zcela hotové, lze jej označit spíše za první verzi. V případě skutečného využití aplikace jsou v této části sepsány návrhy pro rozšíření. Tyto návrhy byly sestaveny na základě poznámek testovacích subjektů a poznatků autora.

Užitečným rozšířením by byla implementace hodnocení studentů i skupin. Zákazníci by pak určovali minimální hodnocení studenta, který se smí přihlásit. Spolu s hodnocením by bylo možné udělovat varování, například při absenci studenta. Při dosažení určitého počtu varování by byl studentovi účet deaktivován.

Serverová část by zároveň měla mít možnost odesílat emailové zprávy pro ověřování vytvořených účtů nebo pro notifikace nových událostí. Také by mohl být vyvinut algoritmus pro doporučování nabídek studentovi na základě jeho preferencí a bydliště.

V klientské části lze zdokonalit vzhled grafického rozhraní, dbát na zásady přístupnosti a sémantiku.

6 Závěr

Cílem této bakalářské práce byl návrh a implementace řešení pro sjednávání studentských prací. Dílčími cíli bylo vytvoření dokumentace vyvinutého webového API a přiblížení využitých technologií.

V teoretické části byla zmíněna doporučená pravidla tvorby webového rozhraní. Byl objasněn princip komunikace mezi klientskou a serverovou částí aplikace pomocí protokolu HTTP. Dále byly popsány využité technologie z architektury .NET, kterými jsou programovací jazyk C#, framework ASP.NET Core, balíčkovací systém NuGet a nástroj pro objektově-relační mapování Entity Framework Core. Byl popsán princip využití frontendového frameworku Vue.js. Byly také vysvětleny nezbytné pojmy jako je například middleware a jednostránková aplikace.

První část praktické části práce tvořila analýza vyvíjeného řešení. Především byly sepsány autorem stanovené požadavky na výslednou aplikaci. Dále byly určeny zásady pro pojmenování ve zdrojovém kódu a byl navržen datový model.

Druhou částí byla samotná implementace backendu i frontendu. Kvůli množství vyvinutých funkcí byly představeny pouze ty nejdůležitější a byly doplněny o ukázky zdrojového kódu. Cíl vytvořit dokumentaci webového API byl splněn v kapitole 4.2.1.9, kde bylo popsáno zapojení nástroje pro automatickou tvorbu zmíněné dokumentace. V kapitole Výsledky a diskuse byla aplikace otestována a byly sepsány návrhy pro další rozšíření.

7 Seznam použitých zdrojů

[1] REYNDERS, Fanie. *Modern API Design with ASP.NET Core 2*. ISBN 9781484235188.

[2] ARORAA, Gaurav a Tadit DASH. *Building RESTful Web services with .NET Core*. ISBN 9781788291576.

[3] PATTANKAR, Mithun a Malendra HURBUNS. *Mastering ASP.NET Web API*. ISBN 9781786463951.

[4] Identifying resources on the Web. *MDN web docs* [online]. [cit. 2020-12-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Identifying_resources_on_the_Web

[5] HTTP Messages. *MDN web docs* [online]. [cit. 2020-12-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

[6] HTTP. *MDN web docs* [online]. [cit. 2020-12-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP

[7] FIELDING, Roy. *Hypertext Transfer Protocol -- HTTP/1.1* [online]. [cit. 2020-12-03]. Dostupné z: https://tools.ietf.org/html/rfc2616

[8] .NET architectural components. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/components

[9] Cross-platform targeting. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/library-guidance/cross-platform-targeting

[10] Overview of .NET Framework. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview

[11] PRICE, Mark J. *C# 8.0 and .NET Core 3.0: modern cross-platform development : build applications with C#, .NET Core, Entity Framework Core, ASP.NET Core, and ML.NET using Visual Studio Code*. Fourth edition. 2019. ISBN 978-1788478120.

[12] Introduction. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction

[13] Introduction to the C# language and .NET. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/

[14] ASP.NET Core. *GitHub* [online]. [cit. 2020-12-03]. Dostupné z: https://github.com/dotnet/aspnetcore

[15] An introduction to NuGet. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/nuget/what-is-nuget

[16] How NuGet resolves package dependencies. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/nuget/concepts/dependency-resolution

[17] Database Providers. *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli

[18] ŽŮREK, Michal. Lekce 1 - MS-SQL krok za krokem: Úvod do MS-SQL a příprava prostředí. *ITnetwork.cz* [online]. [cit. 2020-12-03]. Dostupné z: https://www.itnetwork.cz/ms-sql/mssql-tutorial-uvod-a-priprava-prostredi

[19] Choose Between Traditional Web Apps and Single Page Apps (SPAs). *Microsoft Docs* [online]. [cit. 2020-12-03]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps

[20] What is JSON?. *W3Schools* [online]. [cit. 2020-12-03]. Dostupné z: https://www.w3schools.com/whatis/whatis_json.asp

[21] Introducing JSON. *JSON* [online]. [cit. 2020-12-03]. Dostupné z: https://www.json.org/json-en.html

[22] Vue.js 1.0.0 release. *GitHub* [online]. [cit. 2020-12-03]. Dostupné z: https://github.com/vuejs/vue/releases/tag/1.0.0

[23] YOU, Evan. Vue.js. *GitHub* [online]. [cit. 2020-12-03]. Dostupné z: https://github.com/vuejs/vue

[24] Vue.js guide. *Vue.js* [online]. [cit. 2020-12-03]. Dostupné z: https://vuejs.org/v2/guide/

[25] Vue router guide. *Vue.js* [online]. [cit. 2020-12-03]. Dostupné z: https://router.vuejs.org/guide/

8 Přílohy

Na přiloženém CD se nachází zdrojový kód aplikace Studentby, která byla v rámci této práce vyvinuta.