Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vývoj serverové aplikace za užití REST API v prostředí .NET

Bc. Martin Jermář

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Martin Jermář

Systémové inženýrství a informatika
Informatika

Název práce

Vývoj serverové aplikace za užití REST API v prostředí . NET

Název anglicky

Development of a server application using REST API in an environment. NET

Cíle práce

Práce je zaměřena na problematiku využití REST API v prostředí Microsoft .NET Framework. Hlavním cílem je vytvoření serverové aplikace v jazyce C# vystavující rozhraní pro klientské aplikace v restauračním prostředí.

Metodika

Diplomová práce je složena ze dvou hlavních částí. V teoretické části práce jsou popsána teoretická východiska z oblasti softwarového inženýrství, návrhový vzor Dependency injection a REST API architektura.

Praktická část práce se bude sestávat z implementace serverové části aplikace za využití teoretických poznatků. Dále bude implementována ukázková klientská aplikace, které bude demonstrovat možnosti využití API vytvořené serverové části. Poznatky získané při implementaci budou následně shrnuty a budou navrženy případné další možnosti budoucího rozvoje aplikace.

Doporučený rozsah práce

60-80 stran

Klíčová slova

rest, api, c#, .net framework, serverová aplikace, orm, mssql, http, json, dependency injection, json

---

Doporučené zdroje informací

KURTZ, Jamie. ASP.NET MVC 4 and the Web API: building a REST service from start to finish. Berkeley, CA: Apress, [2013]. Expert's voice in ASP.NET.

NASH, Trey. Accelerated C# 2010. New York: Distributed to the book trade worldwide by Springer Science+Business Media, c2010. Expert's voice in C#. ISBN 1430225386.

Referenční příručka [online]. 2018 [cit. 2018-06-24]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/618ayhy6.aspx

SHARP, John. Microsoft Visual C# 2013 step by step. Sebastopol, California: O'Reilly Media/Microsoft Press, [2013]. Developer technology series. ISBN 073568183X.

Předběžný termín obhajoby

2019/20 ZS – PEF (únor 2020)

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 11. 3. 2019

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 11. 3. 2019

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 22. 03. 2020

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Vývoj serverové aplikace za užití REST API v prostředí .NET“ jsem vypracoval samostatně pod vedením vedoucího diplomové 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 diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 25.03.2020 _____

Poděkování

Rád bych poděkoval vedoucímu práce panu Ing. Jiřímu Brožkovi, Ph.D. za odborné vedení mé práce, jeho čas, ochotu a připomínky a všem, kteří mě při studiu podporovali.

Vývoj serverové aplikace za užití REST API v prostředí .NET

Abstrakt

Diplomová práce je zaměřená na problematiku vývoje REST API v prostředí. NET. V teoretické části je popsán architektonický styl REST, historie, HTTP metody a stavové kódy. Dále jsou zde uvedeny doporučené postupy a pravidla, jak navrhovat vlastní REST API. Je zde věnována kapitola návrhovému vzoru Dependency injection, který se využívá ve vývoji v serverových a klientských aplikací. V poslední části jsou popsány technologie a nástroje, které jsou použity k vývoji aplikace. V praktické části je stručně provedená analýza požadavků a následně je vytvořen diagram užití spolu s logickým modelem databáze. Ve vývoji je popsána funkčnost serverové části, postupně je popsána implementace nastavení projektu, persistence dat pomocí ORM NHibernate, vytvoření a seskupení akčních metod, úlohy na pozadí, logování a bezpečnost aplikace. V rámci diplomové práce je vyvinuta i klientská část v Angularu, díky které je možné demonstrovat využití API vytvořené v serverové části. Cílem práce je vytvořit open-source řešení pro restaurační business.

Klíčová slova: rest api, c#, .net core, orm hibernate, mssql, http, json, dependency injection

Development of server application using REST API in .NET enviroment

Abstract

This thesis is focused on the development of REST API in the environment. NET. The theoretical part describes the architectural style of REST, history, HTTP methods and status codes. It also lists the best practices and rules for designing your own REST API. There is a chapter devoted to the design pattern Dependency injection, which is used in the development of server and client applications. The last part describes the technologies and tools that are used to develop the application. In the practical part, there is a brief analysis of requirements and then a usage diagram is created together with a logical model of the database. The development describes the functionality of the server part, gradually describes the implementation of project settings, data persistence using ORM NHibernate, creation and grouping of action methods, background tasks, logging and application security. In the framework of the thesis is also developed the client part in Angularu thanks to which it is possible to demonstrate the use of API created in the server part. The aim of this work is to create an open-source solution for restaurant business.

Keywords: rest api, c#, .net core, orm hibernate, mssql, http, json, dependency injection

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

Na většině základních a středních škol je běžné, že se objednává jídlo vždy alespoň den dopředu, aby se nakoupily suroviny, které tak akorát vystačí na uvaření objednaného počtu jídel. Cena zde hraje hlavní roli, protože se zde počítá na jednotky korun. Systém skvěle funguje, žádné jídlo se zbytečně neuvaří navíc, tudíž zde nedochází k zbytečnému vyhazování jídla. Zároveň je možné ceny přesně optimalizovat. Stejně, jako děti ve škole, se chodí stravovat dospělí lidé v práci o polední pauze na oběd do restauračních zařízení. Vytvoření podobného systému by mohlo vést ke snížení ceny, zpřesnění nákupu surovin a následně ke snížení vyhazování nepoužitých surovin či uvařeného jídla. Restaurace přistupují k tomuto problému různě, některé uvaří menší počet jídel, a pokud přijdete na oběd později, nemusí se na vás už dostat jídlo, které jste si předem naplánovali, jiné restaurace uvaří více porcí, a když se nesní, jsou buď zamraženy (což je podle zákona zakázané), nebo se vyhodí. Vzniká tedy nejen finanční škoda. Majitelé restaurací při kalkulaci s touto ztrátou počítají a zahrnují ji do cen jídel.

Do restaurací na polední menu chodí převážně zaměstnanci z okolí. Je pravděpodobné, že ne všichni si chtějí objednávat jídlo na druhý den. Stále je zde ale prostor vytvořit řešení pro lidi, kteří by se zapojili do objednávkového systému, zvláště v případě určité kompenzace z pohledu restaurace. Může se jednat o benefity typu: polévka zdarma nebo procentní sleva na oběd atd. Pro restaurace by to znamenalo zlepšení odhadů při určování počtu navařených obědů a také možný příliv dalších zákazníků, kteří mají rádi benefity. Problémem vytvoření a zavedení tohoto systému je určitě cena. Vývoj systému nikdo nebude ochoten zaplatit. Existují tři cesty. Vytvořit profesionální produkt a nabízet ho, koupit profesionální produkt a nabízet ho nebo vytvoření open-source řešení. Ideální řešení je platit pouze měsíční provoz serveru, který poskytne nejen objednávkový systém, ale i komplexní řešení od vytváření poledního menu, pravidelného zasílání newsletteru, systému rezervací na kliknutí, stahování hodnocení, až po celkovou administraci webu v uživatelském rozhraní. Vytvoření serverové části v Net Coru umožňuje vyvíjet aplikace nezávislé na operačním systému, a díky Visual Studio Code také zdarma. Následně díky využití REST API je možné vyvinout klienta v jakémkoliv jazyce a pouze se spojit se severem pomocí HTTP metod. Vývojáři tedy mohou nadále vyvíjet klientské aplikace ve svých oblíbených jazycích.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem diplomové práce je vytvoření serverové aplikace, která umožní již zmíněnou funkčnost, pomocí vystavení REST API. Serverová aplikace je vytvořena v .Net Coru, databáze v MS SQL. Propojení serverové aplikace a databáze je řešeno pomocí objektově relačního frameworku NHibernate. Pro demonstraci využití API je vytvořena ukázková klientská aplikace v Angularu. V obou aplikacích je využíván návrhový vzor dependency injection.

2.2 Metodika

Ke splnění uvedených cílů je potřeba nastudování literárních a webových zdrojů a využití poznatků získaných během praxe. Na základě syntézy získaných informací budou formulována teoretická východiska. Dále jsou popsány technologie a nástroje použité k vývoji aplikace.

V praktické části je provedena analýza, ve které vznikne diagram užití, specifikace případu užití a logický model databáze. Implementace serverové části je seskupena do logických celků. Nejdříve je popsáno základní nastavení, tj, registrace použitých frameworků, služeb a modulů. Poté jednotlivé části jako persistence dat pomocí objektově relačního mapovaní, vytavení API, úlohy na pozadí, logování a bezpečnost. Využití API je popsáno v klientské aplikaci. Na základě zkušenosti a poznatků jsou formulovány závěry a navrženy další možnosti rozšíření aplikace.

3 Teoretická východiska

3.1 Úvod do REST

REST je akronym pro Representational State Transfer. Je to architektonický styl pro hypermediové distribuované systémy. Poprvé byl představen v disertační práci Roye Fieldinga v roce 2000. Roy Fielding je dále jeden ze spoluautorů protokolu HTTP. REST byl vyvinut spolu s protokolem HTTP/1.1 na základě stávajícího HTTP /1.0. REST API, které může být volně zaměněno se stejným významem s RESTful API. Jedná se o aplikační rozhraní, které využívá HTTP metody pro přístup ke zdrojům, které jsou dále popsány v kapitole 3.1.2. Tento přístup je často využíván pro komunikaci webových služeb. REST technologie je dnes obecně preferována před robustním řešením webových služeb SOAP. REST již v roce 2017 využívalo 83 % veřejných API. Používají je společnosti jako Amazon, Google, LinkedIn nebo Twitter. Rozdíl oproti SOAP je, že REST je orientovaný datově, a nikoliv procedurálně. Další hlavní benefity jsou popsány v následujících bodech. [1]

- REST umožňuje velké množství datových formátů (JSON, XML, CSV, HTML, prostý text…), zatímco SOAP pouze XML.
- Díky spojení s JSON formátem (které typicky funguje lépe s daty a nabízí rychlejší parsování) je obecně REST považován za jednodušší na práci.
- Díky JSON nabízí REST lepší podporu pro webové klienty.
- REST poskytuje vynikající výkon, zejména prostřednictvím ukládání informací do mezipaměti, které se nemění a nejsou dynamické.
- REST zatěžuje méně síť.
- Je mnohem snazší na implementaci a následnou integraci, nejsou zde potřeba žádné WSDL soubory, které popisující komunikaci. Vývojáři tak pracují mnohem rychleji a jednoduše mohou přidávat či upravovat funkčnosti.
- Dobře funguje s Javascriptem. [2]

3.1.1 Vznik webové architektury

V prosinci na konci roku 1990 vnikla celosvětová síť. Na stejnojmenném projektu bylo vynalezeno a implementováno následující:

- Jednotný zdrojový identifikátor (URI - Uniform Resource Identifier), který každému dokumentu jasně přiřadil unikátní adresu.
- Hypertextový transferový protokol (HTTP - HyperText Transfer Protocol), jazyk založený na zprávách pomocí kterých komunikuje počítač na internetu.
- Hypertextový značkovací jazyk (HTML – Hypertext Markup Language), který reprezentuje dokumenty na webu.
- První webový server.
- První webový prohlížeč, nejprve pojmenovaný po názvu projektu, později Nexus.
- První WYSIWYG HTML editor, který byl součástí webového prohlížeče.

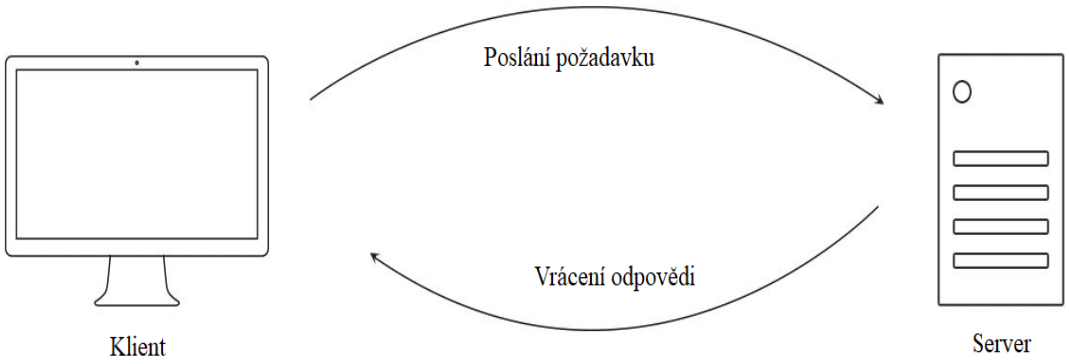
Následně od tohoto momentu začal exponenciálně růst počet uživatelů internetu. V průběhu 5 let to bylo až 40 miliónů uživatelů. V důsledku tak obrovského a rychlého růstu vše mířilo ke kolapsu. Webový provoz již přerůstal kapacitu internetové infrastruktury. O rozšiřitelnost internetu se začal zajímat již zmíněný Roy Fielding. Na základě provedené analýzy o škálovatelnosti webu, se rozhodl definovat omezení. Tato omezení měla za úkol ovlivnit výkon, škálovatelnost, zjednodušení, interoperabilitu, spolehlivost a komponentní přenositelnost. Tyto vlastnosti jsou zapouzdřeny do následujících 6 skupin. [3]

1) Klient-Server (Client-Server)
2) Jednotné rozhraní (Uniform Interface)
3) Vrstvený systém (Layered Systém)
4) Mezipaměť (Cache)
5) Bezstavovost (Sateless)
6) Kód na vyžádání (Code-On-Demand)

3.1.1.1 Klient-Server

Omezení klient-server uplatňuje separaci mezi konzumentem a serverem (možno také označit používaným anglickým termínem „back-end“), kde probíhá obchodní logika a často také databázová implementace. Toto omezení patří mezi klasickou aplikační architekturu, která je hojně využívána u webových aplikací. Funguje tak, že klient odešle požadavek na server, který požadavek zpracuje a pošle zpět odpověď. Komunikace je znázorněna na obrázku 1. Uplatněním tohoto omezení zaručuje zcela nezávislý vývoj jak na straně klienta, tak straně serveru, při zachovaní rozhraní. [4]

Obrázek 1 – Aplikační architektura klient server



Zdroj: [25]

3.1.1.2 Jednotné rozhraní

Výsledkem jednotného rozhraní je to, že požadavky různých klientů vypadají stejně, ať už jde o prohlížeč Chrome, linuxový server, python skript, aplikaci pro Android nebo cokoli jiného. Toto omezení se skládá ze čtyř základních částí:

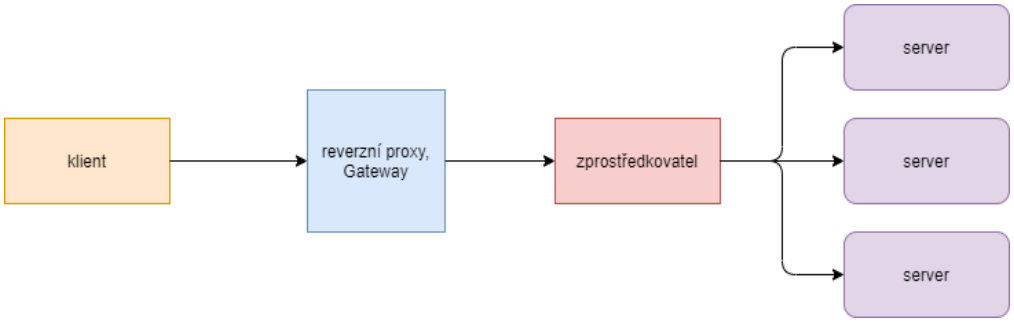
1) Požadavek na server musí obsahovat identifikátor zdroje.
2) Odpověď, kterou server vrátí, obsahuje dostatek informací, takže klient může požadavek upravit.
3) Každý požadavek na rozhraní API obsahuje všechny informace, které server potřebuje k provedení požadavku a každá odpověď, kterou server vrátí, obsahuje všechny informace, které klient potřebuje, aby porozuměl odpovědi.

4) Hypermedia jako stav aplikace. To znamená, že server v odpovědi klientovi posílá informace (linky/odkazy), jak může klient změnit stav webové aplikace, např.: Klient prvotním požadavkem na konkrétního uživatele dostane v odpovědi vracené severem různé možnosti na změnu stavu, tedy například odkazy na smazání uživatele, úpravu dat uživatele atd. [5]

3.1.1.3 Vrstvený systém

Za účelem zlepšení chování požadavků se přidávají další vrstvy do systémového omezení. Vrstvený systém umožňuje, aby se architektura skládala z hierarchických vrstev omezením chování komponenty tak, aby každá komponenta nemohla „vidět“ za bezprostřední vrstvu, se kterou interaguje. Zprostředkovatelé mohou být také použiti ke zlepšení škálovatelnosti systému. Zlepšení se dosáhne umožněním vyrovnávání zatížení služeb ve více sítích a procesorech a přidáním reverzní proxy nebo gateway (brána – síťový uzel), viz obrázek 2. Hlavní nevýhoda vrstvených systémů spočívá v tom, že ke zpracování dat přidávají režii a latenci, což snižuje výkon. [6] [7]

Obrázek 2 – Vrstvený systém



Zdroj: vlastní zpracování [7]

3.1.1.4 Mezipaměť

Aby se zvýšila účinnost sítě, přidávají se do stylu REST omezení mezipaměti. Omezení mezipaměti vyžadují, aby data v odpovědi na požadavek byla označena jako kešovaná nebo nekešovaná. Je-li odpověď uložená v mezipaměti, klientská mezipaměť může znovu použít tato data pro pozdější požadavky. Výhodou je, že se omezí částečně, nebo úplně posílání některých požadavků, zlepší se účinnost, škálovatelnost a výkon vnímaný uživateli.

3.1.1.5 Bezstavovost

V rámci komunikace mezi klientem a serverem není na serveru nikde ukládán stav klienta. Je tedy potřeba s každým požadavkem posílat na server veškeré informace, které jsou potřeba v daném kontextu. Díky této klíčové funkci je velmi snadné dále rozšiřovat server. [7]

3.1.1.6 Kód na vyžádání

Posledním a volitelným omezením je kód na vyžádání, který klientovi umožňuje přístup ke konkrétním prostředkům ze serveru bez znalosti toho, jak je zpracovat. Tento styl je obvykle implementován webovými aplikacemi, které mají klienty používající skriptovací jazyky na straně klienta, například JavaScript. Je tedy možné přenést vykonávání určitých operací na klienta a ušetřit tak výkon na straně serveru. [4]

3.1.2 Jednotný indentifikátor zdroje – URI

REST API používají URI k adresování zdrojů. Tim Berner-lee prohlásil o neprůhlednosti URI, že je potřeba požívat URI pouze na odkázání se na objekt, a ne se dívat na obsah řetězce z URI za účelem získání informací. [5] [8]

3.1.2.1 Obecná pravidla pro URI

Formát URI, který je znázorněný na obrázku 3, je definován v RFC 3986. [9]

Obrázek 3 – URI formát

```
URI = scheme "://" authority "/" path [ "?" query ] [ "#"
fragment ]
```

Zdroj: [6]

- Lomítko (/) slouží k označení hierarchického vztahu.
- Nepoužívá se lomítko na konci URI. REST API by to nemělo očekávat a ani poskytovat zdroje k těmto linkům. Je ovšem pravda, že i když uživatel zadá lomítko na konci adresy, tak si s tím většina frameworků poradí a očekávané zdroje nalezne, přestože nebyly definovány.
- Použití pomlčky může pomoci k lepší čitelnosti URI. Je to z důvodu, že v URI se nepoužívají notace Pascal Case ani Camel Case (notace pro víceslovné názvy, kde

se nepoužívají mezery, ale každé nové slovo začíná velkým písmenem, respektive až druhé slovo začíná s velkým písmenem), jelikož vše v části scheme a authority je převedeno na malá písmena a vše v části path je ponecháno. Podtržítko se nepoužívá z důvodu špatné viditelnosti.

- Nepoužívají se ani přípony souborů. Místo toho se formát souborů posílá v hlavičce ve vlastnosti media type. [4]

3.1.2.2 Pravidla pro segment cesty v URI

Každý úsek cesty URI je oddělený lomítkem (/). Přiřazení smysluplných hodnot každému segmentu cesty pomáhá jasně komunikovat hierarchickou strukturu REST API. V první řadě neexistuje jednotný standard, jak navrhovat cestu ke zdrojům. Návrh tedy vychází z konkrétní situace a doporučeních z praxe (Best-Practicies), jak je možné navrhovat cesty. V hlavních doporučeních z praxe je dobré dodržovat hlavně 2 pravidla:

1. Nepoužívat slovesa v názvu cesty.
2. Pokud se pracuje s kolekcí, je doporučené používat podstatné jméno v množném čísle.

Dále je potřeba si dát pozor na vytváření zbytečného množství zdrojů. V následujícím příkladu jsou zobrazeny 2 cesty ke zdrojům.

/api/články/25/autor

/api/články/25

V obou případech je vyslán požadavek na autora článku s id = 25.

- První cesta je zbytečná, protože objekt autor je již součástí odpovědi. Vznikla by tak další cesta na datum článku apod.
- Odpadne i problém s počtem autorů, kdy autorem článku může být jedna nebo více osob.

Vždy ovšem bude záležet na konkrétních potřebách a možnostech. Je možné také pracovat se zanořenými kolekcemi,

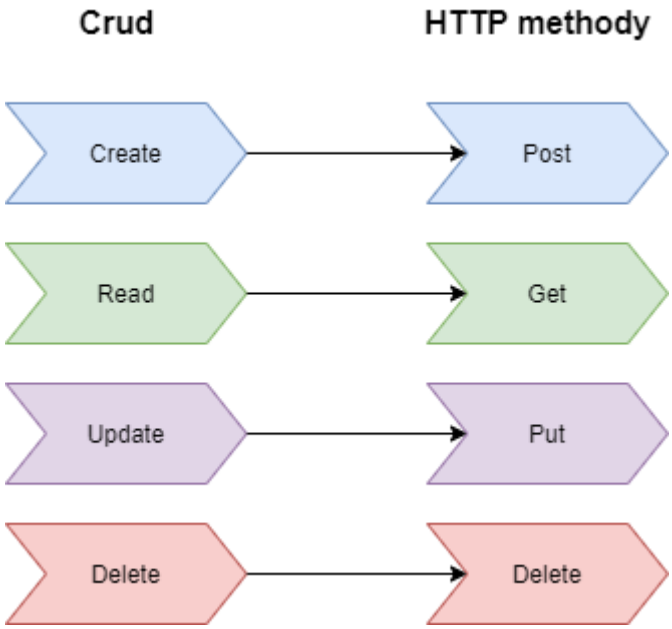
/api/články/25/autoři,

kde požadavek očekává pouze informace o autorech o konkrétním článku. [10]

3.1.3 HTTP metody

HTTP metody (Request Methods) slouží k interakci klienta se serverem a k dotazování se na zdroje. Počet metod (akcí) působí na první pohled jako velmi limitující hlavně pro lidi zvyklé na SOAP, ale právě díky takto limitujícímu počtu metod, kdy v základu se používají hlavně tzv. CRUD metody (GET, POST, PUT a DELETE), je možné predikovat, co který požadavek má vykonat.

Obrázek 3 – URI formát



Zdroj: vlastní zpracování

3.1.3.1 Bezpečnost a idempotence

Metoda HTTP je bezpečná, pokud se nezmění stav serveru. Metoda je tedy bezpečná, pokud vede jen k operaci pro čtení. Mezi bezpečné HTTP metody se řadí GET, HEAD nebo OPTIONS. Všechny bezpečné metody jsou také idempotentní, ale ne všechny idempotentní metody jsou bezpečné. Například PUT i DELETE jsou idempotentní, ale jsou nebezpečné. Přirozeně je vždy možné, že na straně serveru může vzniknout problém nebo chyba, ale za to už není zodpovědný klient.

Metoda HTTP je idempotentní, pokud lze stejný požadavek provést jednou nebo několikrát v řadě se stejným účinkem, přičemž server zůstane stále ve stejném stavu. Idempotentní metoda by tedy neměla mít žádné vedlejší účinky. HTTP metody GET,

HEAD, PUT a DELETE, jsou idempotentní, metody, POST a PATCH ne. Přehled HTTP metod je v následující tabulce. [4] [11]

Tabulka 1 – Přehled HTTP metod

HTTP metody	Idempotence	Bezpečnost
GET	ANO	ANO
HEAD	ANO	ANO
PUT	ANO	NE
DELETE	ANO	NE
POST	NE	NE
PATCH	NE	NE

Zdroj: vlastní zpracování

3.1.3.2 Popis HTTP metod

GET – Metoda se používá pouze pro získávání dat. Požadavek neobsahuje tělo zprávy, odpověď ano.

HEAD – Metoda požaduje odpověď totožnou s odpovědí na požadavek GET, ale bez těla odpovědi.

POST – Metoda se používá k odeslání entity k zadanému zdroji, může způsobit změnu stavu nebo vedlejší účinky na serveru.

PUT – Používá se primárně k aktualizaci zdroje. Pokud zdroj neexistuje, může se server rozhodnout vytvořit nový zdroj. V případě, že se tak stalo, je potřeba informovat uživatele v odpovědi pomocí stavového kódu 201, který indikuje vytvoření. Pokud je existující zdroj upraven, vrací se kód 200 (odpověď obsahuje tělo zprávy) nebo 204 (bez těla zprávy).

DELETE – Používá se k vymazání zdroje identifikovatelného v požadavku. Úspěšná odpověď může obsahovat kódy 200, pokud odpověď obsahuje entitu popisující stav, 202 v případě úspěšného přijmutí požadavku a 204.

PATCH – Metoda se používá k částečné aktualizaci zdroje. Na rozdíl od metody PUT nenahrazuje nebo nevytváří kompletně celou entitu. U této metody je potřeba dát si pozor na podporu. Metoda není například podporována v prohlížeči Internet Explorer 8 a nemusí správně fungovat i v několika dalších technologiích nebo frameworcích jako PHP, TOMCAT nebo třeba DJANGO.

OPTION – Metoda se používá k popisu možností komunikace pro cílový zdroj.

Dále existují metody **TRACE** a **CONNECT**, které již nejsou předmětem diplomové práce. [5] [11] [12]

3.1.4 Stavové kódy

Stavové kódy jsou obsaženy v odpovědích na klientské požadavky. Stavové kódy jsou rozděleny do 5 kategoriích, viz Tabulka 2. Hlavním cílem je informovat uživatele o zpracování požadavku. Stavové kódy protokolu HTTP spravuje organizace IANA (Internet Assigned Numbers Authority). [13]

Tabulka 2 – Stavové kódy HTTP

Kategorie	Rozsah stavových kódů
Informační	1xx
Úspěch	2xx
Přesměrování	3xx
Chyba na straně klienta	4xx
Chyba na straně serveru	5xx

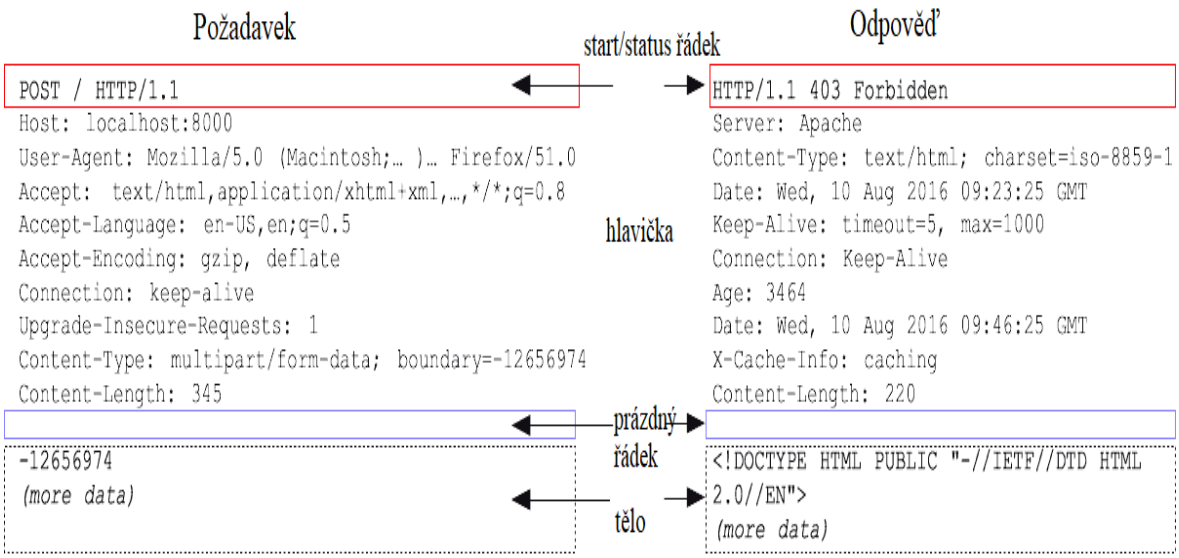
Zdroj: vlastní zpracování

Podrobný přehled všech stavových kódů a jejich význam lze najít na stránkách společnosti IANA. [14]

3.1.5 HTTP zpráva

HTTP zpráva se skládá ze 4 částí, viz obrázek 4. Zpráva je buď požadavek, nebo odpověď. Start řádek je ve zprávě typu požadavek, status řádek pak v odpovědi. [15]

Obrázek 4 – HTTP zpráva



Zdroj: vlastní zpracování [16]

3.1.5.1 Hlavička zprávy

Hlavičky HTTP (Headers) se používají k předávání dalších informací mezi klientem a serverem, jedná se o tzv. metadata v komunikaci. Hlavičky nejsou case sensitive (nerozlišují velká a malá písmena), prvky hlavičky jsou odděleny dvojtečkou a jedná se prvky klíč-hodnota ve formátu řetězců prostého textu. Konec sekce hlavičky je označený prázdným prvkem. V následujících bodech jsou vypsána pravidla pro návrh hlaviček. Pravidla jsou doporučením a nejsou závazná ke komunikaci. [16]

- Content-Type (typ obsahu) by měl být vyplněn, protože tato hodnota udává, jak je možné zpracovat sekvenci bajtů v těle zprávy.
- Contetn-Length (velikost obsahu) se používá ze dvou důvodů. Přijímatel zprávy si může ověřit, že korektně přečetl sekvenci bajtů. Dále je možné pomocí HTTP metody HEAD dopředu zjisti velikost zprávy, než je stažená.
- Last-Modified (poslední úprava) prvek se používá v odpovědích za účelem informování klienta, kdy se naposledy změnil stav zdroje.
- Cache-Control, Expires a Date (kontrola keše, doba vypršení, datum) se používá pro ukládání do mezipaměti. Ukládání do mezipaměti je jednou z nejužitečnějších funkcí postavených na HTTP. Díky využití mezipaměti dojde ke snížení doby odezvy ke klientovi, ke zvýšení spolehlivost a snížení

zatížení serverů API. Ukládání do mezipaměti může probíhat na serveru API, doručováním obsahu sítě (CDN Content delivery network) nebo u klienta.

- Vlastní prvky hlavičky je také možné vytvořit, ale je potřeba dát pozor, aby nebylo ovlivněno chování HTTP metod. [4]

3.1.5.2 Tělo zprávy

HTTP tělo zprávy se používá k přenosu entity spojené s požadavkem nebo odpovědí. Tělo zprávy není povinné. Záleží na typu zprávy a typu HTTP metody. Například metoda GET neposílá v požadavku tělo, ale v odpovědi ho očekává atd. Je ovšem potřeba si opět uvědomit, že jde o doporučení, v praxi je samozřejmě možné poslat tělo zprávy v požadavku metody GET. Technologie to umožňují, avšak jedná se pak o špatný návrh. Na obrázku 5 je vidět stejné tělo zprávy ve dvou nejběžnějších formátech používaných v API. Na levé straně je formát JSON, na pravé pak XML. [5] [15]

Obrázek 5 *– Tělo zprávy JSON, XML*

Zdroj: vlastní zpracování

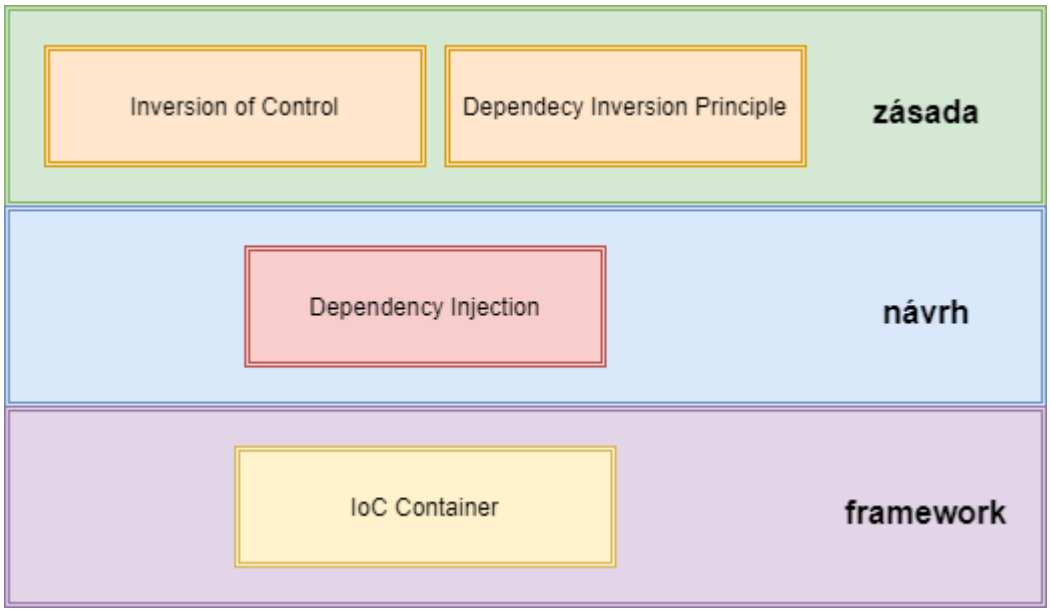
3.2 Dependency Injection

Dependecy injection ve zkratce DI, lze volně přeložit jako vkládání závislostí. V textu diplomové práce se již pracuje se zavedeným anglickým termínem. Dependency

injection je návrhový vzor, který vychází ze zásad Inversion of Control ve zkratce IoC (český termín neexistuje, pod pojmem si lze představit, že se jedná o invertování řízení nebo dohledu nad objektem) a slouží k inicializaci objektů a předání jich jiným objektům, které je vyžadují za běhu programu. Závislost mezi komponentami aplikace je nevyhnutelná ve větších aplikací. Pokud nejsou odkazy závislostí správně navrženy, může to mít velmi negativní dopad na výkon a udržovatelnost kódu. Pojem DI představil Martin Fowler.

Kolem DI a IoC se v literatuře objevují termíny Dependency Inversion Principle ve zkratce DIP (vyšší návrhová zásada) a Inversion of Control Container (IoC container). Jak je zobrazeno na obrázku 6, IoC a DIP by měly být uplatňovány při návrhu tříd. Přestože se jedná zásady doporučující osvědčené postupy, neposkytují konkrétní podrobnosti o implementaci. DI je pak návrhový vzor a IoC container je framework. [6]

Obrázek 6 *– Vkládání závislostí přehled*



Zdroj: vlastní zpracování [17]

3.2.1 Inversion of Control

Návrhová zásada IoC je založena na objektovém programovaní a její cíl je uvolňovat silně provázané třídy (tight coupling classes) v aplikaci. Třída by se měla držet zásady principu jedné odpovědnosti a neměla by nést odpovědnost za vytváření a synchronizaci dat mezi zdrojem a cílem (binding) závislých objektů. IoC je první část, jak se zbavit vysoké závislosti objektů.

V příkladu na obrázku 7 je znázorněna deklarace objektu mathOperation, ale inicializace je už nyní provedena pomocí návrhového vzoru Factory, tzn., že za inicializaci objektu mathOperation třídy MathOperation je zodpovědná třída MathFactory. Je tedy použito invertování řízení inicializace objektu. V prvním kroku je dosáhnuto již nižší závislosti. [17]

Obrázek 7 – Třídy MathFactory a MathLogic

```
public class MathFactory
{
    1 reference
    public static MathOperation GetMathOperation()
    {
        return new MathOperation();
    }
}

public class MathLogic
{
    0 references
    public MathLogic()
    {

    }

    0 references
    public int AddNumbers(int number1, int number2)
    {
        MathOperation mathOperation = MathFactory.GetMathOperation();

        return mathOperation.AddNumbers(number1, number2);
    }
}
```

Zdroj: vlastní zpracování [17]

3.2.2 Dependecy Inversion Principle

DIP patří mezi SOLID zásady, které vynalezl Robert Martin. Dip má 2 definice.

1. Moduly na vyšší úrovni by neměly záviset na modulech na nižší úrovni. Oba by měly být závislé na abstrakci.
2. Abstrakce by neměla být závislá na detailech. Detaily by měly záviset na abstrakci.

V příkladu na obrázku je použito IoC, ale třída MathLogic je stále závislá na třídě MathFactory. Je potřeba použít zásadu DIP za účelem nižší závislosti tříd. Podle první definice by neměla záviset třída MathLogic na MathFactory, ale obě konkrétní třídy by

měly být závislé na abstrakci. Je potřeba vytvořit abstraktní třídu nebo rozhraní, viz obrázek 8, a následně je znázorněna implementace rozhraní. [6] [17]

Obrázek 8 – Rozhraní a implementace IMathOperation

```
public interface IMathOperation
{
    2 references
    int AddNumbers(int number1, int number2);
}


public class MathOperation : IMathOperation
{
    2 references
    public int AddNumbers(int number1, int number2)
    {
        return number1 + number2;
    }
}
```

Zdroj: vlastní zpracování [17]

Dále došlo ke změnám ve třídě MathFactory, kdy statická metoda GetMathOperation vrací typ IMathOperation a ve třídě MathLogic se již nevyskytuje typ MathOperation, ale typ IMathOperation, viz obrázek 9. Význam tohoto kroku spočívá v tom, že třída MathLogic (vyšší modul) již není závislá na konkrétní třídě, ale na abstrakci. Nižší modul MathOperation je rovněž závislý na abstrakci. Povedlo se opět snížit závislost mezi objekty, avšak stále není dosaženo úplného zbavení se závislosti. MathLogic stále obsahuje třídu MathFactory k získaní reference na rozhraní IMathOperation.

Obrázek 9 – Třídy MathFactory a MathLogic

```
public class MathFactory
{
    1 reference
    public static IMathOperation GetMathOperation()
    {
        return new MathOperation();
    }
}
```

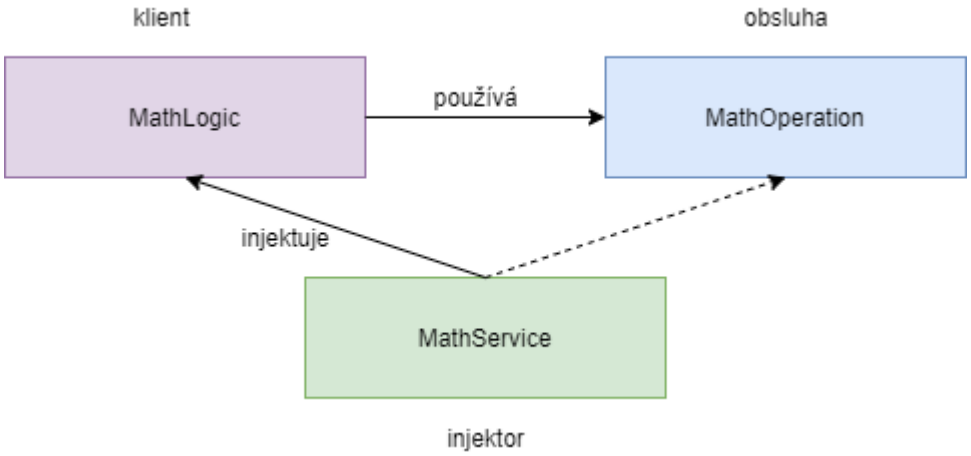
```
public class MathLogic
{
    private IMathOperation mathOperation;

    0 references
    public MathLogic()
    {
        mathOperation = MathFactory.GetMathOperation();
    }

    0 references
    public int AddNumbers(int number1, int number2)
    {
        return mathOperation.AddNumbers(number1, number2);
    }
}
```

Zdroj: vlastní zpracování [17]

3.2.3 Dependecy Injection

DI je návrhový vzor používaný k implementaci IoC. Umožňuje vytvářet závislé objekty mimo třídy a poskytovat tyto objekty třídám pomocí 3 způsobů. Pomocí DI se tedy přesune vytváření bindování závislých objektů mimo třídy, které na nich závisí.

Obrázek 10 – Depedency injection



Zdroj: Vlastní zpracování [17]

Obrázek 10 znázorňuje vztah mezi klientem, službou a injektorem. Třída MathService má na starost vytvořit třídu obsluha a injektuje tento objekt klientovi, tedy do třídy MathLogic. V tomto případě DI předává zodpovědnost za vytvoření obslužní třídy mimo třídu klienta. V následujících bodech jsou uvedeny 3 způsoby, jak lze dosáhnout injektáže.

- Injektáž pomocí konstruktoru třídy
- Injektáž pomocí vlastnosti třídy
- Injektáž pomocí metody

V příkladu na obrázku 11 je použita injektáž pomocí konstruktoru. Třídy MathFactory již není potřeba, a proto je odstraněna. Vytvoření objektu MathOperation a stavení závislosti je nastaveno ve třídě MathService. O implementaci této třídy se již postará IoC container, což je Framework určený k automatické implementaci Dependecy Injection. V této diplomové práci se pro vytvoření REST API používá softwarový framework .Net Core, který má již vestavěný IoC container. O .Net Core více v následující kapitole. [17]

Obrázek 11 – Injektáž pomocí konstruktoru

```
public class MathLogic
{
    private IMathOperation mathOperation;

    0 references
    public MathLogic(IMathOperation mathOperation)
    {
        this.mathOperation = mathOperation;
    }

    0 references
    public int AddNumbers(int number1, int number2)
    {
        return mathOperation.AddNumbers(number1, number2);
    }
}
```

Zdroj: vlastní zpracování [17]

3.3 Technologie a nástroje

3.3.1 .Net Core

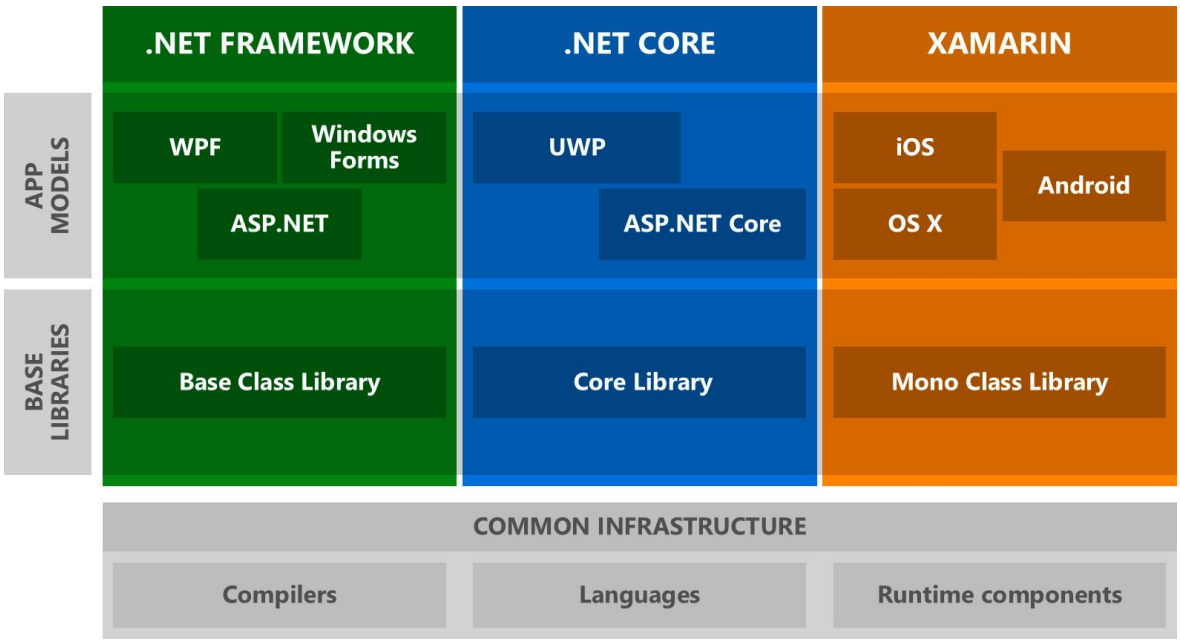
.Net Core je bezplatný open-source, který je primárně vyvíjen společností Microsoft. Hlavním cílem pro jeho vytvoření byla možnost tvorby aplikací bez závislosti na platformě, vytvoření modulární architektury, která je jednodušší na údržbu a mít platformu, která více zohledňuje moderní trendy jako je vývoj klient-server aplikací nebo nasazení do cloudu (např. Microsoft Azure). V dnešní době jsou podporovány Windows, MacOS a některé distribuce Linuxu. Mezi programovací jazyky, které jsou podporovány, patří C#, F#, částečně Visual basic a C++ v poslední verzi (.Net Core 3.1) a pouze

na Windows. První verze .Net Core 1.0 vyšla v červnu v roce 2016. V době psaní diplomové práce je aktuálně vydaná verze .Net Core 3.1, která vyšla v lednu v roce 2020. Příští verze, zatím pojmenovaná .NET 5, je naplánována na listopad 2020. Spolu s .Net Corem vyšlo nové vývojové prostředí (IDE – Integrated Development Enviroment) Visual Studio Code, které je rovněž určeno pro ostatní informační systémy a je volně použitelné. V následujících bodech jsou popsány nejdůležitější funkce .Net Coru.

- **Hosting** (hostování) Hostování je základní vlastností technologie ASP.NET Core a je jádrem každé serverové aplikace. Hostitel aplikace funguje jako kontejner a je zodpovědný za správu životnosti aplikace. Hostitel také obsahuje prostředí konfigurace a servery pro zpracování požadavků. Z pohledu REST tedy hostitelé a servery splňují omezení architektury Klient Server.
- **Middleware** (mezivrstva) funguje skvěle s omezením vrstvového systému. Celkově je architektura požadavek a odpověď řízena mezivrstvami, kde každá mezivrstva zachytí požadavek nebo odpověď, provede specifickou logiku a pokud nezastaví proces, pošle další mezivrstvě.
- **Dependecy injection** (vkládání závislostí) je používán k celkové lepší udržovatelnosti a menší závislosti systému. Dále pak pro lepší testovatelnost. Více o návrhovém vzoru v kapitole 3.2.
- **Configuration** (konfigurace) je jedna z unikátních vlastností. Umožňuje číst z konfigurace během runtime režimu. Konfigurace může být čtena z různých zdrojů, jako jsou soubory, příkazový řádek, proměnné prostředí, nahranná v paměti, šifrovaná v utajeném úložišti nebo přes vlastního poskytovatele.
- **Logging** (logování) je možné logovat rozdílnou závažnost chyb do konfigurovatelných destinací.

Na obrázku 12 je znázorněn .Net ekosystém a místo, ve kterém leží .Net Core. Pro vývoj serverových aplikací v .Netu je možné používat zatím více známý .Net Framework, avšak nové aplikace, které chtějí být multiplatformní, musí být vyvíjeny v Net Coru. [18]

Obrázek 12 – .Net Ekosystém

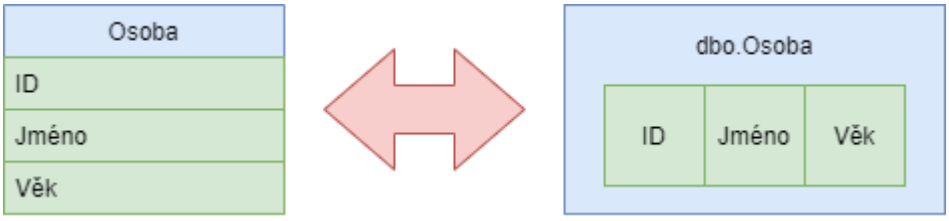


Zdroj: [19]

3.3.2 ORM Hibernate

Objektově relační mapování (ORM - Object-Relational Mapping) je programovací technika pro převod dat mezi systémy nekompatibilního typu v objektově orientovaných programovacích jazycích. Znamená to, že je možné mapovat objekty v kódu aplikace na entity relační databáze. Lze tedy komunikovat s databází pomocí programovacího jazyku, ve kterém je zároveň psána i aplikace, viz příklad na obrázku 13, kde je mapována tabulka Osoba ze schématu dbo a atributy ID, Jméno a Věk na třídu Osoba s vlastnostmi ID, Jméno a Věk. [20]

Obrázek 13 – Objektově relační mapování



Zdroj: vlastní zpracování

NHibernate je bezplatný open-source framework určený pro objektově relační mapování v prostředí .NET. Je postavený nad ADO.NET. Jedná se o port Hibernaty, který je určený pro Javu. První verze vznikla v roce 2005 (v době diplomové práci existuje verze 5.2.6).

Pro používání NHibernaty je ji potřeba stáhnout jako Nuget balíček a vytvořit konfigurační soubor. Aplikační kód používá pro persistenci dat rozhraní ISession a IQuerry, pro řízení transakcí rozhraní ITransaction. Pro mapované objekty je potřeba vytvořit tzv. HBM soubory, kde se nakonfiguruje mapování objektů databáze na objekty v aplikaci. Třídy, které jsou mapované, musí mít všechny vlastnosti jako veřejné a virtuální.

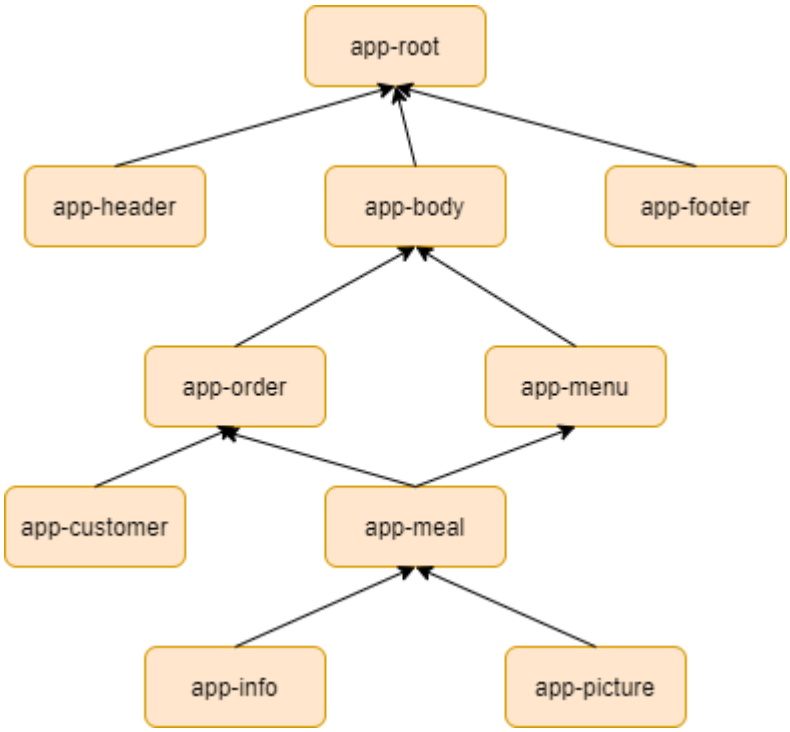
3.3.3 Angular

Angular je platforma určená pro webový vývoj, která poskytuje vývojářům robustní nástroje pro klientské webové aplikace. První verze Angular vyšla v roce 2010 a jmenovala se Angular JS a primárně byla zamýšlena na vývoj tzv. Single Page aplikací (aplikací, kde první načtení trvá déle, protože se stahuje celá stránka, ale poté už si vyměňuje se serverem pouze data). Jak vzrostla popularita v Single Page aplikacích, vyšel Angular v nové verzi, konkrétně Angular 2, která již nebyla kompatibilní s původním Angularem JS. Rozdíl zde byl mimo jiné i jazyk, na kterém je Anuglar postavený. V první verzi se používal JavaScript, od druhé již TypeScript. V době psaní diplomové práce aktuálně vychází Angular 9. Všechny verze od 2 jsou již kompatibilní a bez větších obtíží je vždy možné migrovat na novější verzi. Proto se také nový Angular častokrát označuje jako Angular 2+.
V následujících bodech jsou shrnuty výhody používání Angularu pro vývoj klientských webových aplikací. [21]

- Je podporován společností Google, což z něj dělá velmi důvěryhodnou platformu. Společnostem a vývojářům se vyplatí investovat čas i peníze do Angularu, protože věří, že platforma nějakou dobu vydrží. (Ve světě nynějších tzv. Javasriptových frameworků, kde jich existuje nepřeberné množství, je podpora technologických společností to nejdůležitější.)
- Je detailně zdokumentována, vývojáři mohou najít všechny potřebné informace a přicházet tak rychle s technickým řešením a řešit případné vznikající problémy.
- Angular je komponentově založená architektura, což znamená, že je aplikace rozdělaná na nezávislé, logické a funkční komponenty. Jednoduše tak mohou být komponenty nahrazeny nebo znovu použity. Komponenty jsou také dobře testovatelné díky tomu, jak jsou nezávislé a je tak snadné zjistit, že bezchybně fungují.

- Ahead-of-time kompilátor, který funguje tak, že vezme veškerý TypeScript a HTML a konvertuje ho to do JavaScriptu při sestavení aplikace. Díky zkompilování kódu předtím, než si webový prohlížeč načte webovou aplikaci, je renderování rychlejší. AOT kompilátor je také více bezpečný než klasický Just-In-Time kompilátor.
- Dependency injection má stejné výhody jako v jiných programovacích jazycích. Viz například v .Net Coru v předešlé kapitole.
- Příkazový řádek Angularu je velmi oblíbený nástroj mezi vývojáři. Automatizuje celý vývojový proces, což usnadňuje inicializaci, konfiguraci a vývoj aplikací. Příkazový řádek umožňuje vytvořit nový projekt, přidat do něj funkce, komponenty, direktivy atd. a spustit testy jednotek a koncových zařízení pomocí několika jednoduchých příkazů. [22]

Obrázek 14 – Komponentová architektura



Zdroj: vlastní zpracování

3.3.4 Bootstrap

Bootstrap je CSS Framework určený pro vývoj responzivních a mobilních webových stránek. Bootstrap je open-source sada nástrojů, která používá HTML, CSS a

JavaScript. Umožňuje velmi rychle vyvinout prototyp nebo komplexní aplikaci. První verze byla vydána v roce 2011. V době diplomové práce je Bootstrap ve verzi 4.4. [23]

3.3.5 MS SQL

SQL Server je systém správy relačních databází vyvinutý společností Microsoft. Podobně jako ostatní databázové systémy je Server postaven na SQL, což je standardizovaný strukturovaný dotazovací jazyk pro práci s relačními databázemi. SQL server je vázán na Transact-SQL (T-SQL), což je jazyk vyvinutý společností Microsoft, který přidává další funkčnosti. [24]

4 Vlastní práce

Praktická část diplomové práce je rozdělena na 4 kapitoly. V 1. kapitole je analýza se stručným popisem aplikace, požadované funkce a chování aplikace, dále pak je navržen diagram užití se stručným popisem jednotlivých případů. V následující kapitole je představeno ukládání dat a logický databázový model. Ve 3. kapitole je serverová část, kde jsou stručně popsány jednotlivé části, jako základní funkčnost, persistence dat pomocí NHibernate, ukázka implementace rezervace místa, úlohy na pozadí, logování a bezpečnost. V poslední kapitole je představena klientská část aplikace, hlavně popis obrazovek a jejich funkčnosti.

4.1 Analýza

V životním cyklu vývoje softwaru se nejprve deklarují požadavky a poté probíhá návrh aplikace. Vše vychází z popisu očekávaného a chtěného chovaní aplikace. K návrhu modelů analýzy je použit volně dostupný software Drawio, který umožňuje ukládání v reálném čase do Google Drive, One Drive nebo do paměti počítače.

4.1.1 Nápad, cíl a popis aplikace

Nápad ovlivnily hlavně 2 věci, které spolu z části souvisejí. V době nápadu na aplikaci je zavedeno EET a dochází ke zdražování produktů v gastronomii. Je tedy potřeba zkusit najít možnost, jak být ještě více efektivní v nákladech a zkusit přilákat nové zákazníky. V závislosti na poloze restauračního zařízení (dále jen restaurace) jsou ovlivněny tržby stravováním v době obědů. Na většině základních a středních škol je nutné objednávat oběd den dopředu, aby se vědělo, kolik připravit porcí. Zde vnikl nápad umožnit to i restauracím. Čím lepší budou mít restaurace odhad, kolik prodají obědů následující den, tím lépe dokáží optimalizovat náklady. Motivací pro zákazníky přistoupit na objednávkový systém může být věrnostní program, poskytnutý nápoj či polévka k jídlu zdarma nebo určitá procentní sleva. Lidé mají velmi rádi slevy či bonusové programy, a proto se rádi přidají. To může vézt k dalšímu navýšení počtu vydaných jídel v době obědů. V neposlední řadě to může pomoci s plýtváním nespotřebovaného jídla a jeho následného vyhazování. Problémem existence této funkčnosti je cena softwaru, který si majitelé restaurací nemohou nebo nechtějí dovolit. Řešením situace je vyvinout software, který bude volně dostupný. K nalezení motivace používat tento software je potřeba vyvinout

nejen systém na objednávky obědů, ale vytvořit komplexní software, který umožní uživatelům používat všechny běžné funkce, které jsou spojené s restaurací. Mezi základní funkce restaurace patří například rezervace místa (ideálně tzv. na jedno kliknutí, bez použití telefonu), dále pak umožnit zákazníkům zaregistrovat svoji e-mailovou adresu k odběru newsletteru v podobě zasílaní denního menu, chystaných akcí nebo speciálních nabídek. Mít možnost prezentovat názory zákazníků, tedy jejich hodnocení. Umožnit řídit obsah webu netechnickým uživatelům, a hlavně ho řídit rychle, což znamená plnou kontrolu obsahu majitelem, tedy administrace. Mít rozdělenou administraci webu pro určité role, například šéf kuchař řídí obsah jídelníčku, majitel pak otevírací dobu nebo již zmíněné hodnocení od zákazníků. Je tedy potřeba dodat jeden software, jedno řešení pro různé weby, a proto se nabízí použití architektury klient-server. Jednotlivé restaurace mají jiný vzhled a jsou napsány v různých programovacích jazycích. Pro serverovou část je tedy zvolena architektura REST. Klientská část může využívat funkčnosti, které budou potřeba, a vývoj softwaru se při tom nezdraží. Vývojáři dostávají tímto do ruky možnost, jak rychle vytvořit serverovou část a možnost soustředit se na vývoj klienta. Další výhodou je možnost vytvoření webu a pro administraci mít řešení v podobě oddělené aplikace. Hlavní cílem diplomové práce je tedy vytvoření open-source softwaru pro restaurační prostředí. Dále je implementována ukázková klientská aplikace, která bude demonstrovat možnosti využití API vytvořené serverové části.

4.1.2 Požadované funkce

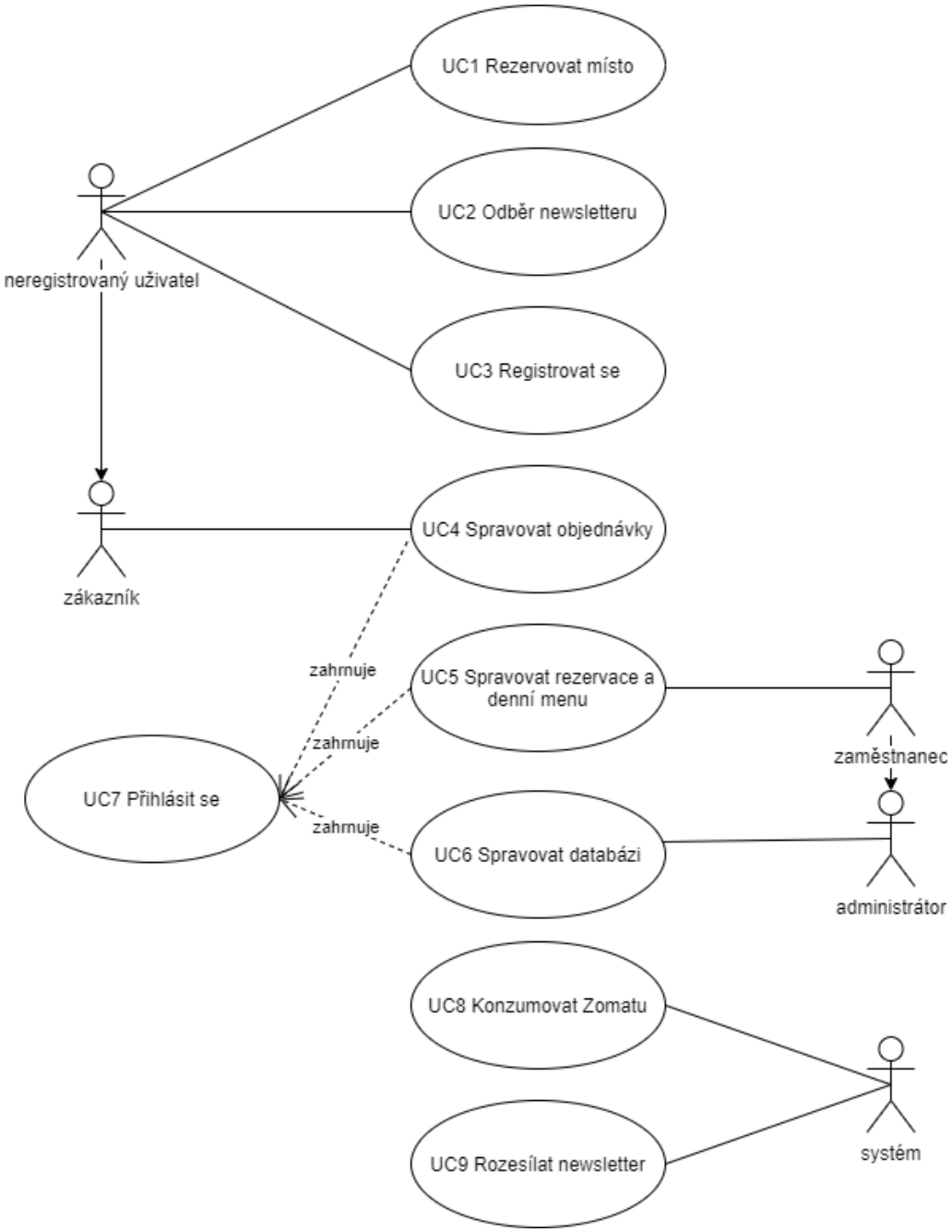
V předešlé kapitole, kde je představen návrh aplikace, jsou vytvořeny konkrétní požadavky neboli funkčnost systému. Výčet níže shrnuje požadované funkce.

- Rezervace místa na klik s potvrzením
- Přihlášení se k odběru newsletteru a jeho pravidelné zasílání
- Registrace uživatelů
- Vytvoření a následné spravování objednávky
- Administrace databázových položek v uživatelském prostředí
- Bezpečné přihlašování
- Pravidelné konzumování Zomatu API pro získávání hodnocení

4.1.3 Diagram užití

Diagram užití (Use Case Diagram) představuje chování systému z pohledu jednotlivých uživatelů. Z důvodů identifikace jednotlivých případů užití je použito pojmenování UC + jednoznačné číslo. Vyplněné šipky u aktérů znamenají dědičnost funkcionality. Čisté čáry jsou spojení, které představuje aktéra, který může inicializovat případ užití. Čárkovaná šipka značí, že případ užití obsahuje jiný případ užití.

Obrázek 15 – *Diagram užití*



Zdroj: vlastní zpracování

4.1.4 Specifikace případu užití

Diagram užití obsahuje hlavní funkce systému a jejich aktéry. Specifikace jednotlivých případů obsahuje navíc vstupní podmínky, které jsou nezbytné pro spuštění daného případu užití. Dalšími prvky jsou základní a alternativní scénář. Ve scénářích je popsán základní průchod případu užití, v tomto případě se jedná o interakci mezi aktérem, klientskou částí (označeno FE – Frontend) a serverovou částí (označeno BE – Backend). Chyby se můžou uvést v rámci alternativních scénářů nebo v samostatné kolonce výjimky, ovšem v této diplomové práci je použit alternativní scénář pouze pro specifický defekt nebo výjimku. Obecné chyby, které mohou nastat během scénářů, a tedy v běhu aplikace jsou v samostatné tabulce, aby se pokaždé neopakovaly, tj. platí pro všechny případy užití. Mezi takové chyby bude patřit: spadnutí serveru, chyba během databázové transakce, neočekávaná chyba a validace atributu v požadavku. Validace jsou součástí i na straně klienta, kde se jedná o Javascript, který je vždy možné upravit, a proto musí probíhat i validace požadavku nejen na straně serveru. V následující tabulce jsou popsány specifikace případu užití.

Tabulka 3 – Specifikace případu užití

Diagram užití	UC1 – Rezervovat si místo
Aktér	Neregistrovaný uživatel, zákazník
Vstupní podmínky	Nastavení Smtp klienta
Základní scénář	1. Aktér vyplní na záložce rezervace formulář a potvrdí. 2. FE odešle validní formulář metodou POST api/reservation. 3. BE načte volné stoly v požadovaném čase. 4. BE uloží rezervaci, odešle potvrzovací e-mail a vrátí odpověď, že je rezervace úspěšně provedena. 5. FE zobrazí notifikaci s výsledkem rezervace.
Alternativní scénář	4.1 BE vrátí odpověď, že je restaurace plná. 4.2 BE vrátí odpověď, že je v restauraci místo, ale ne u jednoho stolu, je potřeba provézt rezervaci telefonicky.
Diagram užití	UC2 – Odběr newsletteru
Aktér	Neregistrovaný uživatel, zákazník

Vstupní podmínky	-
Základní scénář	1. Aktér vyplní na záložce novinky jednořádkový formulář. 2. FE odešle validní formulář metodou POST api/home/nesletter. 3. BE vyhledá e-mail v databázi. 4. BE uloží e-mail a vrátí odpověď, že byl e-mail úspěšně přidán do newsletteru. 5. zobrazí notifikaci s výsledkem, že e-mail byl úspěšně přidán do newsletteru
Alternativní scénář	4.1 BE vrátí odpověď, že email má již aktivovaný newsletter. 4.2 BE vrátí odpověď, že e-mail byl v newsletteru znovu aktivován.
Diagram užití	UC3 – Registrovat se
Aktér	Neregistrovaný uživatel, zákazník
Vstupní podmínky	-
Základní scénář	1. Aktér vyplní na záložce uživatel formulář a potvrdí. 2. FE Odešle validní formulář metodou POST api/user/registration. 3. BE podle e-mailu ověří, jestli uživatel již není registrován. 4. BE uloží nového uživatele a vrátí odpověď, že je možné se již přihlásit. 5. FE zobrazí notifikaci s výsledkem.
Alternativní scénář	4.1 BE vrátí odpověď, že uživatel je již zaregistrován.
Diagram užití	UC4 – Spravovat objednávky
Aktér	Zákazník
Vstupní podmínky	Být autentizovaný a autorizovaný, metoda GET api/order musí vrátit jídla k objednání
Základní scénář	1. Aktér vyplní na záložce uživatel objednávku a potvrdí ji. 2. FE Odešle validní formulář metodou POST nebo PUT api/order.

	3. BE vytvoří objednávku. 4. FE zobrazí notifikaci s výsledkem.
Alternativní scénář	3.1 BE upraví existující objednávku.
Diagram užití	UC5 – Spravovat databázi
Aktér	Administrátor
Vstupní podmínky	Být autentizovaný a autorizovaný
Základní scénář	1. Aktér provede CRUD akci nad daty z databáze. 2. FE odešle danou HTTP metodu. 3. BE provede zápis do databáze a odešle odpověď. 4. FE zobrazí notifikaci s výsledkem.
Diagram užití	UC7 – Přihlásit se
Aktér	Zaměstnanec, zákazník, administrátor
Vstupní podmínky	Být registrovaný
Základní scénář	1. Aktér vyplní na záložce uživatel přihlašovací formulář a potvrdí. 2. FE odešle validní formulář metodou POST api/user/login. 3. BE vygeneruje validační token a vrátí ho v odpovědi. 4. FE zobrazí sekci podle role.
Alternativní scénář	4.1 BE vrátí autorizační chybu. 4.2 FE zobrazí, že uživatel zadal špatný e-mail nebo heslo.

Zdroj: vlastní zpracování

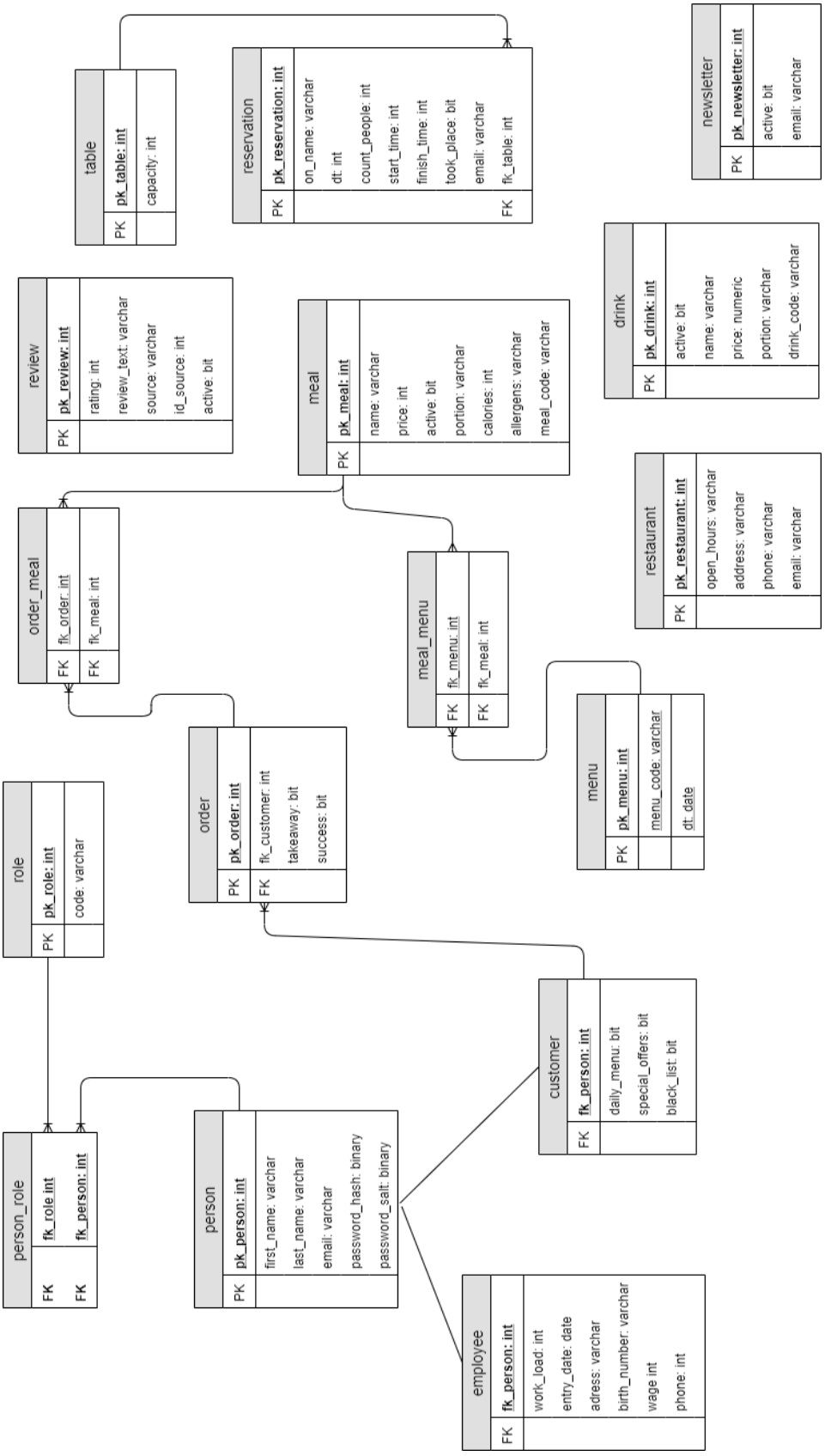
Případ užití 6, kde mají přístup zaměstnanci a administrátoři, poskytuje CRUD operace, ale pouze na datech ohledně denního menu a rezervací. Dále zde pak nejsou podrobné případy užití 8 a 9, kde je aktér systém a nedochází k žádné interakci. V případech, kde je aktér systém, tak se jedná o úlohy na pozadí (více v kapitole Serverová část), kde jsou hlavní vstupní podmínky: správná konfigurace časování a v případě konzumace Zomatu získaný API klíč a id zaregistrované restaurace. V případě konzumování dalších webových aplikací, kde je základní scénář podobný, tj. podle konfigurace načasovaní pro úlohu

na pozadí zavolání metodou GET API a následné uložení získaných dat do databáze, je pouze potřeba správně vždy vyplnit konfigurace a konkrétní zpracování dat podle daného API.

4.2 Ukládání dat

Návrh databáze vychází z návrhu požadavků na aplikaci. Jak již je zmíněno v kapitole 3.3.5, jedná se o relační databázový systém Microsoft SQL Server a relační databázi. Logický model databáze, znázorněn na obrázku 16, se skládá z 16 tabulek a je připraven na rozšiřování ať už o další sloupce, tak i o další tabulky. Model nedodržuje plně normativní formy zejména z důvodu optimalizace a lepší práce s databází. Například tabulka Drink obsahuje atribut drink_code, ve kterém se hodnoty budou opakovat např. beer (pivo), wine, (víno), tea (čaj) atd. Pokud by databáze měla dodržet úplně všechny normativní formy, bylo by potřeba tabulku dekomponovat a mít stávající tabulku Drink, která by místo atributu drink_code obsahovala například fk_drink_code, kde by byl uložen cizí klíč, který by odkazoval na nově vzniklou tabulku, kde by byly obsaženy všechny kódy nápojů právě jednou. Rozdílná situace je v tabulce person, která je ve vztahu M:N k tabulce role. Zde je potřeba mít tabulku na kódy rolí zvlášť, protože jednotlivé osoby mohou mít více rolí. Momentálně se v aplikaci počítá se třemi rolemi, a to zákazník, zaměstnanec a administrátor, tudíž by nestačila jedna role na uživatel. Pokud by se ale v budoucnu aplikace rozšiřovala, a tím pádem i model, třeba role kuchaře, který by měl na starosti jídelníček místo zaměstnance, tak už by bylo více komplikované upravovat model databáze, ORM NHibernatu a kód v aplikaci. V tomto směru je navrhnut model databáze a i celá aplikace. Aplikace je připravená na přidávání funkčnosti bez většího zásahu do stávajícího modelu a kódu. Bližší vysvětlení významu tabulek a sloupců je u služeb a komponent, které s nimi pracují. Databázový skript na vytvoření modelu databáze je součástí adresáře Databáze na GitHubu. Je možné k aplikaci vytvořit i jiný databázový systém, ovšem musí být relační, a je potřeba zachovat stávající tabulky, sloupce a jejich typy. Je potřeba si dát pozor zejména na datové typy sloupců, které se mohou mírně lišit. Při možné úpravě typů, je potřeba vše zohlednit i v mapovacích souborech a konfiguračním souboru NHibernaty.

Obrázek 16 – Logický model



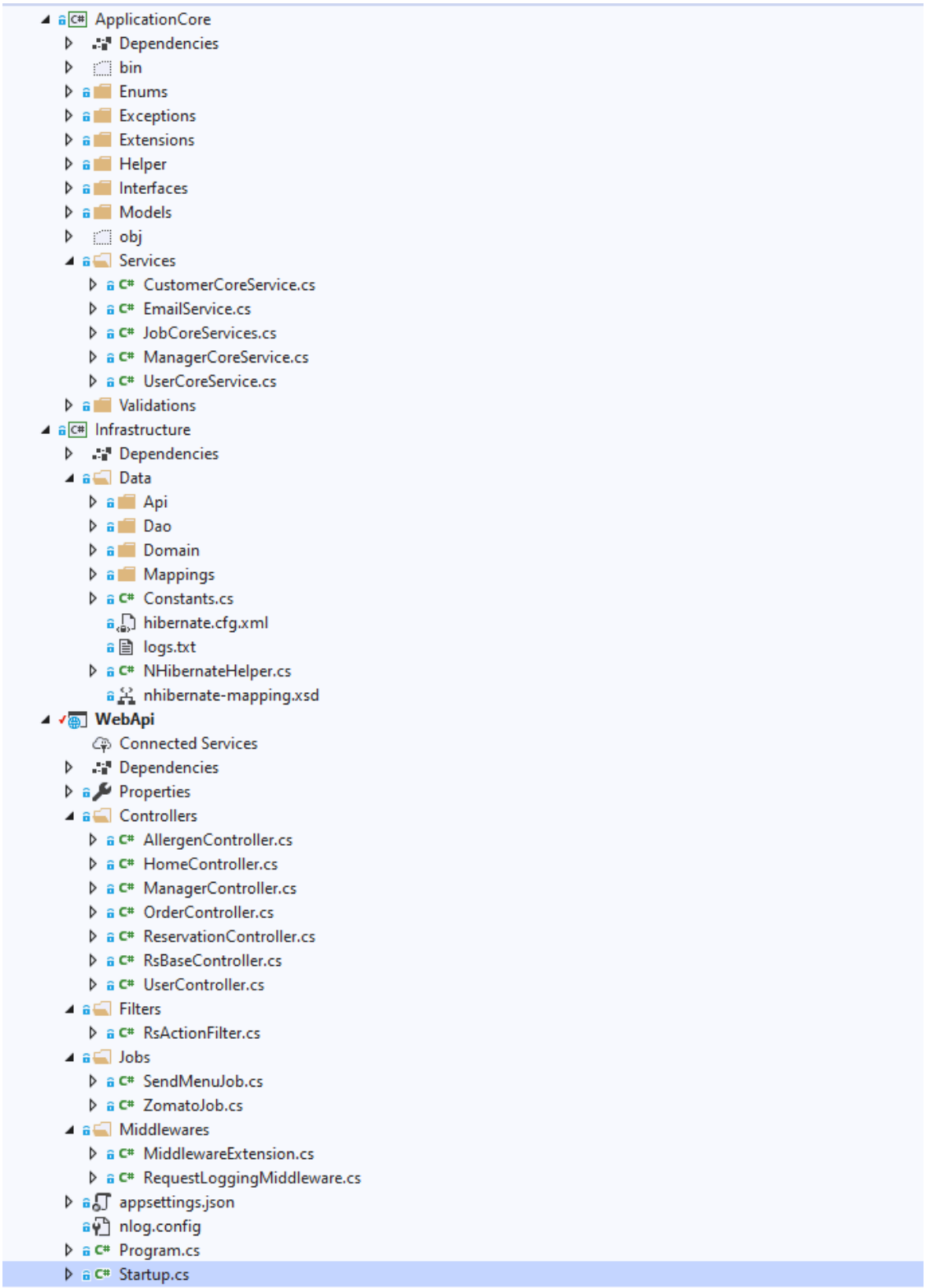
Zdroj: vlastní zpracování

4.3 Serverová část

Serverová část je napsána v .Net Core 3.0. Aplikace psané v Net. Core obsahují vždy 1 „solutionu“ (český výraz řešení není vhodný) a neomezené množství projektů (maximální doporučení od Microsoftu je 100). Jak je vidět na obrázku 17, aplikace používá organizaci aplikační logiky, tzn. rozdělení na 3 vrstvy: uživatelské rozhraní, obchodní logika a přístup do databáze. Těmto 3 vrstvám odpovídají 3 projekty: WepApi, ApplicationCore a Infrastructure. V tomto projektu patří do uživatelského rozhraní vystavení REST API. Pro vystavení REST API se v .Netu používají jednotlivé „kontrolery“ (Controllers nevrací html dokumenty, ale data ve formátu JSON). Kromě kontrolerů obsahuje projekt věci, které souvisejí s vystavením služeb, přímo tedy to, co souvisí s životním cyklem požadavků a odpovědí. Dále jsou zde umístěny úlohy na pozadí a mezivrstvy. Projekt WebApi je zároveň startovacím projektem celé „solutiony“, tzn., že obsahuje nezbytnou metodu Main ke spuštění. Dále jsou zde konfigurace logovaní a konfigurace celé aplikace. Dalším projektem je ApplicationCore, který zastává vrstvu obchodní logiky a u většiny případů funguje jako spojovací vrstva mezi uživatelskou a databázovou s provedením určité logiky, která je umístěna ve složce Services (služby). Nedílnou součástí je složka Interfaces (rozhraní), pomocí které je možné tyto služby injektovat, tam kde je potřeba. Ve většině případech v aplikaci se používá injektáž pomocí konstruktoru třídy. V projektu je dále i složka Domain, která obsahuje doménové modely tříd, které fungují převážně jako DTO (datové transferové objekty), aby se napřímo nepracovalo v modelu z databáze před vykonáním určité obchodní logiky. Složka Domain dále obsahuje modely požadavků a odpovědí, které by měly spíše patřit do projektu WebApi, ale zde by se naráželo na problém s cyklickou referencí, protože projekt WebApi již má referenci na projekt ApplicationCore, není tedy možné, aby ApplicationCore měl referenci na projekt WebApi. Problém by se dal vyřešit, ale znamenalo by to přidat více tříd a více kódů do projektu WebApi, což z hlediska menšího projektu není žádoucí. Posledním projektem je Infrastructure, kde hlavní záměr je přístup do databáze. Projekt obsahuje vše, co souvisí s používaným objektově relačním mapovacím frameworkem NHibernate, více v kapitole o fungovaní NHibernate. Každý projekt obsahuje kromě referencí (závislostí) na jiné projekty také přidané balíčky jak od Microsoftu, tak i od třetích stran. U každého balíčku je vidět před instalací, jestli je kompatibilní s aktuální

verzí .Net Coru., pokud není, tak se nenainstaluje. U balíčků je zejména potřeba dávat pozor v momentu, kdy se přechází na vyšší verze .Net Coru.

Obrázek 17 – Adresářová struktura



Zdroj: vlastní zpracování

4.3.1 Základní nastavení a funkčnost

V této kapitole jsou popsány nezbytné úkony k nastartování aplikace a její funkčnosti. Jak již je zmíněno v předešlé kapitole, aplikace se vždy spouští statickou metodou Main. Metoda je umístěna ve třídě Program a její implementace provede vytvoření webového Host (serveru), konfiguraci logování a konfiguraci hosta. V aplikaci je zatím nastaveno defaultní vytvoření hosta, které má za následek vytvoření webového serveru Kestrel a použití konfigurace appSettting.json. Webový server Kestrel má výhodu oproti tradičnímu IIS (Internet Information Services), že je multiplatformní. Určený je hlavně pro aplikace, které nejsou zamýšleny jako podnikové, ale jsou to malé jednoduché aplikace, kde nejsou potřeba další funkce tohoto serveru a dalších rozšiřitelných modulů. Při vytváření Serveru se rovněž volá důležitá generická metoda UserStartup, která není povinná, ale pro lepší přehlednost a údržbu je vhodnější udržovat celkové nastavení aplikace v jiné třídě, konktrétně Startup.

Ve třídě Startup jsou metody Configuration a ConfigureServices. Celkově se zde přidává funkčnost, nastavení a přidávají se další mezivrstvy. Nejprve je ovšem potřeba jednotlivé mezivrstvy zaregistrovat, což se dělá v první metodě. V metodě ConfigureServices, která se volá jako první v řadě, se registrují všechny závislosti do vestavěného IoC kontejneru. Například přidaní třídy, ve které se realizuje nastavení Smtp klienta a poslání e-mailu, vypadá následovně:

```
services.AddSingleton<IEmailService, EmailService>();
```

Jak ukazuje výpis, do kontejneru je přidána třída a její rozhraní a je s ní zacházeno jako s objektem, který existuje v aplikaci pro každý objekt či pro každý požadavek právě jednou. Další důležitou věcí je zaregistrování Cors, díky kterému je například možné ovlivnit, které URL mohou zavolat vystavené služby. Pokud se nejedná o veřejná API, je vhodné povolit pouze naši URL, kde je naše klientská aplikace. Dále se zde přidává registrování bezpečnosti, díky kterému se bude provádět autentizace na kontrolerech, které jsou označeny atributem Authorise. Je zde rovnou přidáno nastavení pro JSON Web tokens metodou Bearer. Je potřeba ještě zaregistrovat aplikační framework MVC, díky kterému je možné vystavit REST API. Zároveň je rovnou konfigurováno, aby byly logovány požadavky a odpovědi, více v kapitole o logováním. Pro účely testování vývojářem je

vhodné zaregistrovat Swagger, který umožní zdokumentovat všechny koncové body aplikace a v jeho uživatelském rozhraní je provolávat. Na obrázku 18 (nejsou zde vidět všechny zdroje) je vidět příklad Swaggeru pro tuto diplomovou práci. Swagger zároveň perfektně slouží k přehledu všech vystavených koncových bodů, tedy krásně vypadající dokumentaci.

Obrázek 18 – Swagger

Zdroj: vlastní zpracování

V druhé metodě Configure ve třídě Startup, jsou zaregistrované objekty nebo jejich vnitřní objekty většinou spuštěny metodou Use na daném objektu. Navíc jsou zde přidány mezivrstvy pro routování a vlastní mezivrstva pro logování požadavků a odpovědí a pro používání REST API. Pokud je potřeba mít různá nastavení pro rozdílné prostředí, je třída Startup ideální místo. Stačí injeknout pomocí metody prostředí webového hosta a na jeho vlastnosti už je možné zjistit aktuální prostředí.

Jak bylo zmíněno na začátku této kapitoly, při prvním spouštění se nastavuje konfigurace, která je obsažená v appsettinng.json, ve které jsou konfigurační věci určené k nastavení smtp klienta, nastavení pro komunikaci s veřejným Zomatu API a nastavení pro vytváří tokenů pro autentizaci.

4.3.2 Persistence dat pomocí NHibernate

Persistence dat probíhá v aplikaci pomocí ORM frameworku NHibernate. Ke správnému nastavení a fungování je potřeba stáhnout balíček (NHibernate). V případě diplomové práce je stažen balíček do projektu Infrastructure, který je určen jako datová vrstva aplikace. Pro správné fungování slouží vytvoření rozhraní ISessionFactory pomocí třídy Configuration a na ní zavolání metod Configure s parametrem cesty ke konfiguračnímu souboru NHibernaty a zavoláním BuildSessionfactory. V tuto chvíli je vytvořena Session mezi databázovým serverem a aplikací. Ve vytváření dochází nejprve k načtení konfiguračního souboru hibernate.cfg..xml viz obrázek 19, kde je nutné nastavit vlastnost dialect, která označuje databázový systém (ve vlastnosti je sice napsán MS SQL 12, ale ten perfektně funguje nejen pro verzi 12, ale i pro všechny nadcházející) a vlastnost connection string, kde se nastavuje název serveru, jméno databáze a metoda přihlášení. Toto nastavení patří mezi absolutní minimum, které je potřeba k fungování.

Obrázek 19 – Konfigurační soubor NHibernate

```
<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
  <session-factory>
    <property name="dialect">NHibernate.Dialect.MsSql2012Dialect</property>
    <property name="connection.connection_string">
      Data Source=DESKTOP-AA6E89D;Initial Catalog=restaurant;Integrated Security=True
    </property>
    <property name="current_session_context_class">web</property>
    <mapping assembly="Infrastructure" />
  </session-factory>
</hibernate-configuration>
```

Zdroj: vlastní zpracování

Po načtení konfiguračního souboru následuje volání pomocí již zmíněné metody BuldSessionfactory a dojde k mapování tabulek z databáze na objekty. Aby bylo jasné, jestli se v aplikaci pracuje s databázovým objektem, a ne běžným DTO, jsou všechny objekty, které představují tabulky databáze, pojmenovány s prefixem Bdo. Tyto objekty jsou uloženy do složky Domain v projektu Infrastructure. Jednotlivé mapování se řídí mapovacími soubory xml (není podmínka, může být nově i JSON), které obsahují instrukce, jak a co má být mapováno. Tyto soubory musí být pojmenovány následovně:

<název>.hbm.xml

V diplomové práci je tak například pojmenován soubor, který mapuje tabulku Person:

BdoPerson.hbm.xml

Jak je vidět v logickém modelu databáze na obrázku 16, z tabulky Person dědí tabulka employee a customer. Zároveň je tabulka Customer ve vztahu 1:N s tabulkou Orders. Mapování je z databázového modelu nejsložitější, protože všechny tyto vztahy musí být v mapovacím souboru popsány, viz obrázek 20.

Obrázek 20 – Mapovací soubor pro BdoPerson

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" namespace="Infrastructure.Data.Domain" assembly="Infrastructure">
    <class name="BdoPerson" table="person">
        <id name="Id">
            <column name="pk_person" not-null="true" unique="true"/>
            <generator class="native"/>
          </id>

        <property name="FirstName" column="first_name"/>
        <property name="LastName" column="last_name"/>
        <property name="Email" column="email"/>
        <property name="Password" column="password"/>
        <property name="PasswordHash" column="password_hash" type="BinaryBlob"/>
        <property name="PasswordSalt" column="password_salt" type="BinaryBlob"/>

      <bag name="Roles" table="person_role">
        <key column="fk_person"/>
        <many-to-many class="BdoRole" column="fk_role" />
      </bag>

        <joined-subclass name="BdoEmployee" table="employee">
            <key column="fk_person"></key>
            <property name="WorkLoad" column="work_load"></property>
            <property name="EntryDate" column="entry_date" type="Date"></property>
            <property name="Address" column="address"></property>
            <property name="BirthNumber" column="birth_number"></property>
            <property name="Wage" column="wage"></property>
            <property name="Phone" column="phone"></property>
          </joined-subclass>

        <joined-subclass name="BdoCustomer" table="customer">
          <key column="fk_person"></key>
          <property name="DailyMenu" column="daily_menu" type="Boolean"></property>
          <property name="SpecialOffer" column="special_offer" type="Boolean"></property>
          <property name="BlackList" column="black_list" type="Boolean"></property>

          <bag name="Orders" table="order">
            <key column="fk_customer"/>
            <one-to-many class ="BdoOrder"/>
          </bag>
        </joined-subclass>

      </class>
  </hibernate-mapping>
```

Zdroj: vlastní zpracování

V elementu class je plněn název tabulky a název, na který objekt se mát tabulka mapovat. V elementu id, který je jako jediný sloupec povinný, je vyplněn název sloupce a

používaná funkce pro auto inkrement, v tomto případě je použito nativní, což znamená, že je ponecháno nastavení z databáze. Mapování sloupců je v elementu property, kde je nezbytné doplnit název vlastnosti třídy a název sloupce. Pro lepší přesnost mohou být uvedeny typy na obou stranách. Například v .Netu existuje datový typ DateTime a v MS SQL je to Date, DateTime, TimeStamp atd. Zde je proto vhodné vyplnit i typ sloupce a zajistit tak správnou konzistenci dat. Podle návrhu databáze může tabulka Person obsahovat n záznamů z tabulky Role a tabulka Role n záznamů z tabulky Person. Jedná se tedy o vztah M:N, který je zachycen v elementu bag, který značí pole nebo list prvků. V elementu jsou jako předtím určeny, které prvky sloupce jsou mapovány na které vlastnosti objektů. Dědičnost lze popsat více způsoby, zde je použito elementu joined-subclass, ve kterém je připojena, a tedy mapována, další třída na objekt. Vlastnosti jsou mapovány, jak již bylo popsáno a jako klíč je uveden cizí klíč na předka. Jelikož třída tabulka Customer může obsahovat více záznamů z tabulky Order, je zde obdobně použit element bag, jako když se mapovaly role. Stejným způsobem je namapována celá databáze. Pokud chceme, je možné mapovat jen některé tabulky či některé sloupce z databáze, vyjma primárního klíče. Do projektu je přidán soubor nhibernate-mapping.xsd, který slouží k popisu xml, což v našem případě znamená, že je dostupné intellisense v mapovacích souborech.

V relačních databázových souborech slouží k provádění operací s daty jazyk SQL. V případě NHibernate je více možností, jak přistupovat k operacím nad daty. Je možné použít jazyk SQL, dále tzv. HQL, což je hybridní dotazovací jazyk NHibernate, který je již v C#, ale struktura zápisu vypadá, jako by se jednalo o SQL, anebo používat čistě C# a jeho běžný zápis. Poslední způsob je aplikován i v diplomové práci. Než budou ukázány konkrétní dotazy, je potřeba založit třídy, díky kterým bude přistupováno do databáze a ve kterých budou implementovány dotazy. V dokumentaci NHibernate se označují tyto třídy postfixem Dao. V této aplikaci je nastavena konvence jako název objektu, ke kterému se přistupuje bez prefixu Bdo a s postfixem Dao. Například pro přístup k tabulce Person je používána třída:

PersonDao.cs

Všechny třídy, které přistupují do databáze a mají již zmíněný postfix, jsou uloženy ve složce Dao a jejich rozhraní ve složce Api, protože i zde je následně používáno

Dependency Injection. Každá dao třída má specifické dotazy nad tabulkou, ale má i dotazy, které jsou společné pro všechny objekty, je tedy vytvořena generická třída jako předek pro tyto dao třídy, která obsahuje implementaci základních CRUD operací. Na závěr podkapitoly ohledně NHibernate je na obrázku 21 znázorněno, jak vypadá dotaz vyhledání záznamu z tabulky Person podle e-mailu, který dědí z DaoBase obsahující CRUD operace a zároveň implementuje rozhraní IPersonDao. V metodě GetPersonByEmail, do které vstupuje parametr e-mail, je nad „sessinou“ proveden dotaz pomocí generické metody, ve které se určí typ objektu (tabulka) a pomocí LINQ je vyhledán v databázi záznam, ve kterém je atribut e-mail shodný s e-mailem (parametrem metody) a je vybrán maximálně jeden záznam.

Obrázek 21 – Třída PersonDao

```
namespace Infrastructure.Data.Dao
{
    public class PersonDao : DaoBase<BdoPerson>, IPersonDao
    {
        public BdoPerson GetPersonByEmail(string email)
        {
            return CurrentSession.QueryOver<BdoPerson>().Where(x => x.Email == email).SingleOrDefault();
        }
    }
}
```

Zdroj: vlastní zpracování

4.3.3 Web Api

K vystavení API slouží v .Net Coru kontrolery jako logické seskupení aplikace se sadou standartních akcí nebo koncových bodů. Poskytují infrastrukturu pro provádění akčních metod. Akce jsou pouze funkce nebo metody, které přijímají parametry, mohou provádět nějakou logiku a vracejí konkrétní výsledek, který je na konci přenesen do odpovědi. K využití této funkčnosti je potřeba pojmenovávat kontrolery jako:

<Název>Controller.cs

V diplomové práci je přidán předek RsBaseContoller pro všechny kontrolery, který zpřístupňuje logování a obecně je připraven na přidávání sdílené funkčnosti pro kontrolery. K nezbytnému fungování je ještě potřeba přidat atribut Route, který definuje cestu ke zdroji a nepovinný atribut ApiController, který přináší další automatizaci, jako například bindovaní těla požadavků a v případě nevalidního požadavku, automatické

odpovědi. Pokud by se atribut ApiController vynechal, znamenalo by to přidat ke každému tělo požadavku atribut a následné kontroly, jestli je validní. Na obrázku 22 je zobrazena třída HomeController, která používá zmíněné atributy a je potomkem třídy RsBaseController. Dále je zde vidět, jak je provedena injektáž třídy CustomerCoreService, ve které je implementace obchodní logiky, kterou kontroler využívá.

Obrázek 22 – Třída HomeController

```
namespace WebApi.Controllers
{
    [Route( template: "api/[controller]")]
    [ApiController]
    public class HomeController : RsBaseController
    {
        private readonly ICustomerCoreService _customerCoreService;

        public HomeController(ILoggerFactory loggerFactory,
            ICustomerCoreService customerCoreService)
            : base(loggerFactory)
        {
            _customerCoreService = customerCoreService;
        }

        action methods...
    }
}
```

Zdroj: vlastní zpracování

Jak již bylo napsáno v úvodu kapitoly, kontrolery seskupují akce, které se ve frameworku MVC nazývají akční metody (action methods). Akční metody představují HTTP metody, které jsou popsány v kapitole 3.1.3. Pro odlišení akční metody od běžné metody třídy je nezbytné použít atribut, který je intuitivně pojmenovaný. Například HTTP metodu GET označuje atribut HttpGet. Druhou důležitou věcí je ještě potřeba určit cestu k metodě, pokud je jiná než definovaná atributem u třídy kontroleru. Na obrázku 23 je vidět akční metoda AddEmail ve třídě HomeController, která je typu POST a je volána v rámci UC2. Akční metoda očekává parametr request typu NewsletterRequest, který obsahuje tělo požadavku a je znázorněn ve stejném obrázku pod akční metodou. Tělo požadavku obsahuje pouze jednu vlastnost Email, která je typu string. Důležité jsou zde atributy Required a EmailAdrress, které označují, že tato vlastnost je povinná a řetězec textu musí mít strukturu e-mailu. Bindování a následnou kontrolu validity obstarává již

zmíněný atribut ApiController. Aby bylo bidnování z JSON formátu úspěšné, je potřeba zachovat stejné pojmenovávání.

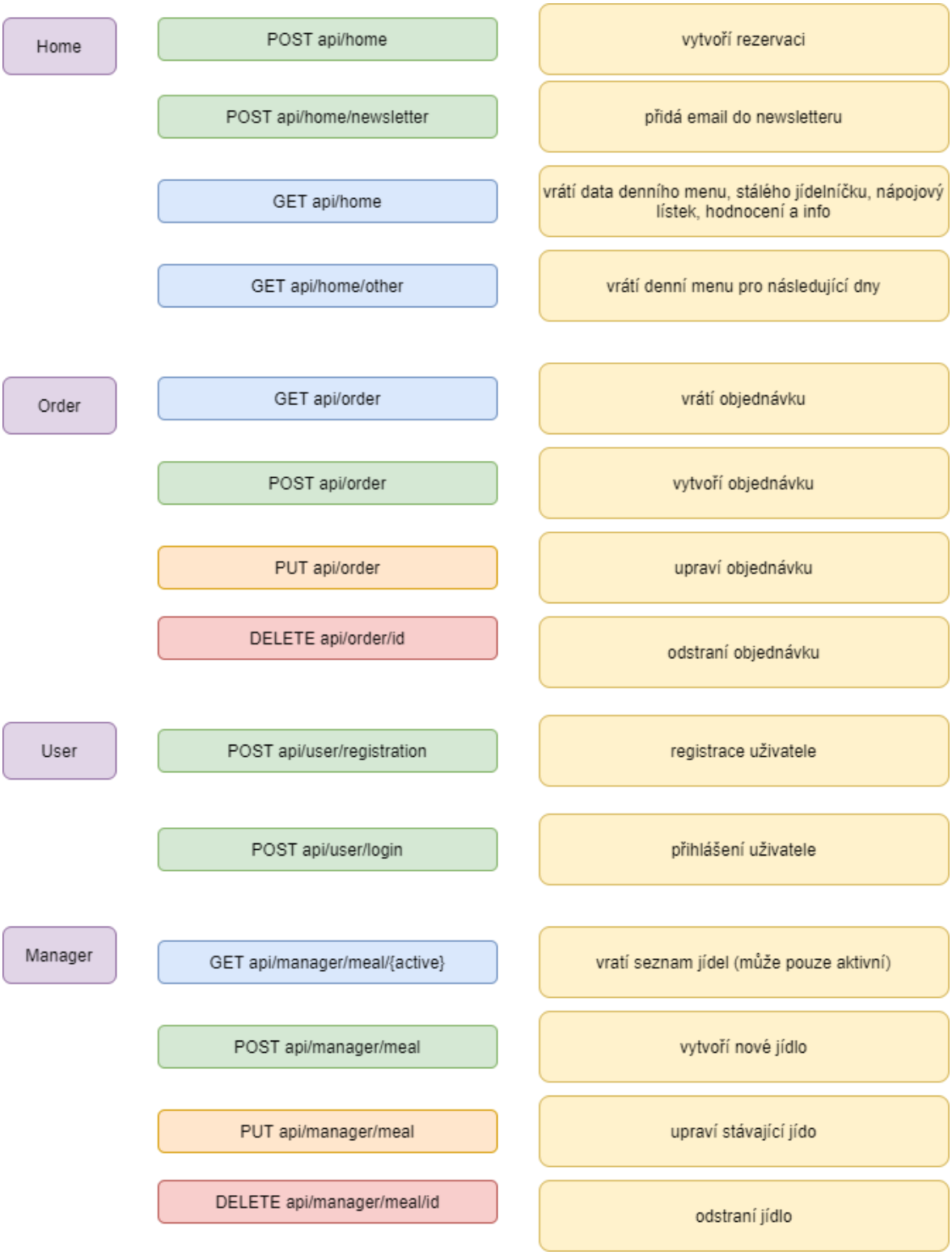
Obrázek 23 – Akční metoda AddEmail a třída NewsllterRequest

```
[HttpPost]
[Route( template: "newsletter")]
public IActionResult AddEmail(NewsletterRequest request)
{
    var response = _customerCoreService.AddEmailToNewslleter(request.Email);

    return Ok(response);
}


public class NewsletterRequest
{
    [Required, EmailAddress]
    public string Email { get; set; }
}
```

Zdroj: vlastní zpracování

Dosud vypsané atributy pro akční metody a kontrolery jsou využívané napříč celou aplikací. V aplikaci se dále vyskytují ještě dva atributy pro speciální případy. Prvním atributem je ResponseCache, kterému se nastavuje vlastnost Duration, která je typu int. Tento atribut je použit pouze u akční metody GetHomeData v HomeControlleru, u které se předpokládá, že bude nejvíce volána, obsahuje největší množství dat a data se zde budou pravidelně měnit pouze jednou denně v podobě změny denního menu. Tato metoda se volá vždy při načtení webového klienta, kde jsou data použita na úvodní obrazovce. Druhým atributem je Authorise, který je použit u OrderController a ManagerController a díky němu je každý požadavek s dotazem na zdroje v těchto kontrolerech podroben autentizaci. Nastavení bezpečnosti je popsáno v kapitole 4.3.6. V aplikaci existují kontrolery, které seskupují akční metody, které pak dále pracují s obchodní logikou. Obchodní logiky jsou obsaženy v projektu ApplicationCore. Pro lepší orientaci je na obrázku 24 popsána základní mapa kontrolerů, HTTP metody a jejich význam. U ManagerControlleru, který slouží hlavně ke správě položek v databázi, je na obrázku znázorněna akční metoda pro tabulku Meal. Vzhledem k rozsáhlé funkčnosti, není popsána detailně implementace všech akčních metod, ale pouze UC1 rezervace místa. Obdobným způsobem je provedena implementace v celé aplikaci.

Obrázek 24 – Základní mapa kontrolerů a akčních metod



Zdroj: vlastní zpracování

4.3.3.1 Implementace rezervace místa

V této části je popsána detailně rezervace místa od momentu zavolání a provedení akční metody AddReservation. Metoda přijímá objekt ReservationRequest, který obsahuje vlastnosti vypsané v následující tabulce.

Tabulka 4 – Vlastnosti objektu ReservationRequest

Validační atribut	**Název**	**Význam**	**Typ**
Required	OnName	Na jméno	String
Required, Range(1,20)	CountPeople	Počet lidí	Int
Required	StartTime	Začátek	Int
Required	FinishTime	Konec	Int
Required	Date	Datum	DateTime
Required	Email	E-mailová adresa	String

Zdroj: vlastní zpracování

Podle tabulky jsou všechny vlastnosti povinné a u vlastnosti CountPeople je vyžadováno, že počet lidí musí být v rozmezí 1-20. Zajímavostí je také typ u začátku a konce, který je integer, což značí celé číslo. Je to z důvodu mnohem lepší práce s celým číslem než s časovým razítkem. Ve vlastnosti tak znamená zápis 1730 čas 17:30 a podobně je to s vlastností konec akce. Je velmi obtížné udržet jeden formát datumu, který se tvoří v klientské části. V této diplomové práci je použit jazyk TypeScript, poté je v JSON poslán na server, kde probíhá bindování na typy jazyka C#, a následně probíhá mapování pomocí NHibernaty na databázové typy. Jak je vidět, probíhá dvojí konverze datumu. Konzistence dat by mohla být ohrožena. V tomto případě plně stačí, když klient pošle celé číslo, které symbolizuje čas.

Po úspěšném přijetí požadavku následuje zavolání stejnojmenné metody AddReservation ze třídy CustomerService, kde bude probíhat obchodní logika. Uvnitř metody se nejprve deklaruje a inicializuje odpověď (response). Následně je převeden i datum na celé číslo opět pro lepší práci s datumem. Tato konverze probíhá obdobně jako u začátku a konce, akorát s tím rozdílem, že je prováděna až na straně serveru pomocí statické metody TransformDatoToInt ve statické třídě Helper. V následujícím řádku jsou získány všechny rezervace stejného datumu jako je datum v požadavku a následně jsou ponechány jen ty, které jakkoliv zasahují do času plánované rezervace.

Obrázek 25 – Metoda AddReservation

```
public AddReservationResponse AddReservation(ReservationRequest reservationRequest)
{
    AddReservationResponse response = new AddReservationResponse();
    int reservationDate = Helper.Helper.
        TransformDateToInt(reservationRequest.Date.Date.ToString(format: "yyyy-MM-dd"));

    var reservations = _reservationDao.GetReservation(reservationDate)
        .NullToEmpty()
        .Where(x => x.StartTime >= reservationRequest.StartTime
                    || x.StartTime <= reservationRequest.FinishTime);

    var fullTables = reservations.NullToEmpty().Select(x => x.Table.Id).ToList();

    var freeTables = _tableDao.GetAll()
        .Where(x => !fullTables.NullToEmpty().Contains(x.Id)).ToList();

    if (freeTables.Any()
        && freeTables.Sum(x=>x.Capacity) > reservationRequest.CountPeople)
    {
        if (freeTables.Max(x=>x.Capacity) > reservationRequest.CountPeople)
        {
            BdoTable table;
            int i = 0;
            do
            {
                table =
                    freeTables
                        .FirstOrDefault(x => x.Capacity == reservationRequest.CountPeople + i);
                i++;
            } while (table == null);

            _reservationDao.Save(new BdoReservation
            {
                Date = reservationDate,
                CountPeople = reservationRequest.CountPeople,
                Email = reservationRequest.Email,
                OnName = reservationRequest.OnName,
                FinishTime = reservationRequest.FinishTime,
                StartTime = reservationRequest.StartTime,
                Table = table,
                TookPlace = true
            });
            var startTime = $"{reservationRequest.StartTime / 100} : " +
                            $"{(reservationRequest.StartTime % 100).ToString(format: "00")}  ";
            _emailService.SendEmail(
                message: $"Rezervace na jméno {reservationRequest.OnName} " +
                $"proběhla úspěšně, budeme se na Vás těšit " +
                $"{reservationRequest.Date.Date.ToShortDateString()} v {startTime}.",
                reservationRequest.Email);
            response.Result = "success";
            return response;
        }
        response.Result = "needCall";
    }

    else
    {
        response.Result = "fail";
    }
    return response;
}
```

Zdroj: vlastní zpracování

Tento dotaz je zavolán díky třídě ReservationDao, která je do třídy CustomerService injektována pomocí konstruktoru a metodě GetReservation s parametrem datumu. Následně jsou do proměnné fullTables (obsazené stoly) vybrány unikátní identifikátory, tj. id, které jsou použity v následujícím dotazu, kde se nejdříve načtou všechny stoly z tabulky Table a vyfiltrují se pomocí id stolů, které nejsou dostupné v tomto čase. Výsledek této operace je uložen do proměnné freeTables (volné stoly). Následně je potřeba zkontrolovat, jestli je dostupná kapacita stolů (každý stůl má v databázi i kapacitu) dostatečná pro potřeby rezervace. Pokud není, je metoda ukončena a vrátí odpověď s výsledkem fail (neúspěch). V případě, že je dostupná kapacita v restauraci, je následně potřeba zkontrolovat, jestli je dostupné místo u jednoho stolu nebo bude potřeba více stolů na rezervaci. V případě více stolu, je metoda opět ukončena s výsledkem needCall (potřeba zavolat), což znamená, že je potřeba konzultovat svoji rezervaci s personálem podniku, který může garantovat, že určené stoly bude možné spojit a vše i případně připraví pro zákazníka. V případě, že je vše v pořádku s kapacitou stolu, je potřeba provézt rezervaci stolu, který přesně odpovídá požadované kapacitě. Toto se provede pomocí cyklu while, který probíhá do doby, než kapacita stolu přesně odpovídá požadované kapacitě v rezervaci. Následně je rezervace uložena do databáze pomocí metody Save, která se nachází na předkovi třídy ReservationDao, tedy třídy DaoBase. Před ukončením metody a vrácením výsledku success (úspěch) proběhne zavolání metody SendEmail ze třídy Email Service. Tato metoda je používána k odeslání e-mailu a na vstupu má 2 parametry. Text zprávy a adresu kam se bude posílat e-mail. V těle zprávy je shrnuta požadována rezervace, je tedy vyplněno jméno, na koho je rezervace, datum a čas od kdy do kdy. Mezitím v databázi proběhl úspěšný zápis do tabulky reservation. Pokud by chtěl například pan Novák rezervaci pro 8 lidí v čase 18:30 až 21:00 na 15. června 2020, objevil by se v případě volna v restauraci řádek v následující tabulce (řádek by obsahoval i primární klíč, ale z důvodu místa není vypsán).

Tabulka 5 – Výpis řádku z tabulky reservation

on_name	count_people	start_time	finish_time	fk_table_code	email	dt
Novák	8	1830	2100	10	novak@seznam.cz	20200615

Zdroj: vlastní zpracování

4.3.4 Úlohy na pozadí

V aplikaci jsou nastavené funkčnosti UC8 a UC9 ve kterých dochází k úlohám, které běží na pozadí v určitých intervalech. Hlavní aktér je zde systém. Pro úlohy na pozadí je běžné používat anglické slovíčko job a podle toho jsou i třídy pojmenovány. Pro úlohy na pozadí je v .Net Coru nezbytné, aby se implementovalo rozhraní IHostedService nebo dědilo z abstraktní třídy BackgroundService, která již implementuje rozhraní IHosted Service. V serverové části aplikace jsou implementovány dvě úlohy na pozadí. První má za úkol posílat každý den denní menu a druhá úloha běží momentálně každé tři dny a její práce je konzumovat cizí veřejné API. V celé aplikaci to je momentálně jediný případ. Princip je podobný, ale jiný v tom, že teď nejsou zpracovávány požadavky a vytvářeny k nim odpovědi, ale sestavuje se požadavek a očekává se odpověď, která se následně zpracuje. V následující kapitole bude představena implementace UC8, která konzumuje služby webové aplikace Zomatu. Zomatu slouží k vyhledávání restaurací, psaní hodnocení, vyhledáváním jídelníčku apod. V případě aplikace je využito Zomatu pouze ke stažením dat, které se týkají hodnocení naší restaurace. K úspěšnému navázání kontaktu se Zomatu API je potřeba postupovat podle vývojářské dokumentace, kterou lze najít na webu https://developers.zomato.com/documentation?lang=cs. K získání hodnocení je volána HTTP metoda GET https://developers.zomato.com/api/v2.1/reviews a k tomu je nezbytné přidat v querry parametru zomato id restaurace a do hlavičky požadavku pak přidat API klíč. Oba dva parametry jsou dostupné po zaregistrování. Počet požadavků je omezen na 1000 denně. V případě vyšší potřeby je nutné si zaplatit vyšší počet požadavků, což při jednom volání za 3 dny není potřeba řešit. Získaná hodnocení mohou posloužit k vlastní prezentaci na svých stránkách a také k tomu, že máme pod kontrolou, co se kde o restauraci píše a je možné reagovat. Implementace se v aplikaci nachází ve třídě ZomatoJob. Kromě samostatné implementace třídy, je potřeba zaregistrovat službu ve třídě Startup v metodě ConfigureServices. Samostatná implementace je na obrázku 26.

Do třídy ZomatoJob jsou injektovány Logger (popsán v následující kapitole), HttpClient (zpřístupnění http infrastruktury), Configuration (zpřístupní konfigurace appsetting.json), JsonSerializer (umožní deserializovat tělo odpovědi) a JobCoreService (obchodní logika). Ve třídě je pouze jedna metoda, která je asynchronní a přijímá jediný parametr stopingToken typu CancellationToken. Implementování úloh na pozadí se vždy skládá z časovače a z cyklu while, který probíhá stále dokola.

Obrázek 26 – Třída ZomatoJob

```
public class ZomatoJob : BackgroundService
{
    private readonly ILogger _logger;
    private readonly HttpClient _httpClient;
    private readonly IConfiguration _configuration;
    private readonly JsonSerializer _serializer;
    private readonly IJobCoreServices _jobCoreServices;

    public ZomatoJob(ILogger<ZomatoJob> logger,  IConfiguration configuration,
        IHttpClientFactory httpClient, IJobCoreServices jobCoreServices)
    {
        _logger = logger;
        _httpClient = httpClient.CreateClient();
        _configuration = configuration;
        _jobCoreServices = jobCoreServices;
        _serializer = new JsonSerializer();
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        var zomatoSection = _configuration.GetSection( key: "zomato");
        var uri = $"https://developers.zomato.com/api/v2.1/reviews?res_id={zomatoSection["zomatoRestaurantId"]}";
        int days = Convert.ToInt32(zomatoSection["zomatoDay"]);

        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.Log(LogLevel.Information,  eventId: 2,  message: "Ordinary zomato job");

            try
            {
                _httpClient
                    .DefaultRequestHeaders
                    .Add( name: "user-key",  value: zomatoSection["zomatoUserKey"]);
                var response =
                    await _httpClient.GetAsync(uri);

                if (response.IsSuccessStatusCode)
                {
                    using var sr = new StreamReader(await response.Content.ReadAsStreamAsync());
                    using var jsonTextReader = new JsonTextReader(sr);
                    var zomatoReviews = _serializer.Deserialize<ZomatoReview>(jsonTextReader);

                    if (zomatoReviews.reviews_count > 0)
                    {
                        _jobCoreServices.UpdateZomatoReviews(zomatoReviews);
                    }
                }
                else
                {
                    _logger.Log(LogLevel.Error,  eventId: 4,  message: "faild communication with Zomato");
                }
            }
            catch (Exception ex)
            {
                _logger.Log(LogLevel.Critical,  eventId: 5, ex.StackTrace);
            }

            await Task.Delay(1000*60*24* days, stoppingToken);
        }
    }
}
```

Zdroj: vlastní zpracování

Na začátku je načtena zomato sekce z konfigurace, která je uložena v souboru appsetting.json. K získání hodnot je vždy potřeba klíč. Například zomato sekce má klíč

zomato, jak je vidět v metodě GetSection. Následně je naplněna proměnná uri, tedy cesta ke zdroji. Zde je vidět naplnění querry parametru id hodnotou z konfigurace. Hned po spuštění cyklu je provedeno logování. Poté je v bloku try odeslán požadavek na cílový zdroj s nezbytnou hlavičkou API klíče, opět doplněný z konfigurace. Pokud se vrátila odpověď s úspěšným status kódem, deserializuje se tělo zprávy na pole hodnocení, která jsou následně pomocí obchodní logiky ukládána do databáze. V implementaci obchodní logiky se ukládají pouze hodnocení, která ještě databáze neobsahuje a implicitně jim je nastaveno nepravda u atributu active. Na obrázku 27 je ukázka odpovědi ze Zomato.

Obrázek 27 – Zkrácené tělo odpovědi


```
"reviews_count": 4,
"reviews_start": 0,
"reviews_shown": 4,
"user_reviews": [
    {
        "review": {
            "rating": 5,
            "review_text": "It has the best butter chicken and butter garlic naan. Have been
                ordering from here from many years and it is always tasty.. must try",
            "id": 45483295,
            "rating_color": "305D02",
            "review_time_friendly": "4 months ago",
            "rating_text": "Insane!",
            "timestamp": 1570359478,
            "likes": 0,
            "user": {
                "name": "Goldy Gbf",
                "zomato_handle": "",
                "foodie_color": "ffd35d",
                "profile_url": "https://www.zomato.com/cs/users/goldy-gbf-3794281?
                    utm_source=api_basic_user&utm_medium=api&utm_campaign=v2.1",
                "profile_image": "https://b.zmtcdn.com/data/user_profile_pictures/8e2/
                    dd804840c526bbcf0ab1d9bee1ec08e2.jpg?fit=around%7C100%3A100&
                    crop=100%3A100%3B%2A%2C%2A",
                "profile_deeplink": "zomato://u/3794281"
            },
            "comments_count": 0
        }
    },
    {
```


Zdroj: vlastní zpracování

Jak je vidět, ze Zomato přijde spousty dat, ovšem do databáze jsou ukládány pouze identifikátor hodnocení, hodnota hodnocení a text hodnocení, viz tabulka 6. Dalším atributem je ještě source (zdroj), kde se plní hodnota zdroje hodnocení. Je možné, že bude

po rozšíření aplikace probíhat konzumace hodnocení i jiných aplikací a je dobré pak webu prezentace hodnoceních napsat, odkud hodnocení pochází.

Tabulka 6 – Výpis řádku z tabulky review

pk_reviews	rating	review_text	id_source	source	active
2	5	It has the b...	45483295	zomato	0

Zdroj: vlastní zpracování

4.3.5 Logování

K zaznamenávání proběhlých činností slouží logování. Přestože .Net Core poskytuje vestavěnou funkčnost logování, tak neumožňuje logování například od souboru, které je pro účel této aplikace chtěné. Proto je do aplikace stažen balíček NLog, který umožňuje mimo jiné logování do souboru. K možnému používání v celé aplikaci je potřeba zavolat ve třídě Program rozšířenou metodu UseNLog při vyváření webového hosta a zároveň je potřeba, aby byl součástí projektu WebApi konfigurační soubor nlog.config, kde se kromě přidání NLogu do projektu nastavuje místo, způsob a název ukládání vzniklých logů. Následně je možné použít třídu Logger v celé aplikaci. Nastavené logování v aplikaci rozlišuje 4 stavy závažnosti nastalé události. Je používán informační, varovný, chybový a kritický log. V aplikaci se loguje zavolání na tříde Logger metody Log s parametry LogLevel (udává stav a je typu enum), EventId (číslo závažnosti) a Message (zpráva, v případě chyby se jedná o popis výjimka, jinak vlastní text). Konkrétní implementace je vidět například na obrázku 26, kde ve třídě ZomatoJob, je metoda Log volána na začátku, kde má za úkol informovat o tom, že začíná úloha na pozadí, tudíž se jedná o informační log. Následně by mělo dojít ke komunikaci s vystaveným Zomato API a v případě, že se vrátí odpověď, která obsahuje status kód, který není úspěšný (nerovná se 2xx), dojde k zalogování chybového logu. Pokud komunikace se Zomato proběhne v pořádku, ale nastane chyba se zpracováním odpovědi a s jeho případným uložením do databáze, je volána metoda Log s parametrem, který označuje log jako kritický.

Jelikož je aplikace postavena hlavně jako API, je logována hlavně komunikace skrze API, tzn. že jsou logovány všechny požadavky a odpovědi. Nastavení logování požadavků je již ve třídě Startup, kdy je do projektu zaregistrován framework MVC. Logování odpovědí pak probíhá ve filtru RsActionFilter v metodě OnActionExecuted.

(Filtry jsou součástí životního cyklu požadavků a odpovědí a například akční filtr obsahuje dvě metody OnActionExceting a OnActionExecuted, které probíhají vždy těsně před a po spuštění akční metody v kontroleru, a proto se tento filtr skvěle používá k logování. Nabízí se otázka, proč je nastaveno logování požadavku v momentu, kdy se registruje MVC a neprobíhá stejně, jako odpověď ve filtru RsActionFilter. Je to z důvodu, kdy použití atributu ApiController v kapitole 4.3.3 vstupuje do životního cyklu požadavků a odpovědí ještě před RsActionFilterem, kde použije validační metodu na požadavek a pokud zjistí, že požadavek není validní, vše je ukončeno a je vrácena odpověď se statusem 4xx, tzn. logování je potřeba nastavit už na začátek životního cyklu požadavek/odpověď). Následující text pod odstavcem je ukázka zápisu 2 logů. První je informační a říká, že proběhl požadavek na zdroj api/Home|action v určitý čas. Druhý log je kritický a říká, že nastala chyba NHibernate, což znamená chyba v komunikaci s databází a zároveň ukazuje celý „stack trace“, což znamená, že ukazuje, kde celkové volání začalo. Jak ukazuje poslední řádek logu, začalo zavoláním akční metody GetHomeData, následně volání obchodní logiky CustomerCoreService a poté zavolaní metody GetMenuByDate, která je ze třídy MenuDao, ve které používá z předka DaoBase vlastnosti CurrentSession a zde to spadlo. Díky logu je tedy známo, co vše proběhlo do vyhození výjimky.

1.log

2020-02-23 8:15:46.5651|2|INFO|WebApi.Middlewares.RequestLoggingMiddleware|Http Request Information: Schema:http Host: localhost:56847 Path: /api/Home QueryString: Request Body: |url: http://localhost/api/Home|action: GetHomeData

2.log

2020-02-23 18:15:47.8109|5|FATAL|My logger| at NHibernate.Cfg.ConfigurationSchema.HibernateConfiguration..ctor(XmlReader hbConfigurationReader, Boolean fromAppSetting)
at NHibernate.Cfg.ConfigurationSchema.HibernateConfiguration..ctor(XmlReader hbConfigurationReader) at NHibernate.Cfg.Configuration.Configure(XmlReader textReader) at NHibernate.Cfg.Configuration.Configure(String fileName)
at Infrastructure.Data.NHibernateHelper.get_Session() in C:\Users\Martin\source\repos\restaurant\RS\Infrastructure\Data\NHibernateHelper.cs:line

4.3.6 Zabezpečení

Bez přihlášení do aplikace jsou používané případy užití UC1-3. K používání další funkčnosti je potřeba úspěšně projít procesem autentizace a autorizace. Je tedy potřeba se přihlásit, což zajišťuje UC7. Přihlásit se může uživatel v případě, že se již zaregistroval pomocí UC3. V registraci v rámci UC3 se registrují uživatelé, kteří obdrží roli zákazníka. Registrace zaměstnanců je prováděna v administrátorské sekci (pouze v případě této aplikace je ideální mít celkově oddělenou klientskou aplikaci pro zaměstnance a zákazníky). Proces registrace a přihlášení obstarává UseController ve které jsou 2 POST akční metody.

V metodě AddUser je volána obchodní logika pomocí metody RegisterCustomer, která je rozdělena na 3 části. V první je ověřeno, že již není e-mail registrován. Ve druhé je vytvořeno heslo. Vytvoření hesla probíhá pomocí třídy HMACSHA512, která využívá hashovací funkci SHA-512. Pomocí konstruktoru této třídy je vytvořena instance a následně, jak je vidět na obrázku 28, uloží klíč hashe (náhodně vygenerovaný) z této instance do proměnné passwordSalt. Poté je z této instance dopočítán hash z hesla, které bylo zasláno uživatelem. Poté se nový uživatel uloží do databáze s hash klíčem a heslem. Existují dva důvody, proč je potřeba hashovat heslo a neukládat původní heslo zaslané uživatelem. Vždy se může stát, že se někomu podaří dostat do databáze a zmocnit se dat a druhým důvodem je, že ani sami provozovatelé aplikace by neměli mít možnost nahlížet na hesla.

Obrázek 28 – Metoda CreatePasswordHash

```
private void CreatePasswordHash(string password, out byte[] passwordHash,
    out byte[] passwordSalt)
{
    using (var hmac = new System.Security.Cryptography.HMACSHA512())
    {
        passwordSalt = hmac.Key;
        passwordHash = hmac.ComputeHash( buffer: Encoding.UTF8.GetBytes(password));
    }
}
```

Zdroj: vlastní zpracování

Po úspěšném zaregistrování je možné se přihlásit. Jak již bylo napsáno, přihlášení poskytuje stejný kontroler, ale akční metoda se jmenuje GetToken. V těle metody se volá opět obchodní logika, ve které je potřeba určit, jestli uživatel je zaregistrován a poslal správné heslo. Ověření hesla proběhne opět pomocí třídy HMACSHA512, kde vznikne

instance třídy, ovšem tentokrát je zavolán konstruktor s parametrem klíče, který je uložený v databázi. Pomocí instance je spočten hash hesla, které bylo v rámci přihlášení posláno a následně je tento spočtený hash porovnán s hashem uloženým v databázi viz obrázek 29. V případě shodnosti hashů je vytvořen autentizační token. V aplikaci je používán JSON Web Token a pro kontrolery CustomerController a ManagerControler je přidána vrstva autentizace, konktrétně se zde používá Bearer. To znamená, že je potřeba v rámci požadavků, které se dotazují na zdroje zmíněných kontrolerů, aby obsahovaly v hlavičce tento token. Token má nastavenou expirační dobu na 1 hodinu.

Obrázek 29 – Metoda VerifyPasswordHash

```
private bool VerifyPasswordHash(string password, byte[] passwordHash, byte[] passwordSalt)
{
    using (var hmac = new System.Security.Cryptography.HMACSHA512(passwordSalt))
    {
        var computedHash = hmac.ComputeHash( buffer: Encoding.UTF8.GetBytes(password));

        for (int i = 0; i < computedHash.Length; i++)
        {
            if (computedHash[i] != passwordHash[i])
            {
                return false;
            }
        }
        return true;
    }
}
```

Zdroj: vlastní zpracování

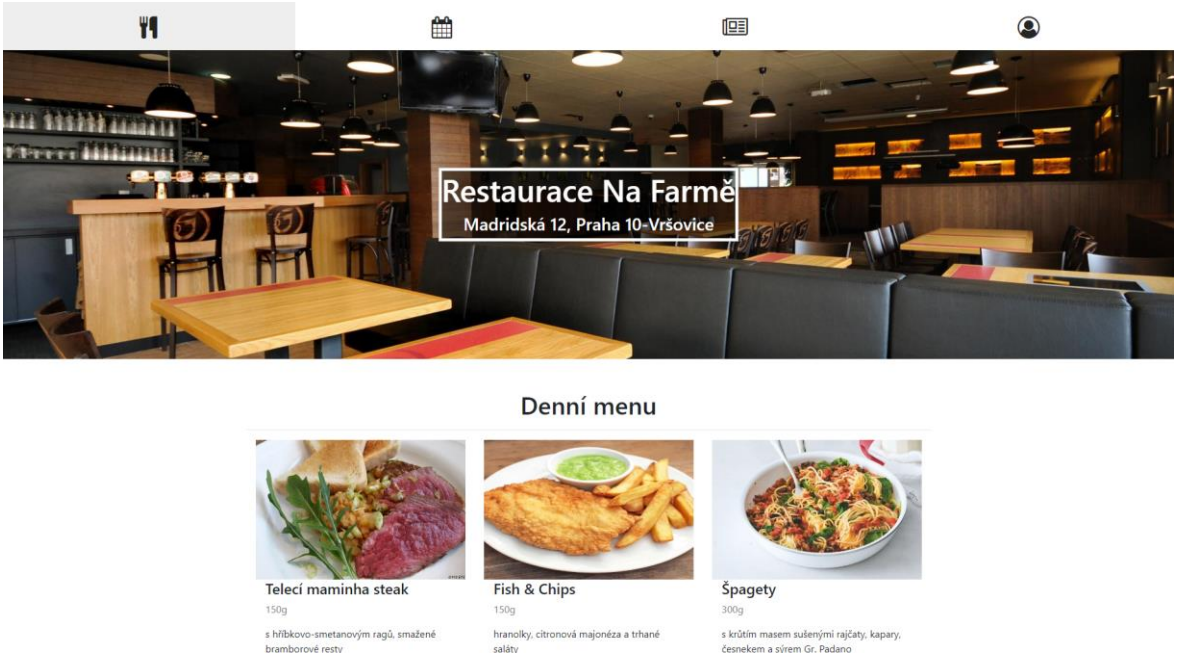
4.4 Klientská část

Klientská část je webová aplikace, která je napsána v Angularu. Používá se zde hlavně HTML, CSS a TypeScript. Celkově je webová aplikace rozdělena na 4 hlavní sekce a jako navigace mezi nimi slouží navigační menu, které je vidět v horní části obrázku 30.

4.4.1 Úvodní obrazovka

Po otevření webové aplikace je vidět úvodní foto, na kterém je adresa restaurace a pod ním se nachází denní menu. V aplikaci se pod denní nabídkou nachází stálá nabídka, nápojový lístek a dále ještě hodnocení restaurace.

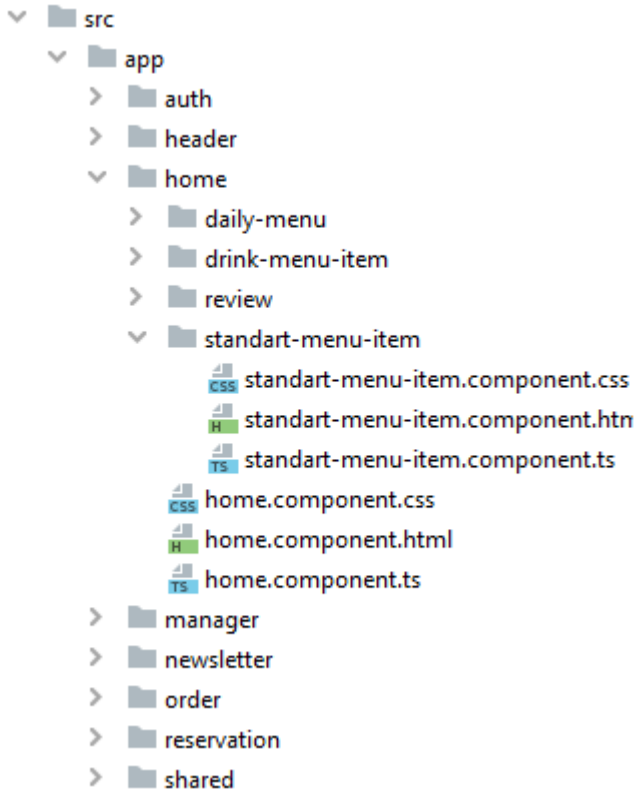
Obrázek 30 – Úvodní obrazovka webové aplikace



Zdroj: vlastní zpracování

Všechna tato data jsou výsledkem volání HTTP metody GET na zdroj api/home. Celá stránka se skládá z komponent. V případě úvodní obrazovky se jedná o komponenty <app-header> a <app-home>. Komponenta <app-header> se zobrazuje v celé aplikaci. Komponenta <app-home> obsahuje další jednotlivé komponenty, a to <app-daily-menu> (komponenta pro denní menu), <app-drink-menu-item> (komponenta pro položku nápojového lístku), <app-standart-menu-item> (komponenta pro položku stálého jídelního lístku) a <app-review> (komponenta pro hodnocení). Výtažek struktury adresáře (hlavně pro komponenty úvodní obrazovky) je vidět na obrázku 31. Práce s komponentami umožňuje lepší čitelnost a udržovatelnost kódu a šetří i výkon, protože jednotlivé komponenty jsou zobrazované pouze v případě, že jsou data pro jejich naplnění. Jsou také lépe testovatelné, protože vystupují jako samostatné jednotky. I v případě chyby je nalezení rychlé a je vidět, ve které komponentě došlo k chybě. Jak je vidět na obrázku, každá komponenta se skládá ze tří souborů. Jedná se o povinný TypeScript soubor a nepovinné soubory HTML a CSS.

Obrázek 31 – Část adresářové struktury



Zdroj: vlastní zpracování

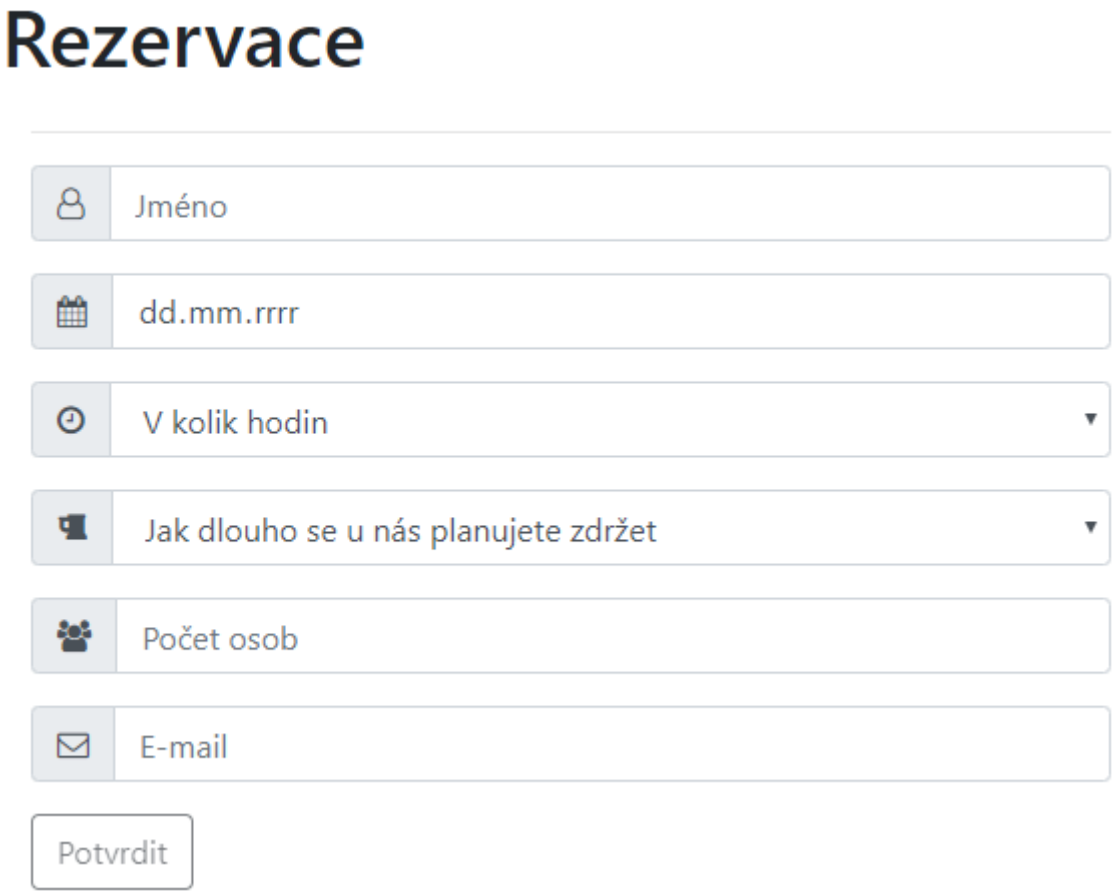
Sdílené věci napříč aplikací, jako hlavně různé stylové efekty, služby a doménové modely, jsou umístěny ve složce shared. Jednotlivé používané služby jsou injektovány pomocí konstruktoru. Implementace služeb v rámci sdílených souboru funguje tak, že se zavolá HTTP klient a něm jedna z metod get, post apod. Díky tomu, že jsou metody generické, je znám typ odpovědi. Volání metod probíhá tedy ve sdílené části, ale jednotlivé zpracování odpovědi v konkrétních místech, kde jsou volané. Na zpracování odpovědi se v Angularu volá povinná metoda Subcribe, ve které je umožněno podle odpovědi jednat, tzn. v případě úspěchu a vrácení dat, naplnit lokální proměnné a nechat Angular, ať je reaktivně zobrazí a v případě neúspěchu, tedy vrácení chyby v odpovědi, zareagovat notifikací, aby uživatel věděl, že něco nefunguje, tak jak očekával.

4.4.2 Záložka rezervace

Záložka rezervace je složená ze dvou komponent. První komponenta je čistě informační a má název kontakt, kde je možné nalézt kontaktní informace, jako otevírací dobu, telefon, e-mail a mapu. Druhá komponenta slouží jako rezervační formulář, viz

obrázek 32, kde je možné vidět, které hodnoty je potřeba vyplnit. Jsou to stejné hodnoty, jako jsou uvedené v tabulce 4, která představuje ReservationRequest v serverové aplikaci.

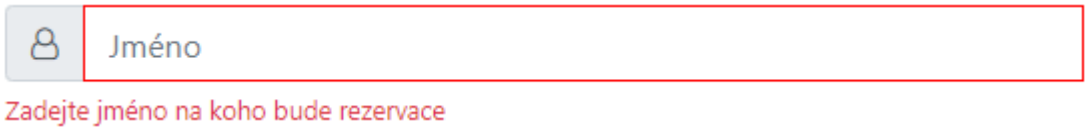
Obrázek 32 – Komponenta rezervační formulář



Zdroj: vlastní zpracování

Aby se šetřil výkon na serverové straně a neodesílal se nevalidní formulář, je pomocí atributů v Angularu nastaveno kontrolování vyplněného formuláře. U všech kolonek je použit atribut Required (požadované) a u speciálních kolonek další atributy. Například položka „e-mail“ musí mít podobu e-mailu, položka „datum“ povolí vestavěná komponenta prohlížeče vybrat jen dnešní datum a pozdější, položka „v kolik hodin“ má předdefinované hodnoty po půlhodině, počínaje od 15:00. Pokud je nějaký prvek nevalidní, formulář není možné odeslat (tlačítko je vypnuté) a ve formuláři se objeví chybová hláška, jak je možné vidět na obrázku 33.

Obrázek 33 – Nevalidní prvek formuláře



Zdroj: vlastní zpracování

V případě validního formuláře je použita HTTP metoda POST a tělo této metody může vypadat, jak ukazuje následující výpis:

{"onName": "Martin Jermář", "countPeople": 6, email: "m.jermar@seznam.cz", "startTime": 1700, "finishTime": 2000, "date": "2020-03-27"}

Pokud rezervace proběhla úspěšně, zobrazí se notifikace v podobě zeleného boxu vedle nadpisu komponenty s oznámením, že rezervace proběhla úspěšně. Tělo odpovědi vypadá následovně:

{"result":"success"}

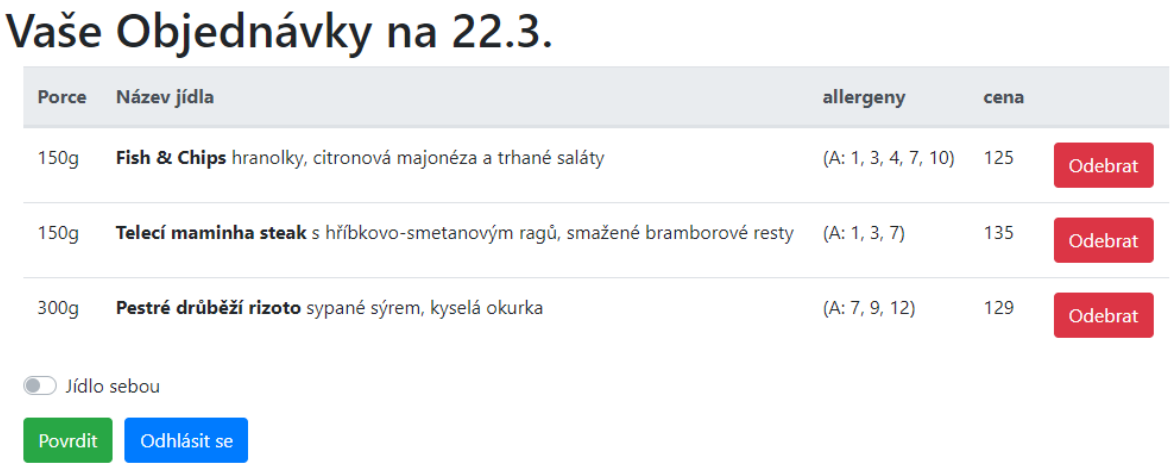
4.4.3 Záložka uživatel

Na poslední záložce je uživatelská sekce. Pokud uživatel není přihlášený, vidí komponentu umožňující přihlášení nebo registrování. Komponenta využívá služby auth.service.ts, na které jsou 2 POST metody, které volají UserController ze serverové části. Metody obstarávají registraci a přihlášení. Registrace na této komponentě je určena pouze pro zákazníky. Stejně jako zbylé komponenty obsahují formuláře potřebné validace. Komponenta pro přihlášení v případě neúspěchu rozlišuje dva stavy a umí o nich notifikovat uživatele. Jedná se o stavy, kdy není znám e-mail nebo je zadané špatné heslo. V případě registrace dochází k chybě pouze, pokud je již e-mail registrován.

Po úspěšném přihlášení se zobrazí komponenty na základě role uživatele. V případě role „customer“ je zobrazena komponenta, díky které lze spravovat objednávky jídel z polední nabídky. Je možné objednat více než jedno jídlo. Objednávku je možné spravovat vždy do půlnoci. Poté je spravována objednávka na další den. Zákaznická sekce obsahuje dvě komponenty, které spolu reaktivně komunikují. První komponenta, viz obrázek 34, ukazuje, jak vypadá správa objednávek. Všechny změny je potřeba vždy potvrdit. Zaškrtnutím checkboxu je možné označit objednávku jako jídlo sebou. Přidávání

jídel do objednávky má na starost druhá komponenta na stránce, která vypadá stejně jako komponenta denní menu na obrázku 30 s jedinou změnou, kde černé políčko ukazující cenu, nahradilo tlačítko funkcí přidat jídlo do objednávky. Zde na tomto příkladu je vidět možnost znovu využití komponent.

Obrázek 34 – Komponenta objednávka



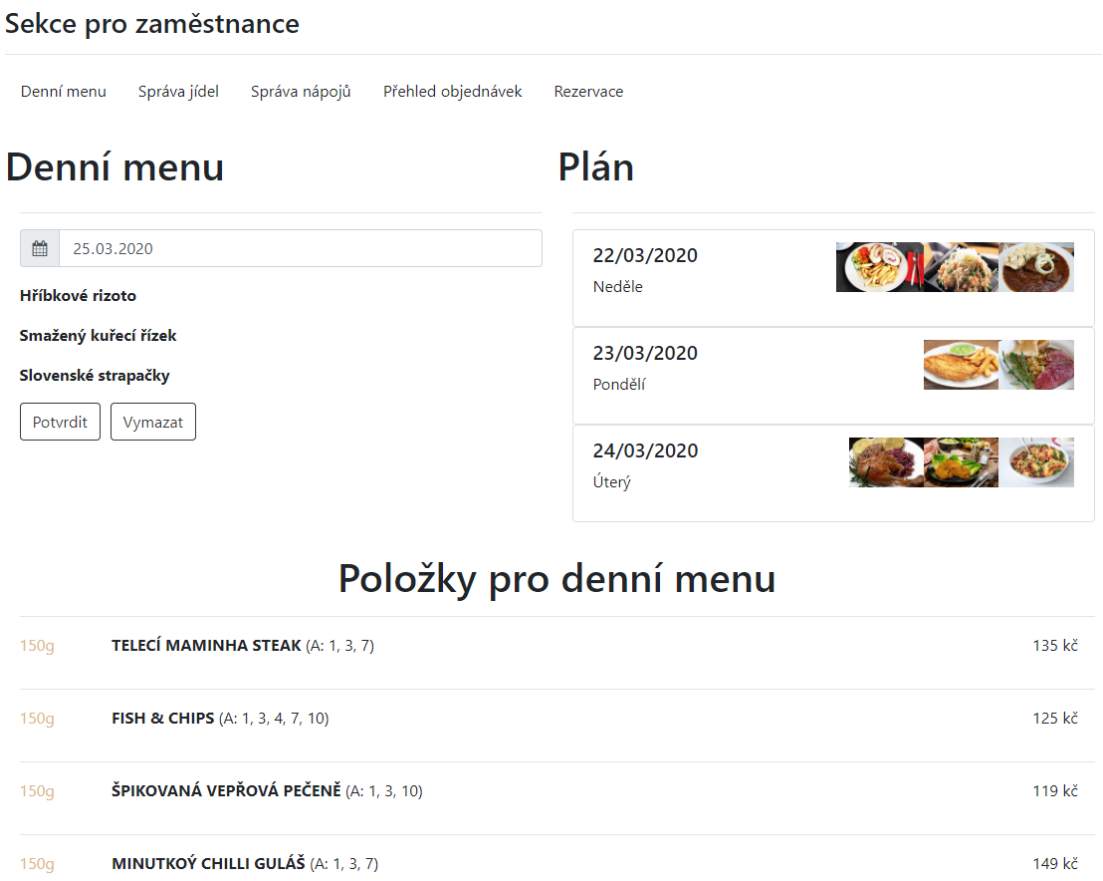
Zdroj: vlastní zpracování

V případě, že se přihlásí uživatel s jinou rolí než „zákazník“, je zobrazena zaměstnanecká komponenta. Tato komponenta umožňuje ovládání a přístup do podřízených komponent jako denní menu, správa jídel, rezervace atd. podle role uživatele. Klientská aplikace rozlišuje role „empoloyee“, „manager“ a „admin“. První a druhá zmíněná role neumožňuje přidávání nových zaměstnanců, správu hodnocení a správu informací ohledně restaurace. Role „manager“ je určena hlavně pro provozní a kuchaře, kteří se starají o denní menu, běžné menu a správu položek jídel a pití v databázi s netechnickým přístupem. Poslední role umožňuje přístup do přehledu objednávek a správu rezervací. Systém je nastaven tak, že vždy vyšší role má přístupy jako podřízená role viz následující pořadí rolí: „employee“ => „manager“ => „admin“. Jako příklad je v následujícím odstavci uvedena komponenta pro správu denního menu.

Sekci pro zaměstnance je možné vidět na obrázku 35, kde je zobrazena komponenta pro správu denního menu. Sekce pro zaměstnance je zobrazena pro roli „manager“. Správa pro denní menu se skládá z částí, kde v první je možné vytvářet nebo spravovat denní menu, ve druhé části s nadpisem Plán, kde je vidět na které dny je vytvořeno denní menu.

Po kliknutí na dané menu je možné ho upravovat a poslední část je znovu využitá komponenta pro zobrazování uložených jídel, které mají příznak aktivní. Všechny tři komponenty spolu reaktivně komunikují. Kliknutím na jídlo je přidáno v komponentě denní menu. Komponenta dovoluje přidat vždy unikátní položku jídlo. Při vytváření nového menu se po potvrzení ihned zobrazí nová položka v Plánu.

Obrázek 35 – Sekce pro zaměstnance



Zdroj: vlastní zpracování

5 Výsledky a diskuze

Výsledná serverová aplikace byla naimplementována podle stanovených cílů, které splňuje. Hlavně v programování je ale běžné, že stávající řešení je skoro vždy možné optimalizovat nebo řešit dílčí úlohy jinými postupy. Stávající aplikace je navržena tak, aby byla lehce rozšiřitelná. Pro lepší přehlednost je aplikace rozdělena na 3 vrstvy. V prezentační vrstvě je možné používat méně tříd pro vystavení REST API. To stejné platí hlavně ve vrstvě, která se stará o komunikace s databází. Pokud se jedná o malé projekty, není potřeba vytvářet pro každou tabulku zvlášť dao třídy a konfigurační xml soubory. Důvodem pro toto použití je, aby se vývojář, který se rozhodne používat toto řešení, byl schopen se zorientovat v aplikaci a také připravenost na již zmíněnou rozšiřitelnost aplikace.

Ve vývoji aplikace probíhaly pouze vývojářské testy. Všechny základní funkčnosti byly otestovány v rámci testovacího prostředí. S přechodem na ostrý provoz je nepochybné, že se v aplikaci objeví méně či více závazné chyby, které bude potřeba řešit. Díky tomu, že je zdrojový kód nahrán na veřejně dostupné místo, je možné neustále aplikaci vylepšovat. Právo pro schvalování změn do kódu zůstane alespoň ze začátku autorovi diplomové práce.

Pro aplikaci nebyly prozatím v České republice nalezeni přímí konkurenti. Je možné najít konkurenty, kteří nabízejí dílčí funkčnosti. Např: aplikace Restu nabízí rezervační modul, který obstarává rezervace. Rezervace jsou ale potřeba potvrdit restauracemi, a proto se jeví jako lepší použití rezervačního systému na kliknutí, který je přímo propojen s databází a umí vyhodnotit obsazenost. Dalším nepřímým konkurentem jsou objednávkové aplikace DámeJídlo, Wolt, UberEats atd. Pomocí těchto aplikací je možné si objednat jídlo s dovozem nebo vyzvednutím v restauraci. Aplikace ale nepřináší zmíněné výhody v podobě známého počtu jídel den dopředu. Co dnes restaurace umí, je rozesílání newsletteru. Řešení probíhá pomocí aplikací třetích stran nebo jako vlastní řešení. Ze srovnání s konkurencí vyplývá, že obdobné dílčí funkce již existují. Tato diplomová práce přináší souhrn základních funkcí do jedné aplikace, která je poskytována jako open-source.

6 Závěr

Hlavním cílem diplomové práce bylo vytvoření open-source řešení pro restaurační zařízení v podobě serverové aplikace. Aplikace obsahuje objednávkový systém pro denní menu, pravidelné zasílaní newsletteru, systém rezervací na kliknutí, konzumace webových služeb pro stahování hodnocení a celkovou administraci databázových položek v přívětivém uživatelském prostředí. Komunikace s aplikací je zajištěna skrze vystavené REST API. Mezi dílčí cíl práce patřilo vytvoření klientské aplikace, která demonstruje možnosti využití API ze serverové části.

V diplomové práci jsou popsána teoretická východiska, ze kterých se vycházelo v praktické části. Byla zde popsána architektura REST API, HTTP metody a doporučené postupy, metody a pravidla pro jejich vytváření. V další podkapitole je popsán návrhový vzor Dependency injection, který je využíván ve vývoji obou aplikací. V praktické části je zdokumentován celý vývojový cyklus. Na základě požadavků byla provedena analýza, ve které vznikl diagram užití a specifické případy užití. Následně byla navržena a vytvořena databáze reflektující požadavky z analýzy. Vývoj serverové části byl detailně popsán. Jsou zde uvedeny příklady počátečního nastavení, persistence dat, akční metody, úlohy na pozadí, logování a bezpečnost. V klientské části byly popsány důležité komponenty. U jednotlivých komponent byl uveden význam a její funkčnosti.

Při vývoji aplikace byly použity následující technologie a nástroje: databáze MS SQL a jazyk SQL. Jedná se o relační databázi. Propojení serverové aplikace a databáze je řešeno pomocí objektově relačního frameworku NHibernate. K vývoji serverové části je využit .Net Core a programovací jazyk C#. Klientská část aplikace je vyvinuta v Angularu, ve kterém se hlavně používá HTML, CSS a TypeScript. Pro verzování kódu byl využit systém Git.

Závěrem lze konstatovat, že výsledné aplikace splňují požadované funkčnosti. Cíle práce byly splněny. Vzhledem k návrhu aplikace a jejímu vystavení na BitBucket je možné serverovou aplikaci dále rozšiřovat o nové funkčnosti. Nabízejí se tři významné směry rozšíření. Prvním je vyvinout funkčnosti spojené s platbami. Aplikace by mohla převzít roli, kterou mají dnes na starost platební systémy, tzn. propojení s kasou, bankovním terminálem a napojení na EET. Dalším směrem vývoje jsou zásoby a objednání v rámci B2B. Údaje o aktuálních zásobách surovin by se udržovaly v digitální formě a na základě těchto dat se mohou objednávat elektronicky u dodavatelů. Objednání by mohlo probíhat

manuálně nebo pomocí umělé inteligence. V aplikaci by se také mohl vyvinout administrativní modul pro chod restaurace např. systém plánování směn, ukládání dokumentů atd. Celkově lze očekávat v budoucnu vyšší zapojení informačních technologií do restauračního prostředí.

7 Seznam použitých zdrojů

[1] MOTROC, Gabriela. *SOAP vs. REST* [online]. [cit. 2020-03-04]. Dostupné z: https://jaxenter.com/state-of-api-integration-report-136342.html

[2] *SOAP vs. REST* [online]. [cit. 2020-03-04]. Dostupné z: https://stackify.com/soap-vs-rest/

[3] REYNDERS, Fanie. *Modern API Design with ASP.NET Core 2: Building Cross-Platform Back-End Systems*. 1. Apress, 2018. ISBN 978-1-4842-3519-5.

[4] MASSÉ, Mark. *REST API design rulebook*. Sebastopol, CA: O'Reilly, 2012. ISBN 9781449310509.

[5] ALLAMARAJU, Subbu. *RESTful Web Services Cookbook*. USA: O'Reilly Media, 2010. ISBN 9780596801687.

[6] VAN DEURSEN, Steven a Mark SEEMANN. *Dependency Injection Principles, Practices, and Patterns*. Manning Publications, 2019. ISBN 9781617294730.

[7] FIELDING, Roy. *Representational State Transfer (REST)* [online]. [cit. 2020-03-04]. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[8] BERNER-LEE, Tim. *Universal Resource Identifiers* [online]. [cit. 2020-03-04]. Dostupné z: http://www.w3.org/DesignIssues/Axioms.html

[9] BERNERS, Tim. *RFC 3986* [online]. [cit. 2020-03-04]. Dostupné z: https://tools.ietf.org/html/rfc3986

[10] HOLEC, Miroslav. *RESTful API Design* [online]. [cit. 2020-03-04]. Dostupné z: https://cdn.miroslavholec.cz/articles/restful-api-design/restful-api-design.html

[11] FIELDING, et al. *Method Definitions* [online]. [cit. 2020-03-04]. Dostupné z: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9

[12] *HTTP request methods* [online]. [cit. 2020-03-04]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[13] FIELDING, et al. *10 Status Code Definitions* [online]. [cit. 2020-03-04]. Dostupné z: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

[14] *Hypertext Transfer Protocol (HTTP) Status Code Registry* [online]. [cit. 2020-03-04]. Dostupné z: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

[15] FIELDING, et al. *HTTP Message* [online]. [cit. 2020-03-04]. Dostupné z: https://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html

[16] DIXIT, Achal. *HTTP headers* [online]. [cit. 2020-03-04]. Dostupné z: https://www.geeksforgeeks.org/http-headers/

[17] *Dependency Injection* [online]. [cit. 2020-03-04]. Dostupné z: https://www.tutorialsteacher.com/ioc

[18] LOCK, Andrew. *ASP.NET Core in Action*. USA: Manning Publications, 2018. ISBN 9781617294617.

[19] *MSSQL* [online]. [cit. 2020-03-04]. Dostupné z: https://cs.wikipedia.org/wiki/Microsoft_SQL_Server

[20] *ORM* [online]. [cit. 2020-03-04]. Dostupné z: https://www.tutorialspoint.com/nhibernate/nhibernate_orm.htm

[21] *Angular documentation* [online]. [cit. 2020-03-04]. Dostupné z: https://angular.io/start

[22] HARKUSHKO, Liliia. *Angular* [online]. [cit. 2020-03-04]. Dostupné z: https://yalantis.com/blog/when-to-use-angular/

[23] *Bootstrap* [online]. [cit. 2020-03-04]. Dostupné z: https://cs.wikipedia.org/wiki/Bootstrap

[24] *Zapier* [online]. [cit. 2020-03-04]. Dostupné z: https://zapier.com/learn/apis/chapter-2-protocols/

8 Přílohy

Aplikace se zdrojovými kódy jsou dostupné v přiložených zip. souborech.

Odkazy na veřejné úložiště:

https://bitbucket.org/kulidev/restaurantfe/src/master/

https://bitbucket.org/kulidev/restaurant/src/master/