Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Vytvoření webové aplikace s využitím .NET 5 a MVC6

Bc. Martin Brabec

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Martin Brabec

Informatika

Název práce

Vytvoření webové aplikace s využitím .NET 5 a MVC6

Název anglicky

Web application development using .NET 5 and MVC6

Cíle práce

Cílem práce je s využitím nejnovějších nástrojů, postupů a vzorů v oblasti vývoje webových stránek navrhnout a implementovat moderní webovou aplikaci sestávající z webové prezentace a související administrační části. Výsledná webová aplikace bude implementována v prostředí Microsoft ASP.NET 5 v jazyce C# s použitím MVC 6.

Metodika

Metodika práce je založena na analyticko-syntetickém přístupu. Bude provedena analýza odborných informačních zdrojů a na základě syntézy zjištěných poznatků budou popsány vlastnosti nových technologií a způsob jejich použití. Těchto poznatků bude dále využito při návrhu a implementaci webové aplikace. Postup návrhu, implementace, testování a nasazení bude popsán a zhodnocen.

Doporučený rozsah práce

60-80 stran

Klíčová slova

web, MVC6, .NET, webová aplikace, ASP.NET, analýza, implementace

---

Doporučené zdroje informací

JOSHI, Nimit. Programming ASP.NET MVC 5. C# Corner, 2013. 135.
SHARP, John. Microsoft Visual C# 2010 : krok za krokem. Praha 4: Albatros Media a. s., 2012. 696. ISBN 978-80-251-3147-3
TORRE, Cesar. .NET Technology Guide for Business Applications. Microsoft Press, 2013. 70.

Předběžný termín obhajoby

2015/16 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 20. 2. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 03. 2016

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Vytvoření webové aplikace s využitím .NET 5 a MVC6" 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 29.3.2016 _____

Poděkování

Rád bych touto cestou poděkoval panu Ing. Jiřímu Brožkovi, Ph.D. za pomoc a rady při sběru informací, jejich následném zpracování a zakomponování do finálního projektu.

Vytvoření webové aplikace s využitím .NET 5 a MVC6

Souhrn

Hlavní náplní této práce je návrh a implementace moderní webové aplikace v prostředí ASP.NET Core 1.0, skládající se z prezentačního webu a administrační části. V rámci práce je také navržena vlastní obecná architektura datového zdroje (CRUDQ) libovolné aplikace, jejíž implementace je dále předvedena ve vyvíjené webové aplikaci. Jako databázový systém je zvolena dokumentová databáze MongoDB, do které je přistupováno právě pomocí autorem navržené architektury. Mezi další významné aspekty práce patří implementace efektivního propojení architektur MVC a MVVM, které je docíleno pomocí vlastního generátoru JavaScriptového datového modelu. Tento generovaný datový model je využíván frameworkem KnockoutJS pro zajištění funkčnosti uživatelského rozhraní. Výsledná webová aplikace je plně funkční jednoduchý CMS, spustitelný na třech nejpoužívanějších desktopových operačních systémech, využívající nejmodernější technologie, nástroje a návrhové vzory.

Klíčová slova: ASP.NET, Core, C#, CMS, MongoDB, CRUDQ, KnockoutJS, MVC, MVVM

Creating web application using .NET 5 and MVC6

Summary

The main concern of this thesis is the design and implementation of advanced Web application in ASP.NET 1.0 Core, consisting of a presentation web and administration section. A part of the thesis is also design of a custom architecture for a data source (CRUDQ) suitable for any application, whose implementation is further demonstrated in the developed web application. Selected database system is document database called MongoDB, which is accessed using the author designed architecture approach. Other important aspects of the work includes for example effective interconnection between MVC and MVVM architectures, which is achieved by custom generator of JavaScript data model. The generated data model is used by KnockoutJS framework to ensure the functionality of user interface. The web application is fully functional simple CMS, executable on three most widely used desktop operating systems, using the latest technology tools and design patterns.

Keywords: ASP.NET, Core, C#, CMS, MongoDB, CRUDQ, KnockoutJS, MVC, MVVM

Obsah

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

1 Úvod

Rozmanitost technologií a platforem na tvorbu webových stránek se v posledních letech značně zvýšila. Nabízí možnosti použití od statické webové prezentace až po velmi komplexní webové aplikace. To vše většinou v příjemném a uživatelsky přívětivém prostředí. Ovšem zvolená technologie nikdy nemusí být plodem jedné, či dvou společností, jako tomu bývalo dříve. A zároveň také nemusí být kompatibilní jen s jednou platformou. Ve velké míře jsou totiž zastoupeny technologie, sestavené z několika dílčích nástrojů, vytvořených různými společnostmi či jednotlivci, přičemž odlišné je pouze běhové prostředí. Zmíněné nástroje však zřídkakdy řeší komplexní problém. Drtivá většina řeší vždy pouze jednotlivý aspekt webové aplikace. Tato skutečnost jim ale dodává volnost ve smyslu možnosti použití v různých technologiích a na různých platformách.

Jednou z komplexních technologií, sestavených z mnoha menších nástrojů, je i nejnovější produkt v oblasti vývoje webových aplikací od společnosti Microsoft. Tato technologie se jmenuje .NET Core, a je, dle vyjádření společnosti Microsoft, vytvořena zcela od začátku a bez zpětné kompatibility se staršími verzemi technologií pro vývoj webových aplikací od této společnosti. Tento zcela nový framework je nejen iterací přidávající několik nových funkcí, ale jedná se i o kompletní změnu přístupu, neboť je .NET Core open-source a kompatibilní se třemi nejpoužívanějšími operačními systémy.

Mimo to však nabízí i celou řadu novinek, jako je například integrovaný IoC kontejner, integrované služby pro logování a konfiguraci, strukturu projektu, balíčkování a v neposlední řadě také nový přístup pro automatizování opakujících se úloh. Ve spojení s ostatními nástroji se otevírají zcela nové možnosti a přístupy pro vytváření webových stránek.

Tato práce tak v první řadě obsahuje přiblížení technologií, nástrojů, architektur a návrhových vzorů, ze kterých je následně sestaven plně funkční jednoduchý CMS systém, využívající celou řadu moderních a v praxi často využívaných technologií a nástrojů. Kromě již existujících architektur a návrhových vzorů tento CMS systém využívá autorem navrženou architekturu datového zdroje a vlastní implementaci propojení dvou běžně používaných architektur webových stránek.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem práce je s využitím nejnovějších nástrojů, postupů a vzorů v oblasti vývoje webových stránek navrhnout a implementovat moderní webovou aplikaci, sestávající z webové prezentace a související administrační části. Administrační část aplikace bude sloužit pro správu webového obsahu a pro základní nastavení vzhledu webové prezentace. Výsledná webová aplikace bude implementována v multiplatformním prostředí Microsoft ASP.NET 5 v jazyce C# s použitím MVC 6. Databázový systém, který bude pro webovou aplikaci zvolen, musí vyhovovat jejím požadavkům a musí být také multiplatformní.

Dílčím cílem bude navržení vlastní architektury datového zdroje, umožňující snadnou záměnu databázového systému. Tato architektura bude zahrnovat způsoby manipulace a přístupu k datům, řešení návrhu datové struktury a rozvržení souborů se zdrojovým kódem v souborovém systému. Ukázkově pak bude implementována ve vyvíjené webové aplikaci.

Aby bylo možné dodržet hlavní cíl práce, který mimo jiné vyžaduje využití nejnovějších nástrojů v oblasti vývoje webových stránek, musí práce obsahovat i vlastní řešení pro efektivní propojení serverové a klientské strany webové aplikace, což umožní využití moderních client-side frameworků.

2.2 Metodika

Metodika práce je založena na analyticko-syntetickém přístupu. Bude provedena analýza odborných informačních zdrojů a na základě syntézy zjištěných poznatků budou popsány vlastnosti nových technologií a způsob jejich použití. Těchto poznatků bude dále využito při návrhu a implementaci webové aplikace, jejich dílčích součástí, a také při návrhu architektury datové základny. Na závěr bude popsán a zhodnocen postup návrhu, implementace, testování a nasazení.

3 Teoretická východiska

3.1 Úvod do .NET

Základem veškerých technologií společnosti Microsoft, které souvisí s vývojem aplikací, je značka .NET (čteno „dotnet“). Sama o sobě nereprezentuje žádný konkrétní produkt, ale vyjadřuje jistou kategorii, do které spadá celá řada konkrétních produktů. Jména takovýchto konkrétních produktů zpravidla mívají obsaženo slovo .NET (například ASP.NET, .NET Core a další).

3.1.1 Jmenná konvence

Dne 19.1.2016 Scott Hansleman (jeden z vývojářů ASP.NET) na svém blogu uvedl, že předměty této práce budou oficiálně přejmenovány [1]. Změna se týká .NET Core, ASP.NET, Entity Frameworku a dalších. Tuto skutečnost dokládá také pohled do zdrojového kódu [2], kde je možné nalézt oznámení o změně názvu. Vzhledem k těmto skutečnostem bude v této práci použit již nový jmenný styl. Níže uvedená Tabulka 1 obsahuje převod ze starých názvů na nové.

Důvod k uvedené změně je prostý. Dle současného číslování verzí by mohli někteří dojít k závěru, že ASP.NET 5 (.NET Core 5) je nadřazená verze pro ASP.NET 4 (.NET Framework 4.6). To ovšem neodpovídá skutečnosti, jelikož platforma .NET Framework 4.6 bude prozatím provozována současně s platformou .NET Core 5. Jinými slovy bude dle nové jmenné konvence existovat ASP.NET 4.6, které vychází z již léta ověřené platformy, a ASP.NET Core 1.0 reprezentující novou platformu, která je jedním ze základních předmětů této práce.

Slovo „Core“ a verze 1.0 mají také svůj význam. Tímto slovem je označován nový přístup k ASP.NET obecně. Verze 1.0 značí, že se jedná o první iteraci tohoto nového přístupu.

Starý název	Nový název
ASP.NET 5	**ASP.NET Core 1.0**
.NET Core 5	**.NET Core 1.0**
Entity Framework 7	**Entity Framework 1.0**

Tabulka 1 - Převodní tabulka názvů

Od tohoto bodu dále budou v práci používány nové názvy z výše uvedené tabulky.

3.1.2 .NET Framework obecně

Jedna z nejznámějších a nejpoužívanějších platforem pro vývoj a tvorbu aplikací různých druhů je, hned po php, takzvaný .NET Framework [3]. Ve své podstatě se skládá z vývojového prostředí, programovacích jazyků a běhového prostředí. Programy napsané v .NET Frameworku jsou spouštěny ve virtualizovaném prostředí, známém jako CLR – Common Language Runtime. Toto běhové prostředí má na starosti správu paměti, zabezpečení, správu chyb a zajištění běhu aplikací. Detailní popis fungování CLR je však nad rozsah této práce.

Společnost Microsoft nad tímto frameworkem vytvořila celou řadu nadstaveb, díky kterým je možné relativně snadno vytvářet formulářové aplikace pro klasickou plochu systému Windows, složité a robustní webové aplikace a v neposlední řadě nové, univerzální aplikace. Všechny aplikace lze vytvářet v jednom společném vývojovém prostředí, které se nazývá Visual Studio. Součástí Visual Studia je nejen prostředí pro psaní kódu, ale také samotný kompilátor, tedy překladač ze zvoleného jazyka do společného jazyka, který je poté interpretován pomocí již zmíněného CLR. [4]

Jak již bylo řečeno, tento framework spouští aplikace ve virtuálním prostředí (CLR). Z toho důvodu vyžaduje, aby na koncovém zařízení byla nainstalována klientská část. Bohužel, tato část je dostupná pouze pro operační systém Windows a kvůli tomu není možné vytvářet aplikace v .NET Frameworku, které by byly oficiálně podporovány i jinými

operačními systémy. Tento fakt je důležitý, neboť to byl jeden důvodů pro vytvoření nového běhového prostředí .NET Core.

3.1.3 .NET Core

Jak je uvedeno v úvodu této práce, po velmi dlouhé době byl opuštěn model, respektující zpětnou kompatibilitu. Ta byla řadu let předtím vynucována, mnohdy i za relativně vysokou cenu. Výsledkem tak byl jeden velký framework, obsahující celou řadu funkcí zajišťujících kompatibilitu se staršími produkty. I to bylo motivací k ukončení současného přístupu, sestávajícího z postupného vylepšování již existujícího frameworku, a vytvoření zcela nové vývojové větve, obsahující zcela nový framework .NET Core.

Framework .NET Core je modulární verze .NET Frameworku, vytvořená především pro použití u webových aplikací, konkrétně tedy ASP.NET. Jedním ze základních vlastností je modularita. Jednotlivé funkce tak nejsou součástí jedné knihovny, ale jsou to samostatné jednotky, takzvané balíčky. Ty jsou do projektů stahovány pouze v případě potřeby. Tento přístup se tak snaží zabránit situaci, do které se dostala například již velmi dobře známá knihovna z .NET Frameworku System.Web, obsahující funkce, rozhraní a konfigurace hostovacího prostředí (serveru) a projektu dohromady. .NET Core tak dává vývojářům možnost agilnějšího vývoje tím, že umožní výběr pouze toho, co je opravdu v daném projektu potřebné.

V následující citaci je uvedena základní motivace pro vytvoření nového frameworku, převzatá z dokumentace.

> *„For each platform, a separate vertical stack consisting of runtime, framework, and app model is required to develop .NET applications. One of the primary goals of .NET Core is to provide a single, modular, cross-platform version of .NET that works the same across all of these platforms.“* [5]

Hlavním cílem tedy bylo vytvoření jednotného, modulárního a multiplatformního frameworku. Kromě toho ale bylo po nové platformě vyžadováno celkové zmenšení

velikosti a navázanosti na IIS[1]. To si lze reálně představit jako požadavek na odstranění již zmíněné knihovny System.Web, a funkcionality rozdělit do samostatných balíčků. Toho bylo mimo jiné docíleno rozdělením .NET Core na dvě základní knihovny [5], které budou podrobněji popsány v následujících podkapitolách.

1. .NET Core Libraries – CoreFX (kapitola 3.1.3.1)
2. .NET Core Common Language Runtime – CoreCLR (kapitola 3.1.3.2)

Neméně zajímavou skutečností je také fakt, že celá popisovaná nová platforma „Core“ je **zcela** Open-Source, přístupná pod MIT[2] licencí na GitHubu[3]. V dokumentaci na MSDN se přímo uvádí:

> *„In addition to the modularization of the .NET Framework, Microsoft is open-sourcing the .NET Core packages on GitHub, under the MIT license. This means you can clone the Git repo, read and compile the code and submit pull requests just like any other open source package you might find on GitHub.“* [29]

Z citace vyplívá, že samotní uživatelé mohou upravovat zdrojový kód, avšak ne každá změna se automaticky projeví v další verzi. Než je nějaká úprava kódu přijata, musí být nejdříve schválena autory. Hlavní důvod schvalovacího procesu je především snaha o udržení konzistence zdrojových kódů a udržení korektní funkcionality díla jako celku. Tímto krokem se tak společnost Microsoft snaží přiblížit ke svým uživatelům, jako to ostatně dělá i v jiných divizích.

[1] Internet Information Service (IIS) je aplikace, schopna hostovat webové aplikace na platformě .NET pod operačním systémem Windows. Jinými slovy se jedná o server, podobně jako například Apache.

[2] MIT je typ svobodné licence, která ve své podstatě dovoluje použití daného software zdarma.

[3] GitHub je online služba pro sdílení zdrojového kódu.

3.1.3.1 CoreFX

Bylo již uvedeno, že .NET Core sestává ze dvou základních částí. Jednou z nich je CoreFX, což je soubor komponent, konkrétně menších knihoven, vyžadujících jen minimum dalších referencí. Společně tyto knihovny dávají dohromady celý .NET Core framework a implementují tak zamýšlenou modularitu.

Obsahem CoreFX jsou různé základní funkce a objekty pro umožnění základních operací, jako například kolekce, přístup ke konzoli, diagnostiku, IO, LINQ, JSON, XML, a regulární výrazy. Tento výčet je však jen částečný. Kompletní popis je v době psaní tohoto textu dostupný především na GitHubu [6].

3.1.3.2 CoreCLR

V kapitole 3.1.2, popisující základy .NET Frameworku, je uvedeno, že veškerý kód je překládán do strojového, společného jazyka, který je následně interpretován společným CLR. V případě .NET Core existuje CoreCLR, které slouží tomu samému účelu. Zásadní rozdíl je v tom, že CoreCLR je již od začátku koncipován tak, aby mohl být spuštěn v libovolném operačním systému, a zároveň aby byl vždy součástí finálního balíčku společně s aplikací. To znamená, že při publikování webové stránky v ASP.NET Core je vytvořen nejen balíček s implementací oné webové stránky, ale zároveň lze připojit i runtime, který je schopný webovou stránku spustit. Tím se lze zbavit jakékoliv závislosti na již instalovaném software na konečném stroji. Druhá výhoda zahrnutí běhového prostředí do výsledného balíčku je, že není nutné řešit aktualizace hromadně. Každá webová aplikace může na jednom počítači fungovat na jiné verzi .NET Core.

CoreFX a **CoreCLR** jsou dostupné přes balíčkovací službu NuGet[4].

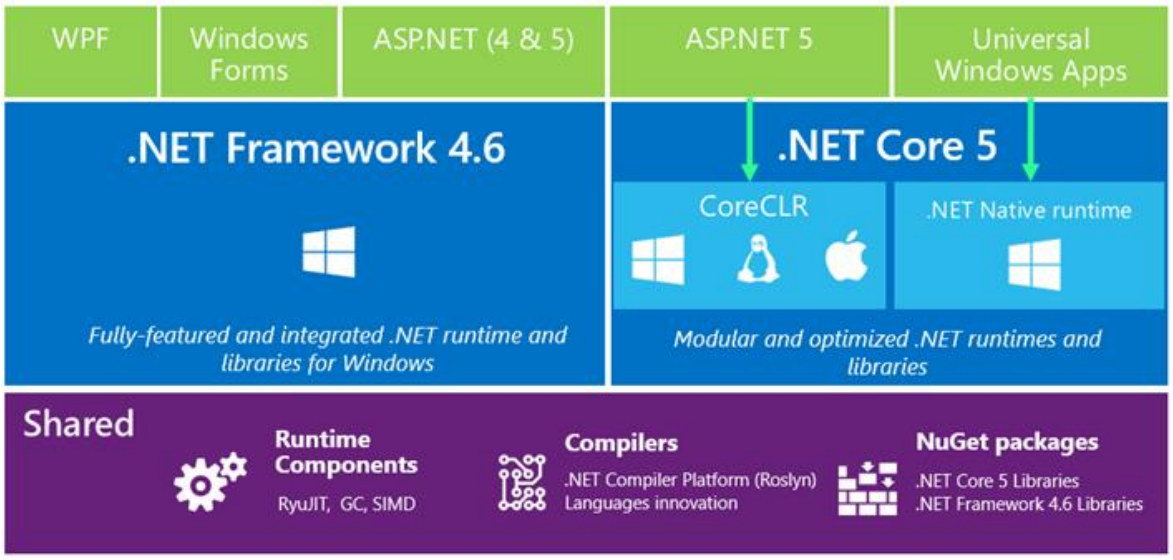
3.1.4 Sdílené části

Nový .NET Core framework nelze považovat za přímého nástupce současného .NET Frameworku, což mimo jiné dokazuje i fakt, že současný .NET Core zatím neobsahuje všechny funkce, které jsou v reálném prostředí potřeba. V době psaní této práce se stále

[4] NuGet je balíčkovací manažer pro vývojovou platformu společnosti Microsoft.

jedná o svým způsobem velmi odlehčenou verzi plnohodnotného frameworku, jak je uvedeno v [7]. I samotná společnost Microsoft upravila jmennou konvenci tak, aby nebyla matoucí a dávala jasnou zprávu - ***.NET Framework 4.6 zůstává a stále se bude vyvíjet. .NET Core je nová, svěží větev, určená především pro ASP.NET Core, avšak v současné době nemá ambice nahradit .NET Framework kompletně*** [1].

Znamená to tedy, že do budoucna lze očekávat další verze .NET Frameworku s novými funkcemi. Současné rozdělení může ještě dokreslit následující obrázek.



Obrázek 1 - Přehled .NET architektury (zdroj [7])

3.1.4.1 Kompilátor Roslyn

Oba výše popisované frameworky mají jednu zásadní společnou věc, a tou je kompilátor Roslyn. Ten zajišťuje kompilaci kódu do společného jazyka, intepretovaného pomocí CLR, respektive CoreCLR. Záleží jen na tom, zda je aplikace napsaná v .NET Framework 4.6 nebo .NET Core 1.0. Roslyn je součástí vývojového nástroje Visual Studio od verze 2015 [8].

3.2 Běhové prostředí DNX

S novým frameworkem přišel také zcela nový způsob práce se zdrojovými soubory, jejich kompilací a spouštěním. Je rozdělen do tří komponent.

- DNX – Běhové prostředí a SDK

- DNVM – Správa verzí DNX
- DNU – Služba pro vytváření balíčků

3.2.1 .NET Execution Environment (DNX)

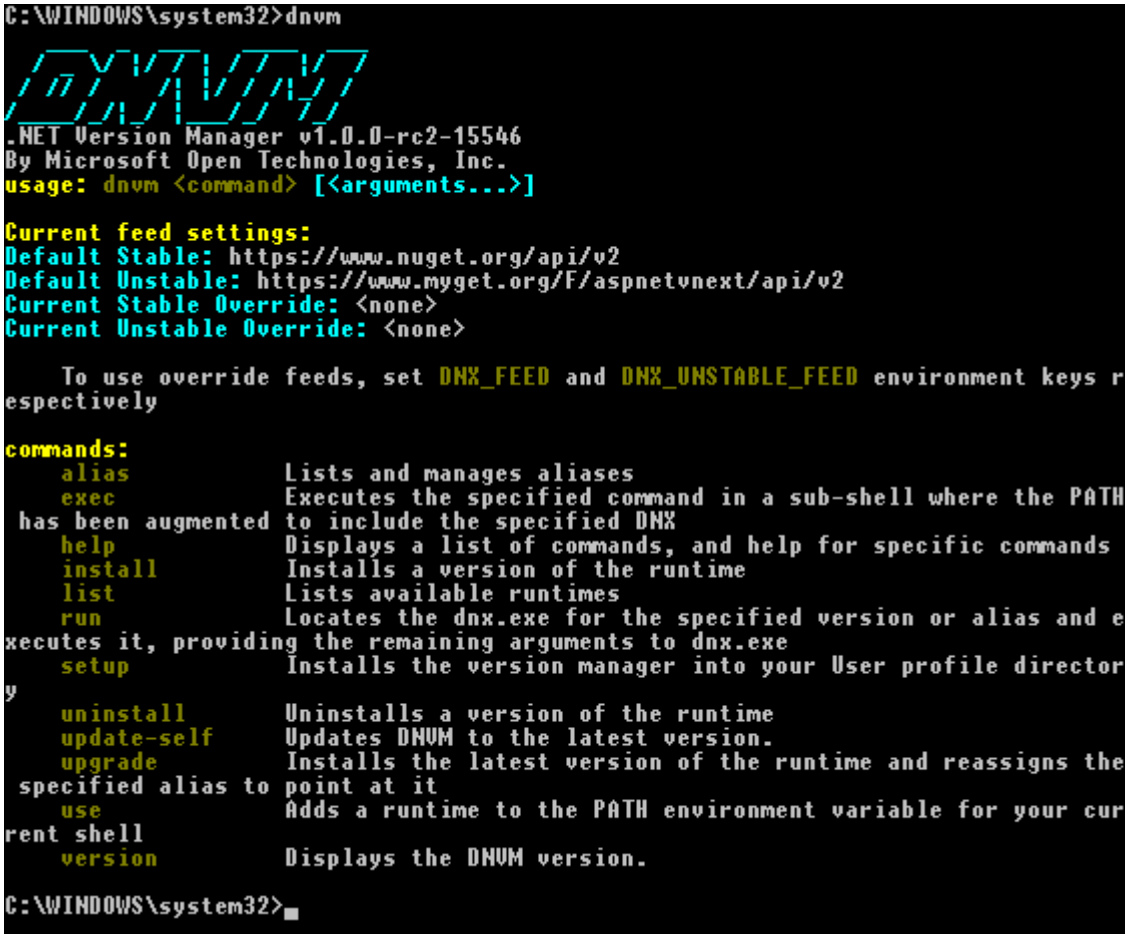
Běhové prostředí a kompilátor v jednom se nově nazývá DNX, což je zkratka pro .NET Execution Environment. Dovoluje kompilovat a spouštět libovolné .NET Core konzolové a webové aplikace a to dokonce v operačním systému Widnows, Mac a Linux. Je však důležité podotknout, že aplikace pro .NET Framework 4.6 jsou také spustitelné a kompilovatelné přes DNX, ale **pouze na operačním systému Widnows s nainstalovaným plnohodnotným .NET Frameworkem**. Na zbylých dvou operačních systémech lze spouštět a kompilovat pouze aplikace vytvořené ve frameworku .NET Core. Pokud je tato podmínka splněna, je možné aplikace kompilovat a vyvíjet na jednom operačním systému, a zároveň je spouštět na jiném.

DNX obsahuje hostovací proces pro aplikaci, CLR (respektive CoreCLR) prostředí a zajistí nalezení hlavního vstupního bodu aplikace, což je u konzolových aplikací metoda Main a u webových aplikací je to třída Startup. Neméně důležitá novinka je také fakt, že všechny konfigurační soubory DNX opustili formát XML a využívají modernější, odlehčenější a přehlednější formát JSON.

3.2.1.1 Spuštění DNX

Pro spuštění DNX je nejdříve nutné nainstalovat si toto běhové prostředí do PC, nebo mít k dispozici balíček aplikace s přiloženým běhovým prostředím. Detailní a aktuální popis instalace na Windows, Mac i Linux, je k dispozici na [9]. Následující ukázky budou předváděny z prostředí Windows. Jedná se vždy o práci s příkazovým řádkem, tudíž nemá smysl demonstrovat následující ukázky pro všechny dostupné platformy. Až na výjimky jsou postupy stejné.

Základní přístup je, jak již bylo řečeno, přes příkazový řádek zvoleného operačního systému. Po instalaci lze tedy otevřít konzoli a zadat příkaz „dnvm“. Po potvrzení by mělo být vykresleno to, co je na obrázku Obrázek 2. Jedná se o vstupní bod do DNX. Z tohoto prostředí lze instalovat různé verze frameworků, určené pro různé procesorové architektury. Zároveň lze ovládat aktuální běhové prostředí DNX a spouštět jeho aktualizace a upgrade.



Obrázek 2 - Úvod do DNVM, zdroj vlastní

Po instalaci DNX přes DNVM jej lze spustit příkazem „dnx“. To je celé. V případě, že se kurzor nachází ve složce, obsahující níže popsaný soubor project.json, lze tak aplikaci spustit zadáním příkazu „**dnx run**“ (konzolová aplikace) či „**dnx web**“ (webová aplikace).

3.2.1.2 DNX projekt

Projekt pro DNX je tvořen vždy adresářem se souborem *project.json*. Uvedený soubor je primární konfigurační zdroj. Lze v něm mimo jiné nastavit, (přidat) na které frameworky se má projekt kompilovat, jaké vazby na externí knihovny vyžaduje a v neposlední řadě také další metadata, která se využívají při balíčkování aplikace pomocí NuGet.

```
{
  "version": "1.0.0-*",
  "compilationOptions": {
    "emitEntryPoint": true
  },

  "dependencies": {
    "Newtonsoft.Json": "8.0.2"
  },

  "frameworks": {
    "dnx451": {
      "dependencies": {
        "Scms.DataApi": "1.0.0-*"
      }
    },
    "dnxcore50": { }
  }
}
```

Ukázka kódu 1 - Struktura konfiguračního souboru project.json

Při podrobnějším pohledu na sekci „frameworks“ v ukázce Ukázka kódu 1 nemůže pozornému čtenáři uniknout, že obsahuje vnořené sekce, pojmenované dle frameworku. Právě na tomto místě je totiž možné zvolit, pro které frameworky má být DNX kompilován. Mimo to je zde však také možné nastavit specifické závislosti na knihovnách pro každý Framework zvlášť.

Uvedená ukázka obsahuje závislost na balíčku Newtonsoft.Json (v sekci „Dependencies“). Tato závislost platí pro všechny frameworky v projektu. Avšak v sekci frameworks je pro „dnx451“ nastavena ještě speciální závislost na balíčku Scms.DataApi. Tato závislost však pro „dnxcore50“ vyžadována není.

3.2.2 .NET Version Manager (DNVM)

Vzhledem k faktu, že v současné době existují dvě hlavní větve .NET, popsané v kapitole 3.1, a obě jsou samostatně verzovány, bylo vhodné vytvořit nástroj, který usnadní správu různých typů a verzí frameworku. Tento nástroj se nazývá DNVM, což je zkratka pro Dot Net Version Manager.

3.2.3 .NET Development Utility (DNU)

Část DNX, která je zodpovědná za přípravu finálního balíčku aplikace, se jmenuje DNU. Ta ze souboru project.json zjistí základní informace, jako je jméno aplikace, verze, závislosti na knihovnách a několik dalších meta informací. Z dostupných dat poté vytvoří finální balíček (adresář), obsahující NuGet balíček, který lze rovnou distribuovat na web.

3.3 ASP.NET Core

S výše představeným frameworkem .NET Core je velmi úzce svázán rozšiřující framework ASP.NET Core, což je sada rozšíření nad .NET Core frameworkem pro použití v prostředí webových aplikací. ASP.NET Core 1.0 je oproti svým předchozím verzím vytvořen zcela od začátku, stejně jako výše popisovaný .NET Core. Sestává z celé řady různých komponent, díky čemuž si lze naprosto přesně zvolit pouze ty části ASP.NET, které jsou nezbytně nutné pro běh aplikace. To přináší mimo jiné výhodu v malé velikosti výsledné aplikace, neboť zdrojové soubory obsahují pouze minimum nevyužité funkcionality.

3.3.1 Struktura ASP.NET Core 1.0 projektu

ASP.NET Core projekt je ve své podstatě vytvořen na základě DNX projektu, jehož struktura je detailněji popsána v podkapitole 3.2.1. To znamená, že ASP.NET Core projekt je adresář, obsahující *project.json* soubor s definicí projektu. Mimo tento soubor ale obsahuje ještě celou řadu dalších specifik, které budou podrobněji popsány.

3.3.1.1 Adresář wwwroot

V předchozích verzích ASP.NET byl kořenový adresář webové stránky shodný s adresářem webové aplikace. Díky tomu stačilo vždy jakýkoliv soubor přidat do tohoto kořenového adresáře, čímž se tento soubor stal dostupným na patřičné webové adrese [10]. Díky existenci routování však toto nebylo nutné pro serverem zpracovávané soubory. Například soubory s implementací controlleru mohly být ve zvláštním adresáři, který nemusel nutně korespondovat s adresou, kterou byl k tomuto souboru umožněn přístup. Toto řešení však mělo jednu zásadní nevýhodu a tou byl fakt, že byly takto smíchány kompilované soubory webové aplikace a statické zdrojové soubory jako obrázky, css a podobné.

V ASP.NET Core je proto statický obsah webové stránky striktně oddělen ve vlastní složce wwwroot. Díky této separaci již není nutné připravovat složitá bezpečností pravidla pro přístup do kořenového adresáře webu, jako například znepřístupňování souborů jako web.config, global.asax a další neveřejné soubory. Adresář wwroot tak nově reprezentuje skutečný kořenový adresář webové aplikace běžící na serveru. Statické soubory, které nejsou ve složce wwwroot, tak nebudou nikdy přístupné, a není nutné vytvářet speciální pravidla pro soubory s určitou příponou, či zvláštním názvem.

Název wwroot je konfigurovatelný a lze ho v souboru project.json upravit.

3.3.1.2 Správa klientských balíčků

Dalším novým adresářem v projektu ASP.NET je adresář Dependencyes. Ten se skládá ze dvou podsložek: **Bower** a **npm**. Každý z nich reprezentuje balíčkovací systém (viz. kapitola 3.3.6) stejného názvu. Oba jsou používány pro stahování klientských balíčků. Po otevření každého z adresářů se zobrazí seznam nainstalovaných balíčků. Zatímco adresář Bower obsahuje JavaScriptové balíčky pro klientskou část webové aplikace, balíčkovací systém npm se stará o balíčky JavaScriptových frameworků, které jsou využívány například během kompilace webové aplikace.

Výhoda těchto balíčkovacích systémů je nejen přehledné zobrazení závislostí na určitých balíčkách, ale zároveň udržování grafu závislostí. Ten pomáhá při stahování nových balíčků a porovnávání závislostí i těch, které již jsou součástí projektu.

Samotné obsahy Bower balíčků jsou k nalezení v adresáři wwwroot\lib. Každý balíček je v podadresáři dle svého názvu.

3.3.1.3 Appsettings.json

Náhradou za již velmi dobře známý konfigurační soubor web.config u starších ASP.NET aplikací je application.json. Oproti svému předchůdci se liší nejen v použitém formátu (JSON versus XML), ale především potom ve způsobu zpracování a čtení. Způsob čtení, a vůbec práce s konfigurací aplikace, je nově v .NET Core také zcela přepracován.

3.3.1.4 Gulpfile.js

Mezi další zásadní novinky ASP.NET Core projektů patří využití automatizačního JavaScriptového frameworku, známého pod jménem Gulp. Gulp je v tomto případě využíván například po minifikaci a sjednocení JavaScriptových a css souborů. Nespornými výhodami tohoto přístupu jsou bezesporu možnosti personifikace a rozšíření automatizovaných procesů oproti předchozímu řešení, jehož možnosti nastavení byly mnohem omezenější [11]. Více je Gulp popsán v kapitole 3.3.5.

3.3.2 **Dependency Injection**

Díky tomu, že je ASP.NET Core vytvořeno zcela od začátku, bylo možné vše navrhnout s podporou Dependency Injection. To umožňuje všem součástem tohoto frameworku odstranit závislost na konkrétní implementaci určité služby. Tím je dosaženo nejen skvělých možností při testování, ale také je takto zásadním způsobem zlepšena přehlednost zdrojového kódu.

3.3.2.1 Funkce Dependency Injection

Dependency Injection je technika OOP, někdy označována také jako Dependency Inversion Principle [12] (dále jen DI), pro docílení nepřímé vazby mezi objekty [13]. Místo přímého odkazování na instancované objekty nebo využívání statických tříd je vytvořena pouze nepřímá vazba prostřednictvím operačního kontraktu, jinak také známého jako **interface**. V případech, kdy je tato vazba na jiný objekt předávána v konstruktoru objektu, se nazývá Explicit Dependency Principle, jinak známý také jako „constructor injection“.

> *„Methods and classes should explicitly require (typically through method parameters or constructor parameters) any collaborating objects they need in order to function correctly.“* [31]

V zásadě tento princip říká, že pokud má některá třída provádět určitou práci, k jejímuž zpracování potřebuje jinou službu, měla by si její závislost vyžádat již při instancování. Explicitními se tyto závislosti označují proto, že realizují vazby napříč objekty, a ne závislosti uvnitř objektu, též známé jako implicitní závislosti. Pokud je tedy jakákoliv

třída navrhována pro DI, nemá přímou vazbu na žádnou jinou třídu nebo statický objekt. Vše je řešeno předáváním daného abstraktního rozhraní, typicky právě interface, do konstruktoru. Samotná implementace je dodána v době konstruování objektu.

Při využití DI principu je ale důležité si uvědomit, že musí existovat nějaký nástroj, který při konstruování objektu dodá potřebné implementace abstraktních DI objektů. Tento nástroj je označován jako **IoC (Inversion of Control) kontejner**, nebo také Dependency Injection Container. Svým způsobem je IoC „továrna“ objektů, implementující určitá abstraktní rozhraní. Tato „továrna“ ale musí objekty nejen umět vytvářet, ale zároveň je musí být schopna ve správnou chvíli také předat objektu, který implementaci dané abstrakce vyžaduje. Tento proces, kdy IoC kontejner předává implementaci nějaké interface do třídy objektu, se nazývá injection (česky patrně nejblíže „vkládání“).

IoC kontejnery už díky své povaze a způsobu použití dokáží sestavit složité závislostní grafy, vyjadřující které třídy jsou závislé na kterých jiných třídách. Takovýto graf samozřejmě může u větších programů dosahovat obrovských rozměrů.

Kromě závislostních grafů se ale IoC kontejnery starají ještě o jednu zásadní věc a tou je životní cyklus objektů. Životním cyklem je myšlena doba od zažádání o instanci objektu až po jeho odstranění z paměti počítače. IoC kontejner lze při startu aplikace nakonfigurovat tak, aby různé implementace předával s různými životními cykly. Například služba pro logování událostí do souborů může mít Singleton[5] povahu, a stačí ji tak instancovat pouze jednou. Oproti tomu například služba pro přístup do databáze může mít nastaveno, že při každém vyžádání pomocí DI objektem vydá IoC kontejner novou instanci.

ASP.NET Core 1.0 nabízí již integrovanou podporu pro IoC kontejner a DI. Jedná se o základní implementaci, která podporuje pouze již zmíněný „constructor injection“, tedy vkládání instancí do DI objektů skrz jejich konstruktor. Tato služba je reprezentována pomocí interface IServiceProvider. V ASP.NET Core je na instance předávané IoC kontejnerem pohlíženo jako na **služby** (services). V následujících kapitolách tak bude slovo

[5] Singleton znamená takovou instanci objektu, jejíž životní cyklus je stejný jako životní cyklus aplikace.

„služby“ odkazovat na instance objektů, vytvořené pomocí ASP.NET IoC integrovaného kontejneru pro potřeby DI.

3.3.2.2 Druhy instancí

ASP.NET služby DI mohou být konfigurovány s následujícími druhy životních cyklů objektů.

Životní cyklus **Transient** vytváří novou instanci objektu, kdykoliv je vyžadován. Tento životní cyklus je nejvhodnější pro malé, bezstavové služby.

Scoped životní cyklus je takový, který vytváří novou instanci objektu pro každý nový HTTP požadavek. Jedná se tedy o životní cyklus začínající při zpracování požadavku a končící při odesílání odpovědi. Je vhodný pro služby, vyžadující shodný kontext vždy v rámci jednoho HTTP požadavku.

Singletone životní cyklus vytváří instanci objektu při prvním použití. Každý další požadavek na tento objekt tak vrátí vždy stejnou instanci. Díky tomu lze tento životní cyklus použít na služby, které vyžadují udržování stavu i mezi různými HTTP požadavky.

Poslední možným životním cyklem je **Instance.** Při konfiguraci životního cyklu lze IoC kontejneru předat konkrétní instanci, která bude následně vždy vrácena. Svým způsobem se jedná o předem definovanou Singleton instanci. Hlavním rozdílem je fakt, že Singleton instance jsou vytvářeny až v případě prvního použití (nemusí tak být instancovány vždy), kdežto Instance služba je vytvořena na začátku běhu aplikace.

3.3.2.3 Nastavení implementací pro interface

Následující ukázka kódu demonstruje nastavení různých typů služeb v metodě Configure Services ze souboru Startup.cs, popsaného v kapitole 3.3.3. Jedná se o ukázkové nastavení různých druhů implementací (DataStore) rozhraní IDataStore.

```
// Nová instance bude předána při každém vyžádání.
services.AddTransient<IDataStore, DataStore>();

// Nová instance je vytvořena pro každý HTTP Request.
services.AddScoped<IDataStore, DataStore>();

// Po vyžádání je vytvořena jedna instance, která je dále používána.
services.AddSingleton<IDataStore, DataStore>();

// Stejné jako Singleton, ale instance je vytvořena
// ihned (nehledě na vyžádání) a jedná s. o tu instanci,
// která je předána jako argument metody.
services.AddInstance<IDataStore>(new DataStore());
```

Ukázka kódu 2 - Demonstrace nastavení služeb pro IoC kontejner

3.3.2.4 Použití

Použití IoC kontejneru v ASP.NET je velmi jednoduché. Jak je uvedeno v předchozích kapitolách, tato jednoduchá realizace IoC kontejneru umožnuje pouze „constructor injection“, tedy předávání instancí pomocí konstruktoru objektu. Pokud tedy některý z objektů vyžaduje funkcionalitu některé nastavené služby, argumentem jeho konstruktoru musí být interface této služby.

```
private IDataStore dataStore;
private IWebService webService;
public HomeController(IDataStore store, IWebService service)
{
    dataStore = store;
    webService = service;
}
```

Ukázka kódu 3 - Demonstrace využití DI

Na ukázce kódu Ukázka kódu 3 je implementace konstruktoru třídy HomeController. Tento konstruktor obsahuje dva argumenty: Interface IDataStore a interface IWebService. Tyto předané argumenty si poté uloží do vnitřních proměnných. To, zda bude při každém požadavku předána stejná instance, záleží na tom, zda je služba nakonfigurována jako Transient, Scoped, Singleton nebo Instance.

3.3.3 **Startup soubor**

Hlavní vstupní bod do jakéhokoliv programu či aplikace je naprosto zásadní záležitost. U konzolových aplikací je tímto bodem statická funkce Main, která přijímá argumenty předané příkazovou řádkou. Nicméně webová aplikace DNX nemusí být nutně spuštěna přes příkazovou řádku. Navíc, konfigurace a nastavení webové aplikace je složitostně mnohem dále než obyčejná konzolová aplikace.

U webových aplikací nejnovějšího ASP.NET Core je stejně jako u již staršího OWIN přístupu jeden soubor, který je běhovým prostředím automaticky nalezen v projektu a spuštěn jeho obsah. Tento soubor se jmenuje **Startup.cs.** Neobsahuje nic jiného než Startup třídu, mající hned několik metod pro konfiguraci různých částí webové aplikace. [14]

3.3.3.1 Metoda Configure

Ke konfiguraci obsluhování požadavků protokolu HTTP slouží metoda Configure. Jako argument vyžaduje implementaci IApplicationBuilder, IHostingEnvironment a ILoggerFacotry. Všechny tyto instance jsou získány přes DI. Obsahem této metody pak mohou být aktivace a konfigurace chybových stránek, obsluha statických souborů (obrázky, css styly a pod.), MVC a v neposlední řadě také aktivace Identity frameworku pro správu uživatelů a rolí. [14]

3.3.3.2 Metoda ConfigureServices

Třída Startup může obsahovat také metodu ConfigureService. Jejím primárním účelem je nastavení a konfigurace Dependency injection. Tato metoda je volána ještě před voláním Configure metody, což je důležité, neboť právě v této metodě se nastavují implementace interface, která má později DI vkládat do konstruktorů ostatních objektů, jako je tomu i v případě metody Configure. Podrobnější popis fungování DI v ASP.NET je popsán v kapitole 3.3.2.

3.3.4 **Logování**

ASP.NET Core má nově již zabudované služby pro logování dat. Logováním je v tomto případě myšleno ukládání určitých dat, informujících o běhu aplikace, do některého

z výstupů, většinou například do souboru [15]. Vyžaduje minimální nastavení, po kterém lze tuto službu využít v kdekoliv v projektu pomocí DI.

Přidání logovací služby do kterékoliv třídy lze zajistit přidáním argumentů typu ILoggerFactory nebo ILogger<T>, jejichž instance budou následně tomuto objektu předány pomocí již popsaného Dependency Injection (viz. 3.3.2).

```
loggerFactory.AddConsole(minLevel: LogLevel.Verbose);
```

Ukázka kódu 4 - Aktivace logování do konzole

V souboru Startup.cs (viz. 3.3.3) je možné v metodě Configure nastavit defaultní logování například do výstupu konzole, jako je to předvedeno v Ukázka kódu 4. Toho je využito i v projektu, který je součástí této práce.

3.3.5 Gulp

Tato kapitola je zpracována podle [16] a [11].

V moderních webových aplikacích je zcela běžné, že pro vytvoření balíčku není nutné pouze zkompilovat a zabalit zdrojové soubory v jazyce C#. Součástí balíčkování může být celá řada různých procesů. Tyto procesy je vhodné automatizovat, a přesně pro tento účel existuje nástroj Gulp.

Gulp je nástroj napsaný v jazyce JavaScript, použitelný pro automatizaci úkonů souvisejících s takzvanými „client-side“ soubory. To jsou soubory, jako například ty s příponou .css či .js, které jsou určeny pro uživatele webové aplikace. Gulp je nejčastěji používán například pro následující:

- Provedení minifikace[6] JavaScript a CSS souborů
- Sjednocení více souborů do jednoho
- Kompilace LESS na CSS
- Kompilace CoffeScript nebo TypeScript do JavaScript

[6] Minifikovaný soubor je takový, který byl zbaven všech komentářů a zbytečných netisknutelných znaků, a neveřejné části kódu byly přejmenovány na krátké názvy.

Výhody Gulp jsou především automatizace běžně prováděných úloh, zjednodušení opakovaných úkonů a snížení celkového času nutného k vývoji. Jak uvádí Matthew Jones na svém webu:

> *„Gulp.js provides a set of automation tools that allow us developers to automate common processes, such as bundling and minification of CSS and JS files.“* [11]

Ve VisualStudio 2015 je konfigurační soubor součástí defaultní struktury MVC projektu. Jmenuje se gulpfile.js a obsahuje defaultně již zmíněnou minifikaci a zabalení CSS a JavaScript souborů.

3.3.6 **Balíčkovací systémy**

V této práci je velmi často uváděno slovo „balíček“, nebo sousloví „NuGet balíček“ či „bower balíček“. Vše vždy odkazuje na soubor, vytvořený některým z balíčkovacích systémů, jako například NuGet, Bower, či npm. Každý ze systémů udržuje jiný druh balíčků.

3.3.6.1 Balíček

Slovo balíček (anglicky package) označuje většinou zkomprimovaný adresář obsahující zdrojové soubory k nějaké službě, knihovně či frameworku. Jeho obsahem tak může být téměř cokoliv. Různé balíčkovací systémy mají vlastní strukturu a pravidla pro vytváření balíčků. Obecně lze však všechny shrnout pod jeden název, neboť se vždy jedná o soubor zdrojových souborů a metadat, určených k instalaci do jiných projektů.

3.3.6.2 NuGet

Balíčkovací systém NuGet byl vytvořen pro usnadnění práce s externími knihovnami .NET frameworku. Je zakomponován přímo do vývojového Visual Studia, kde lze pro každý projekt spravovat celou řadu „balíčků“, reprezentující různé zkompilované knihovny třetích stran [17]. Pro přidání reference (závislosti) na zvolené knihovny tak není nutné manuálně

přidávat záznam do souboru závislostí projektu .NET frameworku. Stačí přes uživatelské rozhraní vyhledat balíček a spustit instalaci.

Instalace balíčku zařídí nastavení závislostí na všechny potřebné doplňkové balíčky (takzvané zjištění vnořených závislostí). To usnadňuje vývojářům práci v tom, že nemusí tyto vazby ručně dohledávat. NuGet však není omezen pouze na knihovny s příponou dll. Lze také do balíčku přidat libovolné další soubory a nastavení, které se při instalaci do projektu zkopírují.

Pro .NET Core je balíčkovací služba NuGet stěžejním distribučním prvkem [17]. Už samotný soubor project.json, což je základní definice projektu, obsahuje informace pro vytvoření NuGet balíčku z daného projektu (kapitola 3.3.1). Nově již není možné k .NET Core projektu přidat závislost na projekt typu „Class library“ (knihovna), ale lze přidat pouze odkaz na balíček typu NuGet. Z tohoto důvodu byly v .NET Core nahrazeny „Class library“ projekty balíčky NuGet.

3.4 MVC 6

Architektura MVC a její základní vlastnosti jsou popsány níže v práci, v kapitole 3.5.2. Označení MVC 6 reprezentuje využití MVC architektury pro návrh a vývoj webové aplikace v novém vývojovém prostředí. Veškerý základ pro MVC 6 je již obsažen v ASP.NET Core 1.0 frameworku a MVC 6 je pouze určitý druh nadstavby nad tímto frameworkem.

Mezi hlavní vlastnosti této nadstavby patří konfigurace zpracování HTTP požadavků, instancování správného controlleru, zavolání správné metody a zpracování View. To vše dle zadané URL adresy. Tato funkcionalita vychází z již samotné MVC architektury, jejíž detailní popis fungování je k dispozici v dokumentaci [18].

3.4.1 TagHelpers

Nová funkcionalita, která je specifická pouze pro MVC 6, jsou TagHelpers [19].

TagHelpers umožňují zapojení serverového kódu při vytváření a vykreslování HTML elementů v souborech typu cshtml (Razor[7]). Například vestavěný TagHelper ImageTagHelper dokáže připojit verzi k libovolné adrese obrázku. Kdykoliv se obrázek změní, server vygeneruje kód nové verze. To v konečném důsledku zajistí, že uživateli se zobrazí vždy aktuální obrázek. V případě, že se změní (jakkoliv) url obrázku, už totiž nemůže prohlížeč využít dříve uloženou (cashed) verzi.

V MVC 6 existuje celá řada podobných TagHelpers, usnadňující práci na mnoha místech. Ve své podstatě jsou TagHelpers náhradou za již zavedené pomocné metody.

```
<!-- Starší využití HtmlHelpers -->
@Html.ActionLink("Produkty", "Index", "Products")

<!-- Využití nových TagHelpers -->
<a asp-action="Index" asp-controller="Products">Produkty</a>

<!— Obojí uvedené produkuje následující kód -->
<a href="/Products">Produkty</a>
```

Ukázka kódu 5 - Ukázka TagHelpers

Na ukázce kódu Ukázka kódu 5 je demonstrováno nejdříve využití doposud užívaného přístupu pro generování odkazů, a následně nový způsob pomocí TagHelpers. První i druhý způsob však produkuje stejný výsledek, a je pouze na vývojáři, který z nich zvolí. Oba přístupy lze také samozřejmě kombinovat, jelikož TagHelpers nejsou náhradou za doposud využívané HtmlHelpers, ale jedná se pouze o alternativu [19].

3.5 Návrhové vzory a principy

Součástí praktické části této práce je mimo jiné také návrh vlastní architektury datové základny. V následujících kapitolách budou přiblíženy již existující architektury, návrhové vzory a principy, z nichž autor při návrhu vycházel.

[7] Razor je syntaktický nástroj, umožňující kombinaci server-side (C#) a client-side (HTML) kódu.

3.5.1 SOLID principy

Tato kapitola a její podkapitoly jsou primárně zpracovány podle [20], [21] a [12].

Pro návrh struktury a architektury jakéhokoli programu či aplikace je k dispozici obrovská škála materiálů o tom, jak vlastně takový program či aplikaci psát. Mezi v dnešní době nejrozšířenější přístup patří bezpochyby OOP – Object Oriented Programming. Ten je samozřejmě respektován ve všech .NET Framework aplikacích a programech.

Sám o sobě OOP určuje možnosti strukturování a tvarování programu. Definuje obecný přístup k zapouzdřování a separaci kódu. Nicméně to neznamená, že v OOP nelze vytvářet špatný, nelogický a obtížně udržitelný kód. A právě to byla jedna z hlavních motivací při definici **SOLID** principů programování. Je důležité upozornit na fakt, že SOLID principy nejsou to samé jako návrhové vzory. Návrhové vzory jsou již uceleně navržené struktury a návody na přístup k tvorbě specifických typů systémů. Oproti tomu SOLID principy jsou pouze pravidla, na kterých by měl být každý dobrý program či aplikace postaven a která by měl každý programátor či softwarový architekt dodržovat.

Zkratka SOLID se skládá z následujícího:

- **S** jako **S**ingle Responsibility Principle (SRP)
- **O** jako **O**pen/Closed Principle (OCP)
- **L** jako **L**iskov Substitotion Principle (LSP)
- **I** jako **I**nterface Segregation Principle (ISP)
- **D** jako **D**ependency Injection Principle (DIP)

3.5.1.1 Single Responsibility Principle

První princip, patřící mezi SOLID, je Single Responsibility Principle. Toto lze velmi volně přeložit jako „princip odpovědnosti za jednu funkci“. Jedná se o princip definující, že by každá třída měla být zodpovědná pouze za jednu funkční jednotku. Jinými slovy lze tento princip popsat jako takový způsob návrhu třídy objektu, kdy pro jakoukoliv její úpravu nesmí být více než jeden důvod. Jak je uvedeno v knize The SOLID Principles od Guarav a spol.:

> *„Class should be designed for single responsibility and there should not be more than one reasons to make changes in this class. The responsibility of this class should be completely tide/encapsulated by the class.“* [12]

Nebo jak uvádí Patkos Csaba ve svém článku:

> *„A class should have only one reason to change.“* [32]

Proč je však tento princip tak důležitý? Neboť zajistí, že každá třída se stará vždy jen o tu funkcionalitu, o kterou má. Žádná třída by neměla obsahovat celou logiku aplikace. Každá struktura aplikace je složena z několika vrstev a funkčních bloků. Každý z nich by měl být oddělen ve vlastní třídě. Jen tak se totiž předejde možným chybám při pozdější změně požadavků a úpravách třídy. Třídy, obsahující velké množství funkčních bloků jsou v budoucnu mnohem náchylnější k chybám.

Předpokládejme existenci třídy, která obstarává připojení do databáze, získání dat a jejich následné uložení. Nejen že takováto třída může narůst do velkých rozměrů, ale například i menší změny v přístupu do databáze mohou vyústit v rozsáhlé úpravy této třídy, které se mohou omylem dotknout i již existujících business pravidel.

Pokud však bude při návrhu vždy brán ohled na SRP, výrazně se sníží náchylnost k budoucím chybám a zároveň se zlepší přehlednost kódu.

3.5.1.2 Open/Closed Principle

Mezi další principy SOLID patří takzvaný Open/Closed Principle. Definice tohoto principu je následující.

> *„Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.“* [20]

Volně přeloženo – Softwarové entity (třídy, moduly, funkce, apod.) by měly být otevřené pro rozšíření, ale uzavřené pro editaci. Jinými slovy to znamená, že entita by měla být editovatelná pomocí přidání dalšího rozšíření, avšak bez nutnosti upravovat či měnit původní zdrojový kód.

Otevřené pro rozšíření reálně v jazyce C# znamená, že implementovaná funkcionalita třídy by měla být dále rozšiřitelná děděním z této základní třídy. Jádro třídy by mělo při každé editaci zůstat netknuté. Typickým příkladem může být filtrování obsahu dle různých pravidel. Pokud bude existovat třída CarsFilter, mající na starosti filtrování určitých objektů typu Car, potom by měla být rozšířená například třídami CarsFilterByName, CarsFilterByColor a podobně, Důležité je, aby třída CarsFilter fungovala jako základní třída, obsahující abstrahovanou funkcionalitu. Detailní implementace pak zařídí dědící třídy.

3.5.1.3 Liskov Substitution Principle

Další z významných principů je označován Liskov Substitution Principle. Oficiální definice Barbara Liskova z roku 1987 je složitější, avšak Robert C. Martin ji ve své knize Agile Principles, Patterns, and Practices in C# popsal slovy:

> *„Subtypes must be substitutable for their base types.* [33]

V překladu si toto pravidlo lze vyložit jako „Podtypy musí být zaměnitelné za své nadtypy“. Toto pravidlo vychází z již samotné podstaty generalizace v OOP [21]. Podtypy by měly být vždy speciální případy svých nadtypů. Třída Vehicle (Vozidlo) tak může být základní typ pro třídy Car, Plane a Ship (Auto, Letadlo a Loď), neboť se jedná o speciální případy nadřazené entity. Nikdy by neměla být třída Vehicle generalizací například třídy Building (Budova), neboť budova není speciální případ vozidla. Toto v praxi bohužel často nastává z toho důvodu, že kupříkladu třída Vozidlo má určité vlastnosti, které jsou vyžadovány i ve třídě Building. Nicméně v tomto případě jsou porušeny i další principy SOLID.

Reálným příkladem může být například třída SenderBase, ze které vychází třídy EmailSender a SmsSender. Díky dodržení LSP jsou speciální typy logicky zaměnitelné za

děděnou třídu. Díky tomu mohou ostatní třídy pracovat pouze s třídu SenderBase a o detailní implementaci se nezajímat.

3.5.1.4 Interface Segregation Principle

Ve všech modulárních aplikacích musí existovat nějaký druh komunikačního protokolu, který může klientská část využít. V jazyce C# se tímto komunikačním protokolem (někdy také servisním kontraktem) označuje typ interface. Metoda či objekt, který jako svůj vstupní argument akceptuje interface, se tak nezavazuje k příjmu jedné určité implementace, ale k příjmu „čehokoliv, co má vlastnosti tohoto interface“.

V některých případech, zvláště potom v robustních aplikacích, je interface využíván pro celou řadu služeb a zastřešuje značnou část funkcionality. To samozřejmě většinou porušuje již popsané pravidlo SRP. Vzniká tak rozhraní s obrovským množstvím metod. Následně, pokud chce kdokoliv vytvořit třídu implementující toto interface, tak je nutné implementovat všechny jeho metody. A to i v případech, kdy je pro správnou funkci vyžadována pouze určitá podmnožina metod. Výsledkem tak bývá částečně implementovaná třída, kde velká část „implementovaných“ metod obsahuje pouze vytvoření výjimky s informací, že tato metoda nebyla reálně implementována.

Tento problém řeší právě ISP princip, který dle definice říká:

> *„The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.“* [33]

Dodržením tohoto principu se mimo jiné snižuje náročnost implementace interface a zefektivňuje se práce vývojářů, kteří musí takovéto interface implementovat.

3.5.1.5 Dependency Inversion Principle

Poslední, ale neméně důležitý princip SOLID programování, je Dependency Inversion Principle. Při dodržení všech předchozích principů je zde velká pravděpodobnost, že zdrojový kód bude připraven na IoC (Inversion of Control). Jedná se způsob návrhu architektury, kdy jednotlivé třídy nevolají instance jiných tříd přímo, ale tyto instance jsou do tříd takzvané „injektovány“ většinou při jejich inicializaci.

Přesná definice DIP od Robert C. Martin uvedená v článku od Patkos Csaba:

> *„A. High-level modules should not depend on low-level modules. Both should depend on abstractions. B. Abstractions should not depend upon details. Details should depend upon abstractions.“* [34]

Více o reálné funkci Dependency Injection (Dependency Inversion Principle) je v kapitole 3.3.2, kde je popsán již konkrétní případ a způsob využití v ASP.NET Core.

3.5.2 **MVC**

Jedna z nejznámějších architektur pro strukturování webových aplikací je MVC. Tuto architekturu respektuje celá řada platforem pro vývoj webových stránek, jako je například Zend, Nette, Ruby on Rails [22], a především potom ASP.NET. Ve své podstatě MVC nabízí oddělení logiky (business pravidel) aplikace od jejího výstupu a vzhledu. Struktura webové stránky je tak rozdělena do tří základních kategorií – **Model**, **View** a **Controller**.

3.5.2.1 Model

První zmíněný je Model. Ten je zodpovědný za samotnou logiku aplikace. Obsahuje například přístupy do databáze, třídy pro práci s entitami a business pravidla nutná pro korektní fungování webové aplikace. Důležitá je zde míra abstrakce, díky které není Model přímo napojen na výstupy pro koncového uživatele.

3.5.2.2 View

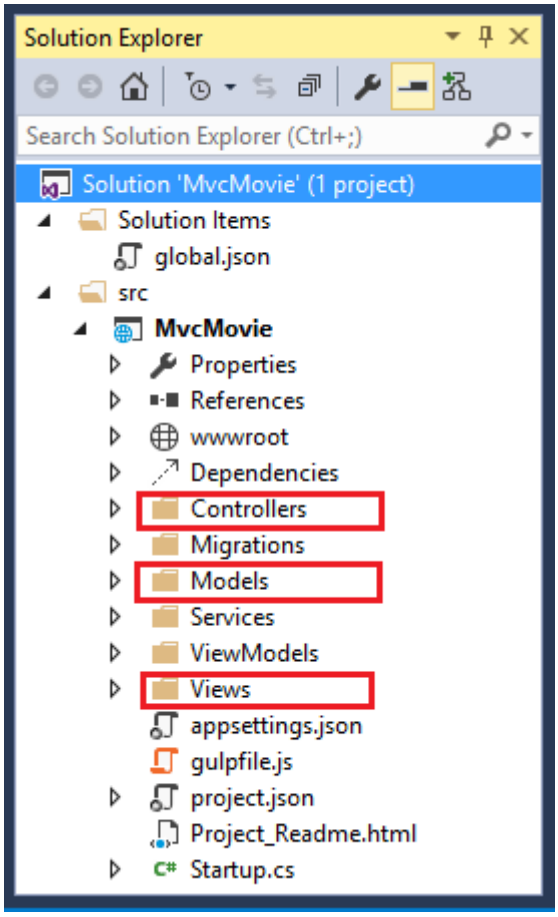
Druhou zmíněnou komponentou MVC je View. View, v překladu „pohled“, slouží k prezentaci Modelu v určité formě. V drtivé většině případů se jedná o určitý druh HTML formátu s prvky jazyka použité platformy. V případě ASP.NET je to Razor syntaxe, umožňující použití C# kódu v HTML kódu.

3.5.2.3 Controller

Třetí komponentou, která pomáhá dotvářet MVC architekturu, je Controller. Ten má na starosti propojení View (pohledu) a Modelu. Controller je první místo, na kterém vždy začne být zpracováván příchozí požadavek. Jedná se o tu komponentu architektury, která se stará o životní cyklus požadavku.

3.5.2.4 Shrnutí MVC

MVC architektura tak pomáhá vytvářet aplikace, mající odděleny různé aspekty funkčnosti, kterými jsou vstupní logika, business logika a logika uživatelského rozhraní [23]. Tyto aspekty však mezi sebou mají určitou vazbu. Architektura dále určuje, kde by měla která komponenta v aplikaci být. Logika uživatelského rozhraní patří do Views, vstupní logika patří do Controlleru a business logika patří do Modelu. Toto oddělení pomáhá udržet komplexnost webových aplikací na rozumné úrovni. Při vývoji lze navíc na každé komponentě pracovat zvlášť, bez nutnosti zásahu do komponenty druhé.



Ukázka kódu 6 - Struktura MVC projektu (zdroj [23])

Na obrázku jsou červeně označeny adresáře pro každou z výše uvedených komponent. Jedná se o standartní strukturu MVC (MVC 6, viz. 3.4).

3.5.3 MVVM

Určitým způsobem doplňkovou architekturou k MVC je architektura MVVM. Na první pohled se může dle jmenné konvence jevit, že se jedná o konkurenční architekturu. MVVM je totiž zkratka pro **Model**, **View**, **ViewModel**. V případě webového vývoje lze však architektury MVC a MVVM velmi efektivně propojit, což je také jedním z dílčích cílů této práce. Reálnou ukázku využití obou architektur lze nalézt v kapitole 5.6.

Model v architektuře MVVM, stejně jako v MVC, vyjadřuje logiku, nebo jinak řečeno business pravidla, aplikace. O Modelu lze mluvit jako o jádru dané webové aplikace, obsahující samozřejmě mimo jiné i soubor tříd objektů, se kterými webová aplikace pracuje.

View je v případě MVVM také velmi podobné (avšak ne stejné) jako u MVC. Je zodpovědné za definování struktury a rozložení zobrazeného uživatelského rozhraní. V ideálním případě se jedná o HTML stránku, která v sobě neobsahuje žádnou logiku aplikace. Jediná aktivní část kódu je takzvaný ***binding***.

Binding reprezentuje nastavenou vazbu mezi View a ViewModelem. **ViewModel** se v architektuře MVVM stává prostředníkem mezi View a Model. Nejdůležitější věcí, která k této architektuře ale patří, je již zmíněný binding, tedy vazba. Propojení ViewModelu a View je zde z toho důvodu, aby se mohly obě komponenty vzájemně ovlivňovat a reagovat na změny svého stavu.

U reálné webové stránky to znamená, že se JavaScriptový datový model aplikace (ViewModel) propojí s DOM na stránce. Lze tak poté nastavit, že při změně hodnoty v některém textovém poli, se toto automaticky promítne také do instance připojeného objektu na pozadí (JavaScript). Tato vazba dokáže fungovat i obráceně, tedy při změně hodnoty v objektu se upraví hodnota ve View. Právě binding pak určuje, které vlastnosti jakého objektu mají být propojené s kterými částmi HTML stránky.

Výhodou této architektury je fakt, že je striktně oddělena UI vrstva od vrstvy datové. Při správné implementaci ViewModel a View lze teoreticky View zaměnit za jakoukoliv jinou implementaci, a aplikace bude fungovat stále stejně, se stejnými omezeními, business pravidly a vůbec celou logikou.

Takováto popsaná funkcionalita však není součástí JavaScriptu jako takového, ale do webové stránky musí být dodána jako JavaScriptový framework, který zajistí aktualizace a reagování na vzájemné změny hodnot u View a ViewModel. Pro použití v této práci byl vybrán framework KnockoutJS, popsaný v následující kapitole. Důvody zvolení právě tohoto frameworku jsou v kapitole 5.6.1.

3.6 KnockoutJS

Knockout (nebo KnockoutJS) je JavaScriptová knihovna, pomáhající vytvářet rozsáhlé, rychle reagující uživatelské rozhraní s minimem JavaScriptového kódu na pozadí. Jejím hlavním účelem je propojení View a ViewModel (z architektury MVVM, viz. 3.5.3) pomocí deklarativních vazeb, které mohou být jednosměrné i obousměrné.

Mezi hlavní funkce patří elegantní sledování závislostí. To lze pochopit jako sledování změn na provázaných objektech a aktualizování všech při změně jednoho. Druhou hlavní funkcionalitou je možnost deklarování vazeb přímo ve zdrojovém kódu uživatelského rozhraní (tedy v HTML). Třetí hlavní kvalitou frameworku KnockoutJS je jednoduché rozšiřování funkcionality, jako je například snadná tvorba vlastního typu vazby.

3.6.1 Fungování

Aby bylo možné využít Knockout v libovolném projektu, nejdříve je nutné uvést referenci na tento framework do hlavičky HTML souboru. Výhodou však je, že tento framework není, jako spousta jiných, závislý na frameworku jQuery, a není tak nutné přidávat další reference.

Jako další krok by mělo následovat vytvoření ViewModelu, tedy objektu, který bude využit jako JavaScriptová část deklarativní vazby. Aby však mohl Knockout aktivovat zmíněné sledování závislostí, musí být proměnné deklarovány funkcí observable(..), která obalí danou proměnnou kódem zajišťujícím fungování všeho výše uvedeného. Ve své podstatě se vnitřně zaregistruje k odběru událostí vyjadřujících změnu této proměnné. ViewModel tak může vypadat například jako na ukázce Ukázka kódu 7.

```
// Definice třídy ViewModelu
function PageEditViewModel() {
    var self = this;

    self.name = ko.observable("");
    self.description = ko.observable("");
}

var viewModel = new PageEditViewModel();

// Aktivace Knockout
ko.applyBindings(viewModel);
```

Ukázka kódu 7 - Aktivace a použití Knockout

Na výše uvedené ukázce je nejdříve vytvořena třída objektu PageEditViewModel, obsahující dvě proměnné – name a description. Za povšimnutí však stojí, že jejich hodnota není rovna pouze prázdnému textovému řetězci, ale funkci ko.observable(…), které jé předána hodnota prázdného textového řetězce jako argument. Následně je tento objekt instancován do proměnné viewModel a nakonec je zavolána funkce **ko.applyBindings(…)**,

která prohledá celý HTML zdrojový kód, nalezne všechny deklarativní vazby a provede propojení View a ViewModelu.

3.6.2 Deklarativní vazba

Samotná deklarativní vazba se vytváří prostřednictvím nastavení **data-bind** atributu v libovolném HTML elementu. Jako argument se předává vždy typ vazby a jméno proměnné k propojení. Typů vazeb je celá řada a lze si vytvářet i vlastní.

```
<input type="text" data-bind="value: name" />
<input type="text" data-bind="value: description" />
```

Ukázka kódu 8 - Knockout a deklarace vazby

Například v ukázce Ukázka kódu 8 jsou proměnné name a description z předchozí ukázky Ukázka kódu 7 napojeny na hodnotu atributu **value** elementu input. Díky tomu poté co zapíšeme jakýkoliv text do jednoho z výše uvedených input elementů, se tyto změny okamžitě projeví i v proměnných objektu viewModel.

Více o možnostech KnockoutJS v dokumentaci [24].

3.7 MongoDB

Čtvrtým nejpopulárnějším databázovým strojem na světě je dokumentová databáze MongoDB [25]. Vzhledem k tomu, že předchozí pozice jsou obsazeny relačními databázemi, lze tak říci, že MongoDB je nepopulárnějších dokumentová databáze na světě (k březnu 2016).

MongoDB je dokumentový typ databáze. Nemá pevně dané schéma, nepodporuje transakce a neumí zpracovat SQL jazyk. Je to naprostý opak jakéhokoliv SQL přístupu, kde je vždy vyžadováno pevné schéma (tabulky), datová normalizace a většinou i podpora transakcí. MongoDB ale neukládá relační, komplexně propojená data. Je uzpůsobena na dokumenty. Dokumenty dělí pouze do kolekcí, což je jakýsi ekvivalent tabulek v relačním přístupu. Zásadní rozdíl je však v tom, že u relačních databází je snaha data co nejvíce rozdělit dle jejich účelu, typu a intenzitě použití. U zmíněné dokumentové databáze je však hlavním účelem ukládat dokumenty, které nemají žádné komplexní vazby na jiné dokumenty.

V této práci je databáze MongoDB použita jako primární datový zdroj, a to z důvodů uvedených v kapitole 5.3.

3.7.1 Způsob ukládání

Databáze MongoDB využívá pro ukládání dokumentů vlastní formát BSON. Definice formátu BSON je dle dokumentace následující:

> *„BSON [bee · sahn], short for Binary JSON, is a binary-encoded serialization of JSON-like documents.“* [36]

Jak z definice vyplývá, jedná se o binární reprezentaci JSON dokumentů. Stejně jako JSON, tak i BSON podporuje vnořování dokumentů a polí v ostatních dokumentech a polích. BSON navíc ještě obsahuje rozšíření, dovolující reprezentaci datových typů, které nejsou součástí JSON. Ovladače pro MongoDB nabízejí možnost konvertování mezi oběma zmíněnými typy, a tak je formát JSON použitelný pro veškeré operace s daty. Formát BSON je využíván pouze pro vnitřní reprezentaci uvnitř databáze.

4 Požadavky na aplikaci

V následujících kapitolách bude popsána implementace jednoduchého CMS, jehož základy budou postaveny na technologiích a návrhových vzorech popsaných v teoretické části této práce. Požadavky na toto CMS jsou sestaveny tak, aby bylo možné využívat maximální potenciál všech představených technologií, návrhových vzorů a nástrojů, a zároveň aby byly možnosti jejího využití co nejuniverzálnější. Současně autor práce představí implementaci vlastního řešení pro návrh datové základny a implementaci vlastní softwarové architektury, která zásadním způsobem zjednodušuje propojení MVC a MVVM.

Souhrnný název pro popisovaný systém je **Scms.** Tato zkratka bude v následujících kapitolách odkazovat právě na autorem implementovaný CMS systém.

4.1 Obecné požadavky na Scms

Scms bude lehký a jednoduchý CMS systém, určený především pro základní prezentační weby. Jeho správu zvládne i běžný uživatel se základními znalostmi ovládání PC. Ovládání systému bude intuitivní a od momentu instalace již nebude potřeba žádného zásahu do zdrojového kódu.

Systém bude obsahovat vlastní administrační část, ve které bude moci správce webu upravovat počet, obsah, názvy a řazení jednotlivých stránek, různých částí obsahu a položek v navigaci. Díky tomu si bude moci sám správce vytvořit a spravovat obsah webu přesně podle svých potřeb, nebo podle potřeb zadavatele. Editace stránek bude obsahovat všeobecně známé ovládání pro editaci a správu textového obsahu (zarovnání text, písmo, styl apod.), nabízejíc mimo jiné také možnost vložení obrázků a tlačítek na libovolné místo do textu.

V rámci editace stránky bude možné libovolně přidávat, odebírat a řadit následující bloky: textové pole, blok HTML kódu, úvodní slides, mapa a určitý druh zvýrazněného textu. Počet textových polí, HTML bloků a polí se zvýrazněným textem bude libovolný s možností řazení. Ostatní bloky mohou být na stránce pouze jednou.

Editace jednotlivých položek v menu dovolí přidávat, odebírat a upravovat již existující položky v menu. Menu samotné bude umožňovat jednu úroveň zanoření, díky čemuž se docílí větší flexibility použití Scms. Při editaci položky menu si správce bude moci

zvolit, zda má být po kliknutí přenesen na manuálně zadanou URL, nebo zda se má po kliknutí dostat na jednu z již vytvořených stránek ve správě Scms.

Mimo výše uvedenou správu obsahu bude možné ještě nastavit barevné schéma celého Scms, díky čemuž se ještě více rozšíří možnosti personalizace. V administrační části tak bude sekce, kde si bude moci správce webu upravit jednotlivé barvy barevného schématu.

4.1.1 Shrnutí obecných požadavků

1. Možnost přidávání, odebírání a editace jednotlivých stránek
    a. Stránky se musí skládat z následujících bloků:
        i. Textové pole
        ii. HTML kód
        iii. Úvodní slides (libovolný počet)
        iv. Mapa
        v. Bloky se zvýrazněným textem
    b. Každý blok či celá stránky musí být deaktivovatelná (bez nutnosti smazání).
    c. Bloky musí mít nastavitelné pořadí při vykreslování.
2. Možnost přidávání, odebírání a editaci jednotlivých položek v menu
    a. Při editaci/přidávání položky menu je vyžadována možnost připojení již existující stránky namísto zadávání přesné URL.
    b. Položky navigace musí být možno seřadit dle potřeby.
3. Nastavení vzhledu
    a. Aplikace musí nabízet nastavení barev schématu.

4.2 Implementační požadavky na Scms

Aby mohl být naplněn cíl práce a veškeré požadavky z kapitoly 4.1, musí výsledná aplikace splňovat několik základních implementačních požadavků. Při implementaci zvolených požadavků a jejich integraci do finálního CMS bude vyžadováno, aby bylo v co nejvyšší možné míře využito nástrojů a technologií, popsaných v teoretické části této práce.

4.2.1 Spustitelnost na operačních systémech

Mezi nejzásadnější požadavky na Scms patří možnost spuštění a hostování v jednom ze tří nejrozšířenějších desktopových operačních systémů. Konkrétně se jedná o Microsoft **Windows, Linux** a **OSX** od firmy Apple. Pokud to není nezbytně nutné, aplikace nesmí pro běh na žádném z uvedených operačních systémů používat neoficiální implementace běhových prostředí.

Ani jakýkoliv typ databáze, kterou bude Scms jistě potřebovat, nesmí být omezena pouze pro jeden operační systém. Zvolená databáze musí být oficiálně podporována pro stejné operační systémy jako samotná webová aplikace.

Důvodem těchto omezení je především zajištění univerzálnosti použití a přenositelnosti aplikace.

4.2.2 Finanční náročnost

Žádný z použitých nástrojů při implementaci nesmí být zpoplatněn. Aplikace Scms musí být vytvořena výhradně za pomoci technologií, knihoven a dalších podpůrných nástrojů, které jsou alespoň ve své základní verzi nezpoplatněné. Výsledný CMS totiž musí být možné nabízet bezplatně, což by ovšem nebylo možné, pokud by samotná implementace využívala některý z placených nástrojů či technologií.

5 Návrh aplikace

Po analýze obecných a implementačních požadavků s přihlédnutím k cílům této práce byl sestaven návrh finální aplikace s popisem a odůvodněním jednotlivých součástí. Následující kapitoly obsahují navrhované řešení.

Při tvorbě navrhovaného řešení autor práce vycházel ze zpracovaných zdrojů v oblasti návrhových vzorů, principů a architektur, popsaných více v teoretické části práce v kapitole 3.4.

5.1 Běhové prostředí

Jako běhové prostředí a tedy i framework byl zvolen .NET Core framework, který je podrobně popsán v teoretické části, konktrétněji poté v kapitolách 3.1 a 3.3. **Použitá verze frameworku v rámci této práce bude 1.0.0-rc2-15546**. Aby byla zajištěna spustitelnost na požadovaných operačních systémech, celá aplikace bude vytvořena tak, aby neobsahovala reference na žádnou z knihoven pro .NET Framework. Díky tomu bude aplikace spustitelná pomocí DNX na již zmíněných operačních systémech.

Zvolení tohoto prostředí nepředcházela žádná analýza variant, neboť podstatou této práce je vytvoření webové aplikace na této platformě.

5.2 Datová základna

V první řadě pro účely webové aplikace Scms autor této práce navrhl vlastní architekturu pro získávání dat z datové základny. Mezi hlavní přednosti této architektury patří možnost snadné výměny implementace přístupu do databáze, výborné možnosti testovatelnosti a především přehlednost pro vývojáře.

Hlavním znakem výše zmíněné architektury je fakt, že pro každou databázovou entitu je připraveno rozhraní s CRUDQ (klasické CRUD, rozšířené o Query) operacemi, které zajišťují veškeré databázové operace s danou entitou. Operace všech entit jsou poté zděděny do jednoho hlavního rozhraní, ze kterého se později přistupuje k implementacím. Vše je tedy reprezentováno v první řadě pomocí rozhraní. To umožňuje nejen připojení libovolné implementace hlavního rozhraní, ale také velmi snadnou testovatelnost, neboť při

testování lze aplikaci podsunout falešnou implementaci, která bude vracet přesně takové výsledky, které jsou nutné pro úspěšný test.

Neméně významným prvkem této architektury je implementace obecného výsledku, který je v určité formě návratovou hodnotu každé CRUDQ metody. Tento výsledek v sobě obsahuje celou řadu informací o právě proběhlém příkazu. Bývá nejčastěji používán v kombinaci s ochranným obalem pro jakékoliv tělo metody, případně blok kódu. Zmíněný ochranný obal je dále nazýván jako **Shield** obecný výsledek je dále nazýván jako **Outcome**.

Detailnější popis a ukázky implementace jsou uvedeny níže, v kapitolách 5.5.1, 5.5.5 a 5.5.6.

5.3 Databáze

Jak je již popsáno v předchozí kapitole, aplikace Scms bude databázi využívat především pro ukládání obsahu jednotlivých stránek. Rychlost a složitost získání celého obsahu jedné stránky bude pro tuto aplikaci zcela zásadní. V předchozí kapitole je psáno, že každá stránka se bude skládat z několika menších bloků, které budou vyplněny různorodým obsahem. Vzhledem k tomuto faktu lze ukládaná data označit za dokumenty, kdy stránka = dokument. Povaha takovéhoto dokumentu také naznačuje, že do budoucna může vyvstat požadavek na přidání celé řady různých dalších typů bloků stránky. Jinými slovy to znamená, že stránka bude vždy sestavena z několika různých pod-objektů. Zvolená databáze tak musí být velmi flexibilní a i při velkém množství různých typů objektů si musí zachovat přehlednost.

Pokud návrh datového modelu obsahuje ve velké míře entity, které lze se svým kontextem snadno vytisknout na stránku papíru, je vhodné alespoň zvážit využití dokumentové databáze.

Výše uvedené skutečnosti vedly autora práce k opuštění klasického modelu, kdy je pro webovou stránku využita některá z relačních databází. Pro takovou strukturu dat, která je v Scms očekávána, bude nejvhodnější použít **dokumentovou** databázi. Jednotlivé stránky budou uloženy jako samostatné dokumenty, což zajistí velmi rychlé čtení a přehlednost. Flexibilita datového modelu bude zajištěna již samotnou podstatou dokumentových databází.

5.3.1 MongoDB

Aby mohla být aplikace Scms multiplatformní, musí být kompletně vytvořena ve frameworku .NET Core. Bohužel ale v době psaní této práce neexistovala žádná prověřená dokumentová databáze, která by nabízela ovladač pro framework .NET Core. Nejlepší možností se stala databáze **MongoDB**, jejíž autoři v době psaní této práce již pracují na ovladači pro .NET Core, viz. [26]. Více o MongoDB v kapitole 3.7.

V rámci této práce tak bude dočasně použit ovladač pro plnohodnotný .NET Framework, který však bude nahrazen ovladačem .NET Core jakmile bude dokončen jeho vývoj. Bohužel, do té doby nebude aplikace Scms možné kompilovat pro .NET Core, což znamená, že **aplikace Scms prozatím nebude multiplatformní**.

Následuje stručný souhrn důvodů pro zvolení MongoDB.

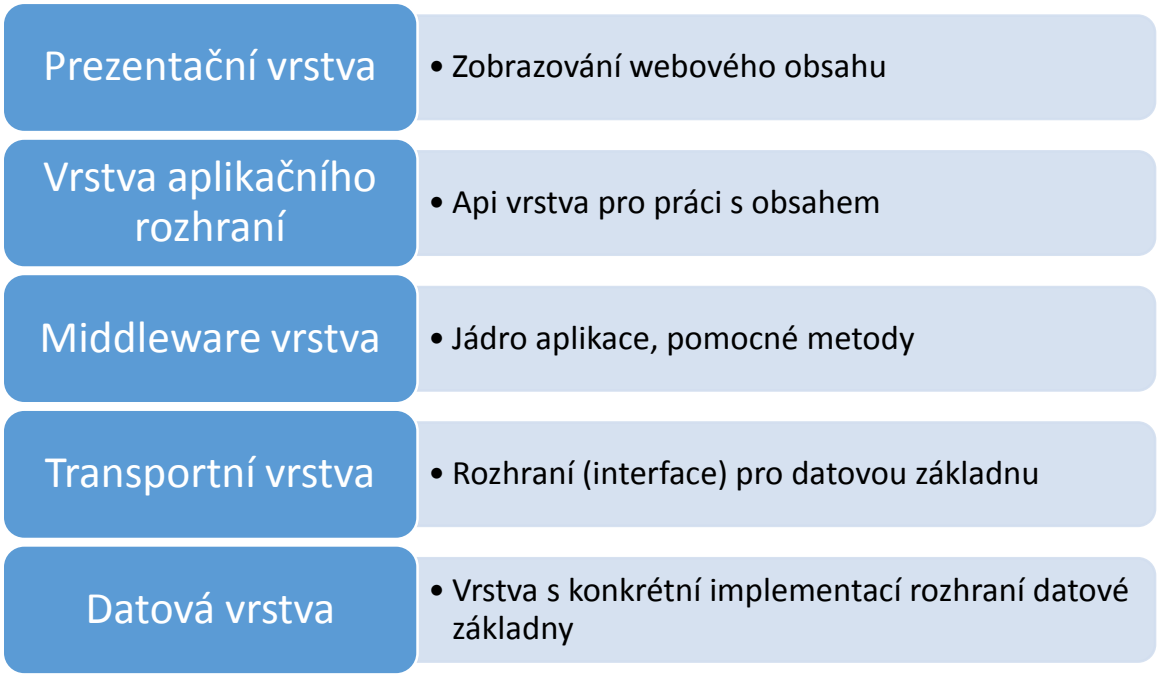
1. Základní verze k volnému využití
2. Multiplatformní podpora
3. Ovladač pro .NET Core je ve vývoji
4. Je velmi flexibilní
5. Dokáže velmi rychle vrátit jednu stránku jako celý dokument bez nutnosti evaluace navázaných entit.
6. Nabízí celou řadu možností pro fulltextové vyhledávání

5.4 Architektura aplikace

Vyvýjená webová aplikace se bude skládat z několika základních prvků, které musí zůstat oddělené, a to z důvodů zachování přehlednosti, udržitelnosti, testovatelnosti, rozšiřitelnosti a nahraditelnosti jednotlivých součástí. Samotná aplikace bude rozdělena na pět základních částí:

1. Prezentační vrstva
2. Vrstva aplikačního rozhraní
3. Middleware vrstva
4. Transportní vrstva
5. Datová vrstva

Následující nákres vyobrazuje seřazení vrstev, kdy nejvýše položená vrstva je směrem ke koncovému uživateli a nejníže položená vrstva vede směrem k software hostovací technologie.



Obrázek 3 - Architektura aplikace Scms (zdroj vlastní)

5.4.1 Prezentační vrstva

Jak již název napovídá, prezentační vrstva zajistí prezentování dat v patřičném formátu a stylu. Jedná se o vrstvu, která zároveň také zajistí interakci s uživatelem, což mimo jiné zahrnuje vykreslování ovládacích prvků, dotazování uživatele a zobrazování dat. Prezentační vrstva zahrnuje obě části Scms – veřejnou i administrační.

Hlavní architekturou pro zobrazování a interakci s daty bude na této vrstvě architektura MVVM na klientské straně, tedy na cílovém stroji. Tento přístup umožní velmi rychlé a přesně reakce UI na změny dat, a zároveň ušetří síťovou náročnost, jelikož na pozadí se bude pracovat pouze s API vrstvou.

5.4.2 Vrstva aplikačního rozhraní

Vrstva aplikačního rozhraní, jinak také API vrstva, reprezentuje soubor koncových bodů, konkrétně HTTP URL adres, které budou sloužit pro manipulaci s daty. Tato vrstva

bude nejvíce využívaná prezentační vrstvou pro úspěšné napojení zobrazovaných modelů (ViewModels) na modely aplikace (Models).

5.4.3 Middleware vrstva

Vrstva, obsahující jádro aplikace, se bude jmenovat Middleware. Tato vrstva bude zajišťovat především plynulý běh aplikace, komunikaci s běhovým prostředím a obsluhu požadavků. Dále bude obsahovat pomocné metody a objekty nutné pro zajištění funkčnosti všech ostatních vrstev.

V neposlední řadě bude obsahovat také kompletní datový model dle class diagramu, vyobrazeného v kapitole 5.5

5.4.4 Transportní vrstva

Propojení datové vrstvy, zajišťující přístup k datům, a vrstvy Middleware bude docíleno pomocí implementace datového rozhraní. Tato vrstva se bude nazývat Transportní, a jako interface bude obsahovat soubor metod, které musí Datová vrstva implementovat pro správnou funkcionalitu.

5.4.5 Datová vrstva

Nejníže položená vrstva v rámci Scms je vrstva datová. Dle názvu lze odvodit její primární účel, který nebude nic jiného než zajištění přístupu do databáze a možnost správy dat v takovéto databázi. Vzhledem k tomu, že se v blízké době očekává výměna databáze, nebo alespoň jejího ovladače (viz. 5.2), znamená to, že **tato vrstva nesmí být přímo závislá na vrstvě Middleware**. Datová vrstva musí být pro Scms snadno zaměnitelná.

5.5 Architektura datové základny (CRUDQ)

Součástí této práce je návrh vlastního návrhového vzoru datového přístupu, využitelného v libovolném projektu. Tento návrhový vzor bude ukázkově využit v aplikaci Scms, která je předmětem této práce. Díky tomu bude možné názorně předvést výhody a možnosti použití autorem navrženého přístupu.

5.5.1 Obecně o architektuře

Mezi první otázky při vývoji nové aplikace patří jistě zvolení vhodné databáze a zároveň vhodného nástroje pro přístup do této databáze. Platforma ASP.NET nabízí Entity Framework, což je nástroj, pomocí kterého lze snadno pracovat s různými druhy SQL relačních databází. V dnešní době rostoucího počtu noSQL[8] databází však mohou nastat situace, kdy by jakýkoliv přístup do databáze vyžadoval napsání značné části kódu. Takovýto kód se pak v různých částech aplikace opakuje a vznikají tak potencionální místa pro určitý druh refaktoringu.

Pro přístup k datovému zdroji tak autor navrhl architekturu, která řeší obecné problémy přístupu k datům. Architektura je připravena pro libovolný datový zdroj (databázi) a lze ji aplikovat v libovolném projektu. Nabízí jednoduché a intuitivní použití, velké možnosti testovatelnosti a možnost jednoduché výměny samotné implementace.

5.5.2 CRUDQ operace

Základní premisou navrhované architektury pro přístup k datovému zdroji je rozdělení operací nad daty do pěti základních kategorií – Create, Read, Update, Delete a Query. Tyto operace vycházejí z již existujícího modelu CRUD a rozšiřují ho ještě o jeden typ operace (Query).

	Popis
Create	Obsahuje metody pro vytváření entit.
Read	Obsahuje metody pro čtení entit dle různých parametrů.
Update	Obsahuje metody pro základní aktualizace entit.
Delete	Obsahuje metody pro smazání entit dle zadaných podmínek.
Query	Obsahuje metody pro složité dotazy, které svým provedením vnitřně využívají více CRUD metod.

Tabulka 2 - Seznam a krátký popis CRUDQ operací

Dle tohoto rozdělení jsou dále v navrhované architektuře rozděleny složky, soubory a rozhraní k implementaci.

[8] NoSQL jsou nekonvenční typy databází, které nejsou postavené na již ověřených relačních databázových přístupech. Jedná se například o objektové, dokumentové, grafové a další druhy databází.

5.5.2.1 IDataStore

Přístup k datovému zdroji musí být jednoduchý a intuitivní. Z toho důvodu je v architektuře počítáno s objektem, který v sobě zapouzdřuje přístupy ke všem CRUDQ operacím. Tento objekt se nazývá IDataStore. Jedná se o primární zdroj všech implementací. Projekt, který chce využít architekturu CRUDQ pro svou datovou základnu, musí mít tento interface připravený.

Samotný interface IDataStore není nijak zvlášť složitý. Obsahuje totiž jen pět vlastností, každá pro jednu z CRUDQ operací.

```
public interface IDataStore
{
    IQuery Query { get; set; }
    IRead Read { get; set; }
    ICreate Create { get; set; }
    IDelete Delete { get; set; }
    IUpdate Update { get; set; }
}
```

Ukázka kódu 9 - IDataStore interface

5.5.2.2 Operační a detailní operační interface

Pro samotné operace je vždy nutné implementovat samostatné interface, reprezentující každou z CRUDQ operací. Tyto interface se nazývají **operační interface** a skládají se z ICreate, IRead, IUpdate, IDelete a IQuery interface. Jejich obsahem je pouze soubor vlastností, reprezentujících **detailní operační interface.** Detailní operační interface je pojmenován jako I{Třída}{Operace}, kde {Třída} reprezentuje název třídy, kterou má tento koncový bod vracet, a {Operace} reprezentuje, stejně jako u operačních interface, jméno operace. Příkladný název detailního operačního interface tak může být ICarRead. Dle názvu je tak hned zřejmé, že se jedná o interface, které implementuje metody pro čtení objektů typu Car.

Každé samostatné detailní operační interface již musí obsahovat vlastnosti, které již popisují danou operaci pro určitou entitu datového modelu aplikace. Za předpokladu, že vyvíjená aplikace obsahuje již zmíněnou entitu Car a mezi požadavky patří možnost čtení hodnoty Car podle různých parametrů z databáze, musí interface IRead obsahovat detailní

operační interface typu ICarRead. Jméno této proměnné **musí** být stejné jako jméno dotazovaného typu (resp. třídy). IRead interface tak může vypadat následovně:

```
public interface IRead
{
    /// <summary>
    /// Operace pro čtení objektů Car.
    /// </summary>
    ICarRead Car { get; set; }
}
```

Ukázka kódu 10 - Možná struktura operačního interface IRead

Detailní operační interface ICarRead poté musí obsahovat metody pro načítání jedné či více instancí objektu Car dle různých parametrů. Může proto vypadat například jako následující ukázka kódu, která obsahuje metody ById(..) a Many(), které vrací jeden objekt, respektive kolekci objektů typu Car.

```
public interface ICarRead
{
    /// <summary>
    /// Vrátí instanci třídy Car dle předaného Id.
    /// </summary>
    /// <param name="id">Identifikace(Id) žádaného objektu Car.</param>
    /// <returns></returns>
    Car ById(string id);

    /// <summary>
    /// Vrátí všechny existující objekty typu Car.
    /// </summary>
    /// <returns></returns>
    IEnumerable<Car> Many();
}
```

Ukázka kódu 11 - Možnost struktury detailního operačního interface ICarRead

Operační interface takto funguje pro všechny operace. To znamená, že postup pro IUpdate, ICreate, IDelete a IQuery bude analogický s postupem uvedeným pro interface IRead. To samé samozřejmě platí i pro detailní operační interface. Následuje tabulka se jmennou konvencí v různých detailních operačních interface. Za proměnnou „{třída}“ se dosazuje jméno třídy, pro kterou je právě daná operace implementována.

Název metody	Návratový typ	Argumenty metody		Popis
		Typ	Jméno	

I{třída}Create (např. ICarCreate)				
New	Outcome<{třída}>	{třída}	data	Uloží předávanou třídu (argument data) do databáze a vrátí předávanou entitu, například s automaticky generovaným a nastaveným Id.
I{třída}Read (např. ICarRead)				
ById	Outcome<{třída}>	string	id	Vrátí danou entitu dle Id.
ManyByName	Outcome<List<{třída}>>	string	name	Vrátí kolekci entit dle parametru name.
AllActive	Outcome<List<{třída}>>			Vrátí kolekci entit, které mají určité nastavení, ze kterého vyplívá, že jsou aktivní.
I{třída}Update (např. ICarUpdate)				
Set	Outcome	string	id	Provede update typu Set, což znamená, že se upraví pouze hodnoty, které nejsou null.
		{třída}	dataSet	
Replace	Outcome	string	id	Provede aktualizaci, která entitu kompletně nahradí.
		{třída}	dataReplace	
I{třída}Delete (např. ICarDelete)				
ById	Outcome	string	id	Odstraní danou entitu z databáze dle Id.
I{třída}Query (např. ICarQuery)				
MergeAllAndUpdate	Outcome<List<{třída}>>	string	data	Provede složitější dotaz, který zahrnuje update a vrácení upravených entit například s některými hodnotami evaluovanými.

Tabulka 3 - Obsah operačních rozhraní

Jména metod v tabulce Tabulka 3 by měla být dodržena, aby bylo docíleno chtěné jednoduchosti a intuitivnosti použití. Jméno a účel metody pro I{třída}Query je smyšlené a

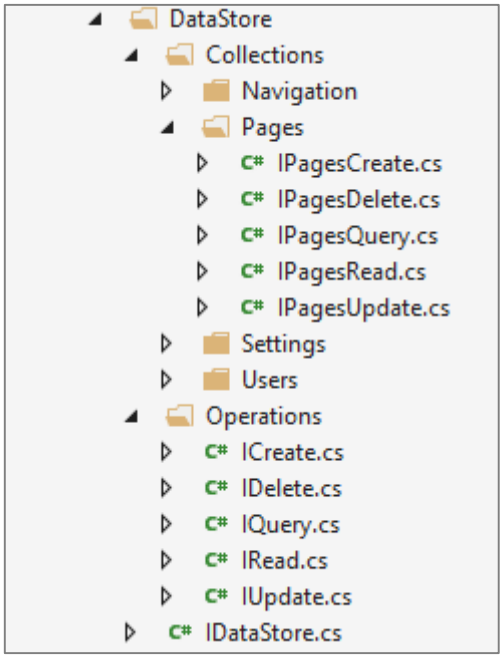
slouží pouze pro představu a upřesnění druhu a povahy metod, které jsou pro tento typ operací vhodné. Ostatní jména metod reprezentují reálné možné požadavky.

Je velmi důležité podotknout, že pro správnou funkci **není nezbytně nutné dopředně vytvářet všechna teoreticky použitelná interface**. Jedna ze silných vlastností CRUDQ architektury je totiž fakt, že není nutné ztrácet vývojový čas metodami, které velmi pravděpodobně ani nebudou nikdy využity.

5.5.3 Struktura souborů CRUDQ architektury

Velmi častý problém při návrhu architektury je také způsob ukládání zdrojových souborů a struktura jejich uložení. Návrh architektury CRUDQ ale počítá i s tímto aspektem. Struktura je navržena opět tak, aby byla logicky uspořádána, a vyhledávání a správa souborů v ní obsažených byly intuitivní.

V kořenovém adresáři by měly být soubor s interface IDataStore a adresáře Operations a Collections. Adresář Operations obsahuje soubory s definicí *operačních interface*, které jsou popsány v kapitole 5.5.2.2. Bude se tak jednat o soubory s názvy ICreate, IRead, IUpdate, IDelete a IQuery. Co má být jejich obsahem je rovněž v této kapitole. Struktura adresáře Collections obsahuje podadresáře dle jmen entit, se kterými má umět IDataStore pracovat. V případě, že chtěná databáze musí obsahovat metody pro práci s objekty Car, User a Rental, budou obsahem Collections adresáře tři podadresáře pojmenované stejně jako uvedené objekty. Obsahem těchto podadresářů jsou vždy soubory s definicí *detailního operačního interface*. Obrázek 4 zobrazuje implementovanou struktury CRUDQ architektury do webové aplikace Scms, jejíž vývoj je předmětem této práce.



Obrázek 4 – Souborová struktura CRUDQ

5.5.4 Způsob práce s IDataStore

Jak již bylo řečeno v úvodu této kapitoly, hlavní výhodou CRUDQ přístupu je jednoduchá obsluha. Následuje krátká ukázka kódu, demonstrující jednoduchý přístup k datům.

```
// Načtení objektu Car dle id
Outcome<Car> carReadOutcome = dataStore.Read.Car.ById("123");

// Aktualizace hodnoty Color objektu Car
Outcome carUpdateResult = dataStore.Update.Car.Set("123", new Car() {Color =
"modra"});
```

Ukázka kódu 12 - Předvedení použití CRUDQ architektury

Na této ukázce je velmi dobře vidět pohodlný přístup k metodám pro správu objektu Car. Naprosto analogický přístup je při práci s ostatními operačními a detailními operačními interface.

5.5.5 Obecný návratový typ Outcome

Návratový typ, který popsaný IDataStore používá, je Outcome. Outcome je autorem navržený obecný návratový typ (třída), který v sobě zahrnuje základní funkcionalitu pro logování chyb, informace o proběhlé události a celou řadu dalších pomocných vlastností. Kromě své obecné verze má i takzvanou generickou verzi Outcome<T>, která navíc dokáže udržovat i instanci návratového objektu. Samotná implementace tohoto objektu je k nalezení ve zdrojových souborech, které jsou přílohou této práce.

Návratový typ Outcome se skládá z následujících vlastností:

Typ	Jméno	Popis
bool	Conclusion	Určuje, zda daný kus kódu proběhl dle očekávání nebo ne.
string	ErrorMessage	V případě, že má Conclusion hodnotu false, obsahuje tato proměnná chybovou hlášku, která způsobila neúspěšné proběhnutí bloku kódu.
string	Guid	Jedinečný identifikátor instance objektu Outcome. Pod touto hodnotu je také chyba zalogována.
Exception	Exception	V případě nastalé výjimky je v této proměnné obsažena.
T	Output	(Dostupné jen u Outcome<T>) Pokud je Conclusion rovno hodnotě true, pak tato vlastnost obsahuje instanci požadovaného návratového typu.

Ukázka kódu 13 - Vlastnosti třídy Outcome

5.5.6 Ochranná služba Shield

Žádná z metod implementace datového interface IDataStore by neměla selhat takovým způsobem, aby nastala nějaká neočekávaná výjimka a aplikace kvůli tomu zaznamenala neočekávaný pád. Z toho důvodu autor připravil pomocnou službu Shield. Její hlavní úkoly jsou následující:

- Spouštění vybraného kódu v try/catch bloku a logování případných chyb
- V případě chyby dokázat uložit rozšířené informace, jako například vstupní argumenty
- Generování instancí objektu Outcome a Outcome<T>

Implementace této metody je rovněž k nalezení ve zdrojových souborech přiložených k této práci.

5.5.6.1 Způsob použití

Instance objektu Shield má jednu hlavní metodu, a tou je Execute, která jako svůj první argument přebírá delegát na libovolnou funkci. Tato funkce je následně spuštěna v try/catch bloku aby bylo zabráněno pádu. Jako návratový typ z metody Execute je pouze typ Outcome (nebo jeho generická alternativa). Jak již bylo řečeno v předchozí podkapitole, objekt Outcome obsahuje mimo instance návratového objektu také možnost navrátit informaci o proběhlé chybě. Tato funkcionalita se hodí právě v kombinaci s použitím metody Execute.

Metoda, která obaluje své tělo metodou Execute objektu Shield a jejíž návratový typ je Outcome, je zabezpečená proti pádu. V případě, že nastane neočekávaná chyba, je odchycena try/catch blokem a automaticky vrácena jako chybová hláška, která je součástí objektu Outcome. Příklady použití lze nalézt v implementaci datového zdroje IDataStore v ukázkovém projektu, který je přílohou této práce.

5.6 Propojení MVC a MVVM

Jak již bylo uvedeno v kapitole 3.5.3, architektury MVC a MVVM jsou v případě webových aplikací vzájemně doplnitelné. Platformou pro MVC nebude samozřejmě nic jiného než ASP.NET MVC 6, nabízející nepřeberné množství vlastností a pomocných

metod. Defaultní struktura projektu je již dána využitou architekturou Model-View-Controller, což znamená, že projekt bude obsahovat stejnojmenné složky, reprezentující komponenty MVC.

Architektura MVC ale neřeší dnes stále rozšířenější přístup, kdy je část logiky webové aplikace zpracována již na klientské části, prostřednictvím JavaScriptu. Toto má nespornou výhodu v rychlosti zobrazování zpracovaných informací a má zásadní vliv na uživatelský zážitek z aplikace. Autor práce uznal za vhodné využít jeden z již existujících frameworků pro implementaci administrační části webu Smcs.

Architektura MVVM tak bude využita pro všechny formuláře, které budou obsahem administrační části aplikace Scms. Tento přístup je a bude zobecnitelný pro návrh kterékoliv webové aplikace. Základním stavebním kamenem tohoto propojení bude **implementace vlastního generátoru JavaScriptového datového modelu**, který bude sloužit jako ViewModel (viz. 3.5.3).

5.6.1 Výběr MVVM frameworku

Frameworků pro implementaci MVVM architektury do klientské části existuje celá řada. Vzhledem k tomu, že tato práce má sloužit i jako demosnstrace využití reálně využívaných technologií, byl výběr vhodného MVVM frameworku zúžen pouze na ty, které mají podíl na trhu větší nebo roven 0,1% (podíl webových stránek, využívající daný framework). Tomuto hlavnímu požadavku vyhovovaly pouze dva frameworky.

	Zastoupení na trhu - březen 2016
Angular JS	0,20%
Knockout	0,10%

Tabulka 4 - Zastoupení MVVM JavaScriptových frameworků na trhu [27]

Mimo výše uvedené frameworky bylo uvažováno také o frameworku Backbone, nicméně autor zjistil, že tento Framework by svou strukturou nebyl vhodným kandidátem na demonstraci autorem navrhovaného řešení.

Výběr vhodného frameworku bude proveden prostřednictvím vícekriteriální analýzy variant, konkrétně pomocí **bodovací metody s váhami**.

5.6.1.1 Požadavky na MVVM Framework

Aby mohl autor dostatečně jednoduše předvést navrhnuté propojení MVC a MVVM, musí zvolený JavaScriptový framework splňovat několik základních podmínek a požadavků. Tyto podmínky jsou ve formě výběrových kritérií následující.

1. Složitost použití
2. Velikost knihovny (čím menší tím lepší)
3. Množství funkcí nad rámec využití (čím více tím hůře)
4. Licence

Hodnoty **vah kritérií** jsou nastaveny v rozsahu **1 až 10,** kdy 1 znamená nejméně důležité a 10 nejvíce důležité.

Hodnoty **bodového ohodnocení** se budou pohybovat v rozmezí **1 až 10**, kdy více znamená lepší ohodnocení.

5.6.1.2 Hodnocení variant

Složitost použití – Váha 10

Složitost použití se skládala z dílčích hodnocených vlastností. Mezi základními byl rozsah a komplexnost dokumentace. Dále byla porovnávána přehlednost návodu pro první použití a zároveň také počet vláken v různých diskuzních stránkách, neboť čím více se o daném frameworku píše, tím je větší šance na rychlé odstranění případné chyby použití.

	BODY	ODŮVODNĚNÍ
ANGULAR	4	Byť je dokumentace strukturována, je stále velmi rozsáhlá a obsahuje obrovské množství informací, které by musel autor před použitím zpracovat.
KNOCKOUT	8	Dokumentace je v přijatelné velikosti, jednoduchá a strukturovaná, avšak o tomto frameworku existuje v diskusních fórech méně vláken.

Tabulka 5 – VAV - Složitost použití

Velikost knihovny – Váha 3

Tímto kritériem byla hodnocena velikost vybraného frameworku v jeho minimálním využití. To znamená velikost JavaScriptového souboru při využití minimálního počtu nutných funkcí.

	BODY	ODŮVODNĚNÍ
ANGULAR	6	AngularJS ve verzi 1.4.5 má velikost 143kb.
KNOCKOUT	10	KnockoutJS ve verzi 3.4.0 má velikost 58,8kb

Tabulka 6 - VAV - Velikost knihovny

Množství funkcí nad rámec využití – Váha 7

Vzhledem k povaze využití MVVM frameworku v této práci, není nutné, aby zvolený framework obsahoval velké množství zbytečné funkcionality, která nebude v projektu využita. Toto by ve výsledku mohlo být jen matoucí a mohlo by to také mít vliv na rychlost implementace a pochopení popisovaných problémů.

	BODY	ODŮVODNĚNÍ
ANGULAR	7	AngularJS obsahuje obrovské možnosti pro vytvářen komplexních clinet-side webových aplikací.
KNOCKOUT	9	KnockoutJS obsahuje základní funkcionalitu pro data-binding a je rozšířen pouze o malý počet extra funkcionality.

Tabulka 7 - VAV - Množství funkcí nad rámec využití

Licence – Váha 5

Mezi důležitá kritéria patří mimo již zmíněné také licence. Naprosto volná licence použití pro veškeré komerční i nekomerční účely je ohodnocena 10 body, jakékoliv odchylky od uvedeného jsou poté penalizovány různým počtem bodů. Nejméně bodů má takový framework, který není volně dostupný.

	BODY	ODŮVODNĚNÍ
ANGULAR	10	MIT licence.
KNOCKOUT	10	MIT licence.

Tabulka 8 - VAV - Licence

5.6.1.3 Výsledky výběru

Výsledné hodnoty byly zadány do tabulky a následně bylo spočítáno vážené bodové ohodnocení. Tyto hodnoty pak byly pro každou variantu sečteny a vybrána byla taková, která má nejvíce bodů.

	Body		Váha	Vážené body	
Kritérium	Knockout	Angular		Knockout	Angular
Složitost použití	8	4	10	80	40
Velikost knihovny	10	6	3	30	18
Množství funkcí nad rámec použití	9	7	7	63	49
Licence	10	10	5	50	50
Součet bodů				**223**	**157**

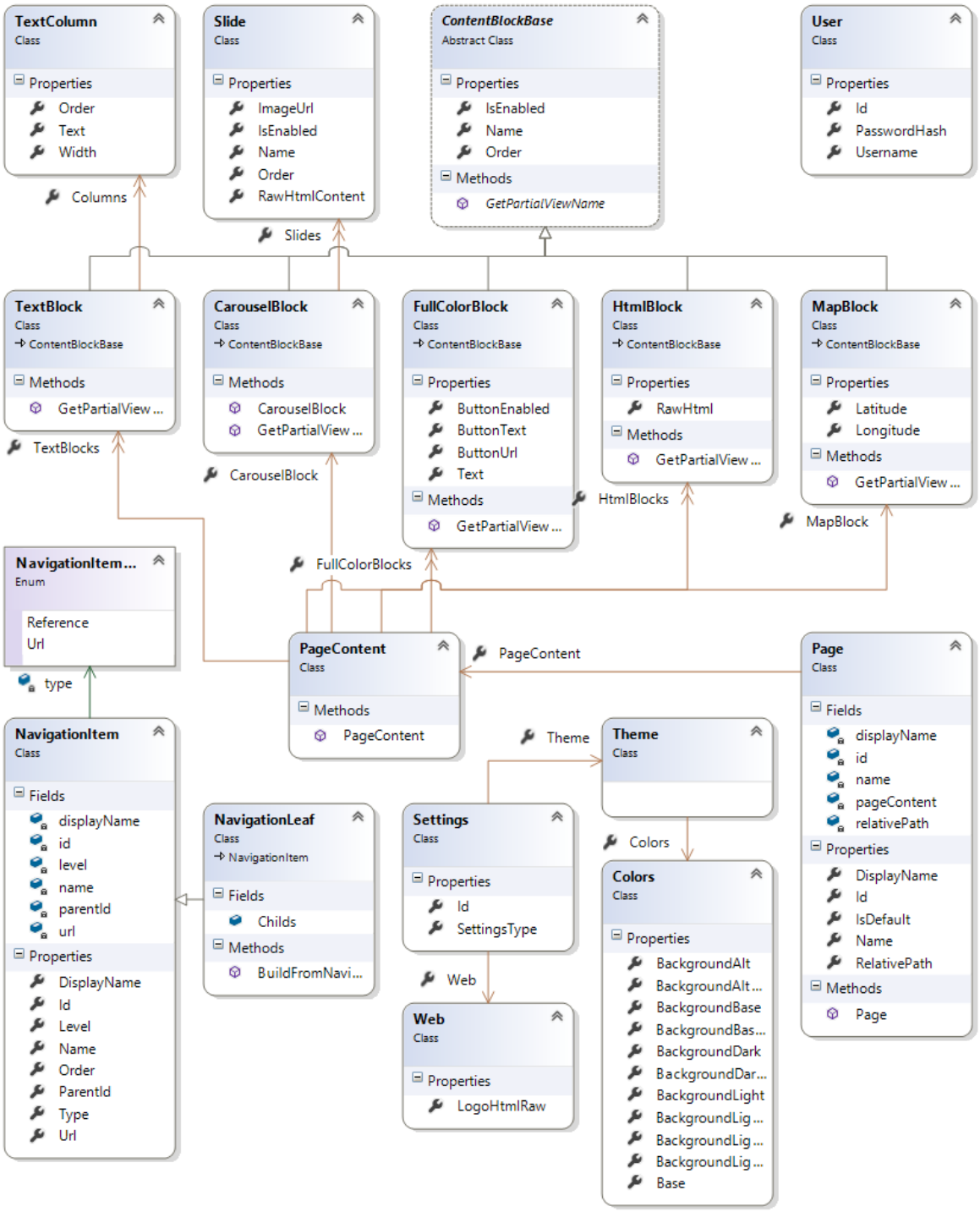
Tabulka 9 - VAV - Vyhodnocení

Vzhledem k tomu, že framework Knockout získal více bodů, jedná se o kompromisní variantu, a v projektu bude pro demonstraci propojení MVVM a MVC využit právě tento framework.

5.7 Class diagram

Pro návrh struktury objektů bylo využito standartního nástroje jazyka UML, konkrétně Class diagramu. Class diagram slouží pro vyobrazení objektových tříd v projektu, včetně jejich závislostí. Následující class diagram obsahuje třídy potřebné pro jádro aplikace Scms. Diagram tříd datové základny bude následovat níže.

5.7.1 Class diagram jádra Scms



Obrázek 5 - Class diagram jádra Scms

Jádro Scms je sestaveno ze tří základních jednotek. Těmi jsou navigace, stránky a obsahové bloky (content block). Asociační a generalizační vazby jsou vyobrazeny na grafu v obrázku Obrázek 5. Základní myšlenka je taková, že se každá jednotlivá stránka (objekt

typu Page) bude skládat z několika bloků, které mohou být různého typu. Jelikož však do jedné kolekce nelze vložit více různých typů objektů, obsahuje třída Page asociační vazbu na pomocnou třídu PageContent.

Pomocná třída PageContent pak udržuje jednoduché a mnohonásobné vazby na různé typy obsahových bloků. Obsahový blok je jakýkoliv objekt, který dědí z abstraktní třídy ContentBlockBase. Abstraktní třída ContentBlockBase udržuje několik základních vlastností, které musí mít každý obsahový blok. Speciální případy ContentBlockBase jsou popsány v kapitole 5.7.1.1.

Aby bylo možné strukturovat stránky do navigace (menu webové stránky), je v návrhu počítáno také s objekty NavigationItem a NavigationLeaf, které je možné jednoduchým algoritmem uspořádat do navigační struktury typu strom. Vazba je udržována pomocí vlastnosti ParentId, která se odkazuje na Id vždy své nadřazené položky v navigaci. Speciálním případem nadřazené položky je položka s hodnotou „null“, tedy prázdná. Ta vyjadřuje teoretický kořen stromové struktury. Ve skutečnosti však navigační položky s prázdnou hodnotou vlastnosti ParentId vyjadřují nultou úroveň zanoření. Jinými slovy se jedná o hlavní rozdělení navigace.

Do této chvíle zůstal nepopsán pouze objekt User. Jedná se o velmi jednoduchou implementace uživatele webové stránky. Defaultní nastavení Scms počítá pouze s jedním uživatelem – „admin“.

5.7.1.1 Speciální případy ContentBlockBase

TextBlock

Hlavní obsahový prvek, ze kterého lze skládat textový obsah libovolné stránky, je TextBlock. Může obsahovat až 12 sloupců textu. Obsah vlastnosti Text je HTML kód s nastaveným formátováním textu. Následující ukázka kódu vyobrazuje ukázkovou instanci objektu TextBlock ve formátu JSON.

```
{
    "order" : 10,
    "isEnabled" : true,
    "name" : null,
    "columns" : [
        {
            "text" : "<h3>Kontakty</h3>\n<p>Libovolný text sloupce 1</p>",
            "order" : 0,
            "width" : 6
        },
        {
            "text" : "<h3>Kontakty</h3>\n<p>Libovolný text sloupce 2</p>",
            "order" : 1,
            "width" : 6
        }
    ]
}
```

Ukázka kódu 14 - Ukázková instance objektu TextBlock ve formátu JSON

CarouselBlock

Každá stránka může obsahovat maximálně jeden CarouselBlok, což je blok s libovolným počtem slides, které slouží pro základní prezentaci novinek a hlavních vlastností. Bloky typu Carousel slouží pro úplně první kontakt koncového uživatele s webovou prezentací. Bývají tak nejčastěji na úvodních stránkách. V Scms lze CarouselBlock nastavit pouze jednou na každé stránce.

FullColorBlock

Pro možnost přidání zvýrazněného bloku textu, například s nějakým typem akce, je v Scms modelu počítáno s takzvaným FullColorBlockem. Ten obsahuje pouze jeden řádek zvýrazněného textu a tlačítko na přesun na libovolnou adresu. Model FullColorBlock tak neobsahuje žádnou vlastnost, která by měla obsahovat HTML kód.

HtmlBlock

Aby byla zajištěna rozšiřitelnost, Scms model obsahuje definici třídy HtmlBlock, jejíž vlastnost RawHtml může obsahovat libovolný HTML kód. Díky tomu je možné přidat do stránky například blok s již hotovou HTML strukturou.

MapBlock

Jako poslední implementovaný speciální typ ContentBlock je MapBlock. Jeho nastavení je velmi jednoduché a skládá se pouze z nastavení defaultních vlastností

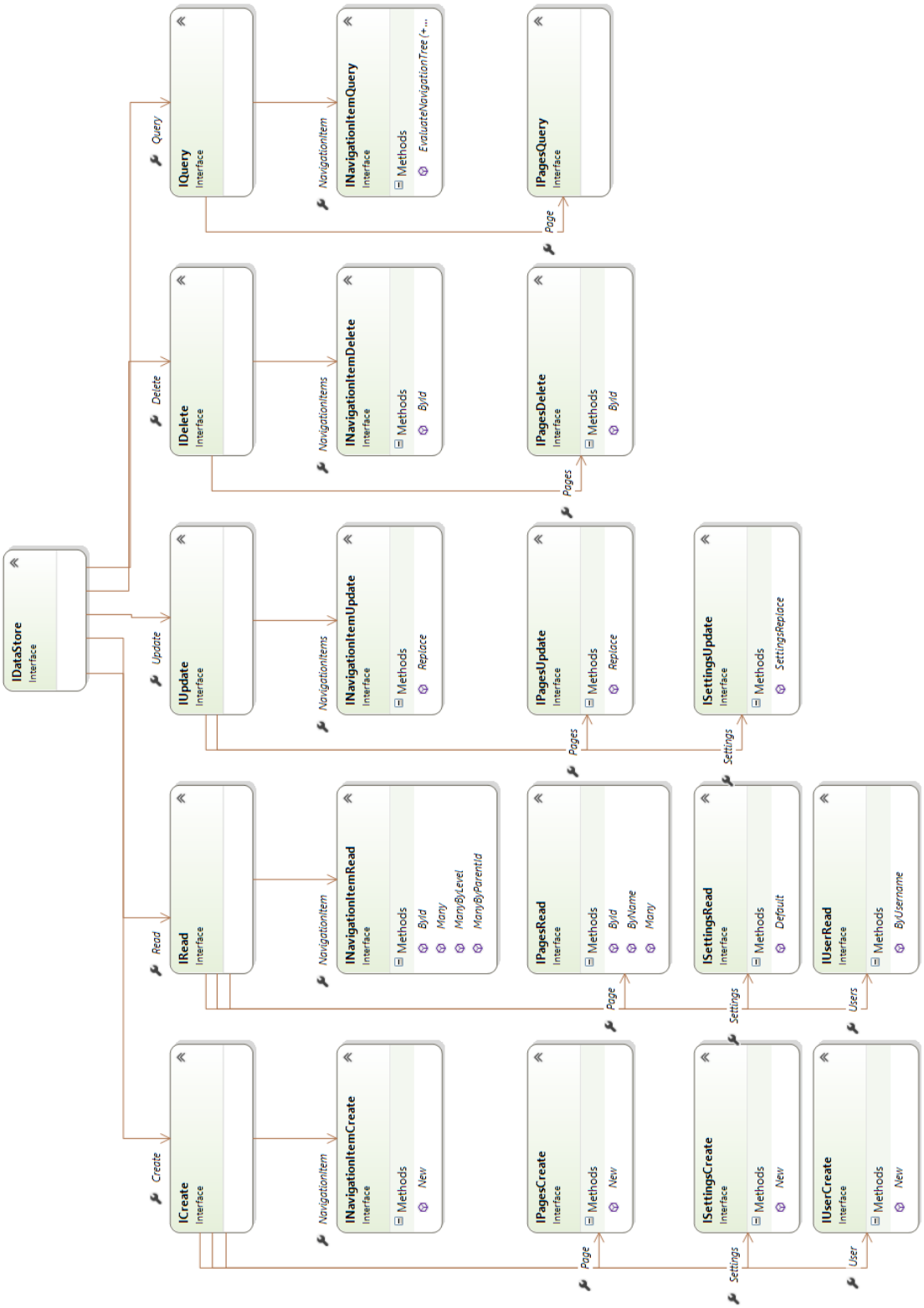
ContentBlockBase a GPS souřadnic. Po aktivaci MapBlocku, který může být stejně jako CarouselBlock pouze jeden, se přes celou šířku stránky (v nákresu Obrázek 7 až za hranice modrých vodících čar) vykreslí interaktivní mapa s jedním bodem, který ukazuje na nastavené souřadnice.

5.7.2 Class diagram datové základny

Jak již bylo uvedeno v kapitole 5.4, tak Datová vrstva a Middleware vrstva musí být odděleny Transportní vrstvou, která je v Scms realizována vytvořením interface a současným využitím architektury CRUDQ z kapitoly 5.5. Interface v jazyce C# vyjadřuje závazek implementovaných metod a vlastností. Middleware vrstva tak bude pracovat pouze s dále popisovaným interface. Samotná implementace tohoto interface bude Datová vrstva. Tato implementace je aplikaci Scms předána při startu.

Toto řešení má zásadní výhodu v tom, že **aplikace není závislá na přímé implementaci přístupu do databáze**. Aplikace používá pouze závazný interface, jehož implementace je aplikaci předána při startu. Implementací tohoto interface může být několik. V rámci této práce a aplikace Scms existuje jedna implementace, která využívá databáze MongoDB. Následující diagram vyjadřuje implementaci vlastní architektury datové základny pomocí interface.

Tento popisovaný interface se jmenuje IDataStore a jedná se o aplikování autorem navržené CRUDQ architektury. Následující diagram obsahuje interface strukturu, která byla vytvořena dle požadavků CRUDQ.



Obrázek 6 - Class diagram CRUDQ pro Scms

5.8 Uživatelské rozhraní

5.8.1 Uživatelské rozhraní klientské části

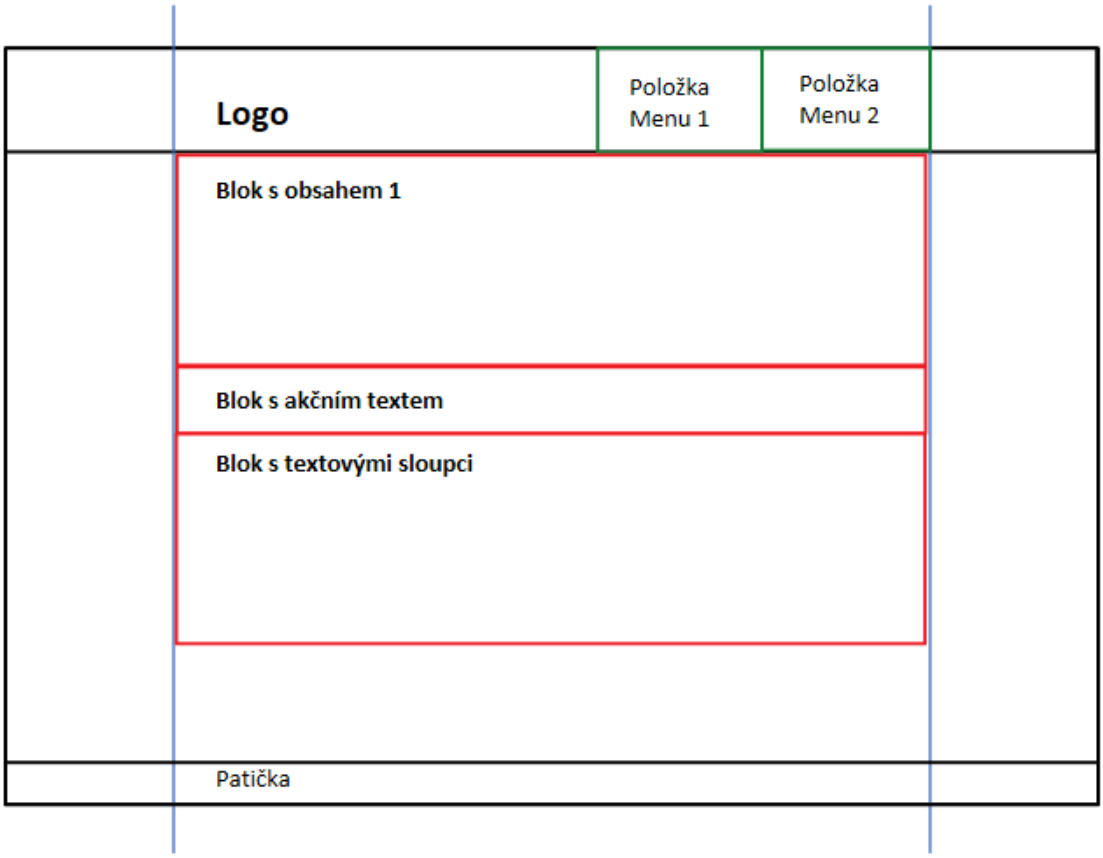
Aby mohl být Scms reálně použitelný, musí být navržen v hezkém a funkčním uživatelském rozhraní. Toho bude docíleno použitím předpřipraveného webového designu, který bude do aplikace Scms zapracován tak, aby s ním bylo možné i nadále pracovat a aby si i běžný uživatel mohl vzhled více či méně přizpůsobit svým potřebám.

Zmíněný předpřipravený design byl vytvořen autorem ještě před vytvářením této práce. Jeho základ je postaven na stylech Bootstrap[9]. Design je optimalizován pro následující webové prohlížeče.

- Internet Explorer od verze 8
- Google Chrome
- Mozzila Firefox
- Opera
- Safari

Základ designu je složen z širokých bloků, které se v různých kombinacích skládají na sebe. V designu existuje celá řada různých bloků, ale pro účely první verze Scms bude využito jen několik z nich. Některé bloky navíc lze dělit až na 12 sloupců. Omezení 12 sloupců plyne z použití již zmíněného Bootstrapu. Na následujícím nákresu je vyobrazeno základní rozdělení běžné stránky. Modré vodící čáry označují okraje obsahové části stránky.

[9] Bootstrap je JavaScriptová a CSS knihovna stylů, sloužící pro vytváření příjemně vypadajících uživatelských rozhraní. Více na http://getbootstrap.com/.

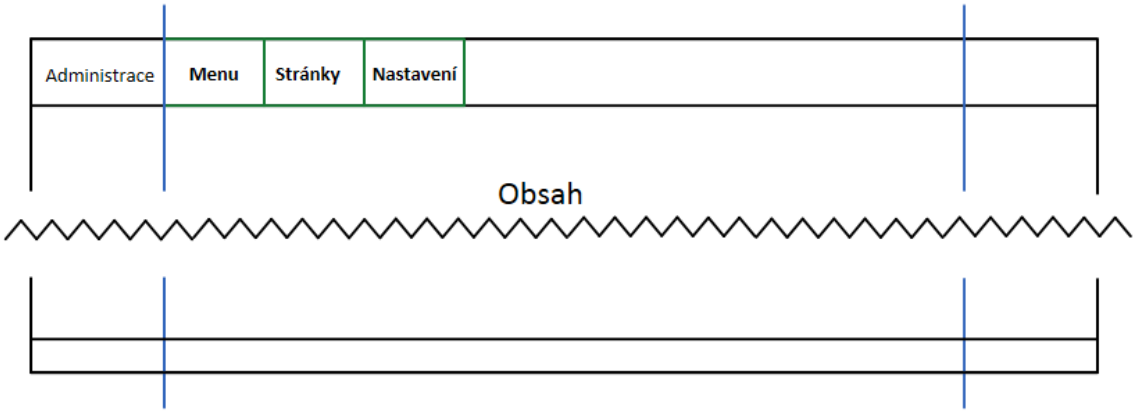


Obrázek 7 - Náčrtek klientské části uživatelského rozhraní

Výše uvedený obrázek znázorňuje obecné rozložení webové stránky, která takto vždy obsahuje logo, několik položek navigace, patičku, a především potom libovolný počet různých typů obsahových bloků. Některé typy obsahových bloků mohou mít pozadí roztažené po celé šířce stránky, avšak obsah bude vždy držen mezi modrými vodícími čárami.

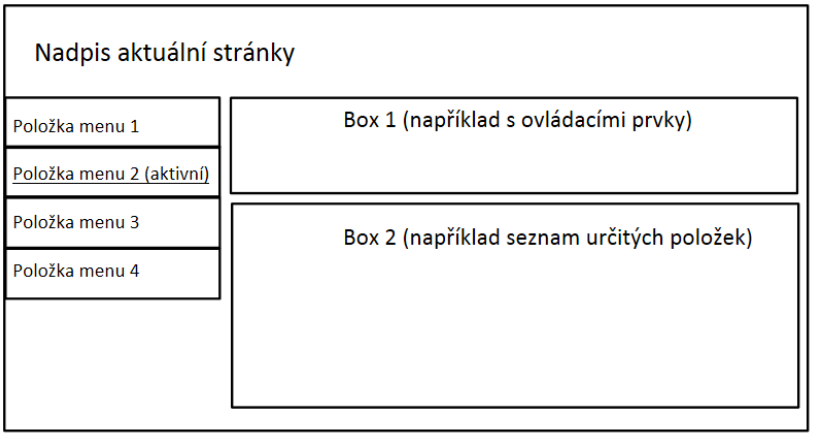
5.8.2 Uživatelské rozhraní administrační části

Uživatelské rozhraní pro administrační část Scms bude vytvořeno pomocí čistého stylu Bootstrap. Administrační rozhraní tak bude sestaveno pouze z defaultních ovládacích prvků Bootstrap, jelikož zde půjde především o funkčnost. I přes to ale použitím již zmíněných stylů bude dosaženo příjemně působícího UI. Následující obrázek vyobrazuje základní zobrazení administrační části Scms.



Obrázek 8 - Nákres rozložení administrační části Scms

V prostoru, který je na obrázku Obrázek 8 označen jako Obsah, a je ohraničen modrými vodícími čarami, bude vykreslen obsah dle aktuální položky menu. Na následujícím nákresu budou vyobrazeny základní UI prvky obsahové části administrační stránky, tedy hrubý náčrtek stránek Menu, Stránky a Nastavení. Všechny kategorie tak budou vycházet vždy z jednoho základního modelu (náčrtku), což pomůže udržet ucelený design v administrační části Scms, a přispěje tak k uživatelské přívětivosti.



Obrázek 9 - Obsahová část stránky v administračním rozhraní Scms

Některé stránky mohou výjimečně využít nákres z obrázku Obrázek 9 v mírně poupravené formě bez menu po levé straně. Tato varianta bude existovat pro stránky, na kterých by levé menu postrádalo smysl, nebo pro stránky, kde by v menu byla pouze jedna položka menu. V takovémto případě může být tedy menu vynecháno a obsahové boxy budou roztaženy tak, aby vyplnily chybějící místo po menu.

6 Implementace

Aplikace Scmc je implementována v jazyce C# na platformě ASP.NET Core 1.0 ve verzi běhového prostředí 1.0.0-rc2-15546. Pro vývoj bylo využito vývojové studio Visual Studio 2015 s pomocným nástrojem ReSharper 9 (studentská verze). Zdrojové soubory aplikace jsou publikovány jako jeden balíček obsahující předkompilované soubory. Tento balíček, společně se zdrojovými kódy, je k této práci přiložen jako příloha.

6.1 Rozdělení projektů

Zdrojové soubory jsou rozděleny do několika dílčích projektů a to především z důvodů větší přehlednosti a oddělení základních vrstev architektury. Při implementaci požadavků bylo v maximální možné míře využito dostupných služeb a nástrojů, které jsou popsány v teoretické části této práce.

6.1.1 Scms.Web

Primárním projektem, který implementuje Prezentační vrstvu, Vrstvu aplikačního rozhraní a část Middleware vrstvy, je projekt **Scms.Web**. Jedná se o ASP.NET Core 1.0 webovou aplikaci, obsahující veškeré prvky MVC a MVVM, které jsou podrobněji popsány v kapitolách 3.5.2 a 3.5.3. Projekt má základní strukturu, která je popsána v teoretické části v kapitole 3.3.1. Implementací Prezentační vrstvy se rozumí HTTP koncové body v adresáři Controllers. Tento adresář obsahuje dva základní soubory, kterými jsou HomeController a AdminController. Jak již název napovídá, první zmíněný obsahuje koncové body pro vykreslování stránek uživatelské části Scms, druhý koncové body pro administrační část Scms.

Ve složce Controllers je však k nalezení ještě adresář pojmenovaný „api“. Tento adresář obsahuje také Controllers (MVC). Jedná se o Controllers pro Vrstvu aplikačního rozhraní, také označovanou jako WebApi. Každý Controller v tomto adresáři obsahuje koncové body různých CRUDQ operací vždy pro danou entitu. Jedná se o způsob zveřejnění CRUDQ operací, popsaných v kapitole 5.5, pomocí REST[10] architektury. Díky tomu lze

[10] REST je architektura pro manipulaci s daty přes HTTP protokol.

volat vybrané operace nad daty pomocí asynchronního JavaScriptu přímo z klientského počítače, na kterém je aplikace Scms otevřena ve webovém prohlížeči. Tím se eliminuje nutnost načítání celé stránky při každé operaci a je možné zobrazit průběh procesu zpracování. To přispívá k přívětivosti uživatelského rozhraní.

V úvodu této kapitoly bylo uvedeno, že součástí Scms.Web projektu je také částečná implementace Middleware vrstvy. Tím je myšlen především adresář Core, obsahující objekty a služby pro generování JavaScripového modelu, popsaného více v kapitole 6.3.1.

6.1.2 **Scms.Utils**

Projekt, který obsahuje čistě jen Middleware vrstvu, je Scms.Utils. Jak již název napovídá, jeho hlavním účelem je separování pomocných metod do samostatného projektu, který poté lze využívat i v ostatních projektech bez nutností kopírování kódu. Jeho hlavní náplní jsou implementace objektu Outcome (kapitola 5.5.5) a služby Shield (kapitola 5.5.6).

6.1.3 **Scms.Model**

Datový model aplikace Scms je v samostatném, odděleném projektu Scms.Model. Mimo jiné se jedná o třídy, popsané v diagramech v kapitole 5.6. Projekt Scms.Model obsahuje dva adresáře, přičemž první z nich se jmenuje DataStore a druhý Entity.

Adresář DataStore obsahuje strukturu autorem představené architektury datové základny přesně tak, jak je navržena a podrobně popsána v kapitole 5.5.3. Skládá se tak z jednoho souboru IDataStore, sloužícího jako vstupní bod při práci s daty, a následně pak adresáře Collections a Operations, které jsou popsány v již zmíněné kapitole o struktuře CRUDQ architektury. Tuto část projektu Scms.Model lze označit jako Transportní vrstva, jelikož obsahuje rozhraní pro práci s datovou základnou a zajišťuje její vyměnitelnost.

Druhý adresář, Entity, pak obsahuje strukturované zdrojové soubory datového modelu aplikace Scms. Jedná se o základní, jednoduchou strukturu, kterou není nutné popisovat hlouběji.

6.1.4 **Scms.DataApi**

Jako poslední projekt, který je součástí aplikace Scms, je projekt s implementací datové základny. Obsahuje tedy soubory s třídami, které implementují rozhraní IDataStore,

a zároveň třídy, které zajišťují přístup do databáze, v tomto případě konkrétně do dokumentové databáze MongoDB. Podrobnější popis teto implementace je k dispozici v kapitole 6.2.

Projekt Scms.DataApi lze označit jako Datovou vrstvu architektury Scms, jelikož zajišťuje přístup přímo do databáze, ale zároveň je použit pouze ve spojení s Transportní vrstvou, tedy přesně dle požadavků.

6.2 Implementace CRUDQ a využití DI

Tato podkapitola si klade za cíl popsat implementaci autorem navržené obecné architektury pro přístup k datové základně, a zároveň předvést její výhody v kombinaci s Dependency Injection službou, která je součástí využitého frameworku .NET Core 1.0.

6.2.1 Implementace IDataStore rozhraní

Hlavní stavební prvek architektury CRUDQ je rozhraní IDataStore. Toto rozhraní má zajišťovat přístup k dalším rozhraním (operační a detailní operační rozhraní, viz 5.5.2.2). Implementace tohoto rozhraní tak nemůže být nikde jinde, než v Datové vrstvě, respektive tedy v projektu Scms.DataApi. Tento projekt obsahuje soubor **DataStore**, který je právě zmíněnou implementací uvedeného rozhraní. Samozřejmostí jsou potom přidružené adresáře a soubor, obsahující další implementace jednotlivých operačních a detailních operačních rozhraní. Struktura a názvosloví jsou stejné jako v CRUDQ architektuře, z čehož plyne, že adresář Operations obsahuje implementace rozhraní ICreate, IRead, IUpdate, IDelete a IQuery. Adresář Collections poté obsahuje implementace detailních operačních interface.

Za zmínku stojí také adresář Mongo. V něm je k nalezení interface IDataClient a třída DataClient. Druhé zmíněné je, jak již název napovídá, implementací prvního zmíněného. Popis konkrétního zpracování této třídy a interface však již nemá pro účely této práce znatelný význam, neboť se nejedná o součást CRUDQ architektury. Stačí pouze uvést, že tato třída využívá knihovnu MongoDb.Driver, díky které je umožněn přístup do databáze MongoDB.

6.2.2 Využití DI

Při implementaci jakékoliv objektově orientované aplikace či programu bývá často problém rozhodnout se, jaký životní cyklus budou různé objekty mít, v jaké části životního cyklu samotné aplikace budou instancovány, a kde budou jejich instance udržovány. Toto rozhodování je v případě použití DI (kapitola 3.5.1.5) výrazně usnadněno a programátor se může soustředit na samotnou strukturu vytvářené třídy objektu a návrh algoritmů.

Architektura CRUDQ byla od začátku navržena tak, aby mohla být využita i v aplikacích, využívajících DI. I to je jeden z důvodů, proč je zmíněná architektura primárně vytvářena prostřednictvím interface, které jsou následně implementovány.

Jak již bylo popsáno v předchozích kapitolách, projekt Scms.Model obsahuje interface IDataStore. Tento interface je implementován v projektu Scms.DataApi, kdy tento projekt využívá pro implementaci MongoDB databázi. Obě tyto části jsou následně spojeny v projektu Scms.Web, kde je ve startup.cs souboru, konkrétně v metodě ConfigureServices (viz. 3.3.1), následující část kódu.

```
#if DNX451
// Předání implementace IDataStore (pouze pro .Net 4.5.1)
services.AddTransient<IDataStore, Scms.DataApi.DataStore>();

// Předání implementace IDataClient
services.AddSingleton<Scms.DataApi.Mongo.IDataClient,
Scms.DataApi.Mongo.DataClient>();

// Spuštění základní konfigurace datové základny
Scms.DataApi.DataClientSettings.Configure();
#endif
```

Ukázka kódu 15 - Předání implementace IDataStore pomocí DI

Ukázka kódu 15 představuje reálné možnosti využití DI pro implementaci použití CRUDQ architektury. Za povšimnutí stojí právě třetí řádek, kde je DI službě předáno nastavení o tom, že kdekoliv bude vyžadována implementace IDataStore interface, má předat instanci Scms.DataApi.DataStore. Podrobnější popis funkce DI v APS.NET Core 1.0 je k nalezení v teoretické části práce, v kapitole 3.3.2.

```
public class PagesController : BaseApiController
{
    private IDataStore dataStore;

    // Instance argumentu store je předána do konstruktoru
    // automaticky DI službou. V Startup.cs je totiž
    // nastaveno, že IDataStore je implementováno
    // třídou objektu Scms.DataApi.DataStore
    public PagesController(IDataStore store)
    {
        dataStore = store;
    }

    [HttpGet("{id}")]
    public IActionResult ById([FromRoute] string id)
    {
        // V tuto chvíli lze volat metody dle CRUDQ
        var readOutcome = dataStore.Read.Page.ById(id);

        // Následuje využití objektu Outcome v praxi
        if (!readOutcome.Conclusion)
        {
            return HttpNotFound(readOutcome.ErrorMessage);
        }

        return SuccessResponse(readOutcome.Output);
    }

    // Další metody
}
```

Ukázka kódu 16 - Ukázka praktického využití DI a CRUDQ architektury

Útržek z implementace Controlleru pro práci s objektem Page je v Ukázka kódu 16. Zde je vidět reálné využití DI pro CRUDQ architekturu. V konstruktoru Controlleru PagesController je jako jediný argument „store“, typu IDataStore. Vzhledem k tomu, že pro tento interface byla zaregistrována implementace (Ukázka kódu 15), je tak automaticky předávána při každém vytvoření objektu PagesController. V metodě ById(..) je k vidění již volání konkrétní implementace metody pro načtení objektu Page dle id.

6.3 Využití MVC a MVVM

V projektu Scms.Web, konkrétně v adresáři Core, je jeden ze stěžejních aspektů pro usnadnění využití MVVM přístupu k návrhu a implementaci uživatelského rozhraní. Jedná se o **KnockoutModelFactory** a **KnockoutModelBuilder**. Obě třídy jsou dílem autora této práce a slouží k automatickému generování JavaScriptového datového modelu (ViewModelu) aplikace, obsahující mimo jiné obalení všech vlastností objektů speciální

funkcí frameworku KnockoutJS – funkcí ko.observable(..). Více o této funkci a popisu frameworku KnockoutJS v kapitole 3.6. Důvod ke zvolení KnockoutJS frameworku je detailně popsán v kapitole 5.6.1. Uvedený generátor modelu je ta část aplikace Scms, která velmi značně usnadňuje propojení MVC a MVVM, což je popsáno v návrhu v kapitole 5.6.

6.3.1 **Fungování generátoru modelu**

Základní fungování generátoru vychází z použití jedné ze známých vlastností jazyka C#, a tou je reflexe[11]. Ta je zde využívána pro načtení všech vyžadovaných objektů a také pro zjištění všech jejich vlastností. Toho je následně využito pro manuálním sestavování JavaScriptového kódu, obsahující objektový model, který je dále využit v klientské části aplikace Scms.

6.3.1.1 KnockoutModelFactory

Část generátoru, která se stará o nastavení typů objektů (tříd), jež mají být vygenerovány do JavaScriptového modelu, se jmenuje KnockoutModelFactory. Jak již název napovídá, tato třída je „továrna“ na JavaScriptový model pro framework KnockoutJS.

Stěžejní částí popisované třídy je konstruktor, obsahující mnohonásobné volání metody PrepareObject(…). Jako argument je této metodě předáván vždy objekt typu TypeInfo[12]. Z toho důvodu je volána právě hned několikrát pro všechny třídy, které mají být součástí generovaného modelu.

[11] Reflexe v jazyce C# odkazuje na funkcionalitu, která dovoluje shromažďování informací o definovaných objektech pomocí speciálních funkcí.

[12] Typ TypeInfo je třída s metadaty o třídě objektu.

```
public KnockoutModelFactory()
{
    knockoutModelBuilder = new KnockoutModelBuilder();

    PrepareObject(typeof(Page).GetTypeInfo());
    PrepareObject(typeof(PageContent).GetTypeInfo());
    PrepareObject(typeof(CarouselBlock).GetTypeInfo());
    // Následuje volání dalších typů ..
}
```

Ukázka kódu 17 - Konstruktor KnockoutModelFactory

Metoda PrepareObject(…) prochází pomocí reflexe všechny vlastnosti třídy, a pro každou z nich vygeneruje ekvivalent v JavaScriptovém modelu prostřednictvím zavolání příslušné metody v KnockoutModelGenerator.

Samotný reálný generátor JavaScriptového kódu je součástí třídy KnockoutModelGenerator. Funguje na principu takzvaných „builders“, což znamená, že pro vygenerování typu je nutné volání metody pro započetí typu, následně libovolný počet volání o zapsání určité proměnné, a nakonec volání s požadavkem pro uzavření typu. Toto lze reálně vidět v metodě PrepareObject(…). Její základní implementace je vyobrazena v následující ukázce.

```
private void PrepareObject(TypeInfo value)
{
    knockoutModelBuilder.StartNewType(value);
    foreach (var property in value.GetAllProperties())
    {
        knockoutModelBuilder.StartProperty(value, property);
        knockoutModelBuilder.EndProperty();
    }
    knockoutModelBuilder.EndType();
}
```

Ukázka kódu 18 - Implementace metody PrepareObject

6.3.1.2 KnockoutModelBuilder

Třída, která se stará o samotnou konstrukci JavaScriptového kódu, se jmenuje KnockoutModelBuilder. Způsob volání a použití této třídy je popsán v předchozí kapitole 6.3.1.1. Při vytváření této třídy se autor nechal inspirovat přístupem, který je aplikován v implementaci serializátoru dat do formátu JSON knihovny Newtonsoft.Json. Jednotlivé

metody této třídy tak slouží pro započetí či ukončení určitého typu bloku. Stejně jako ve všech programovacích jazycích, i v tomto případě je nutné dodržet pořadí při vnořených voláních. Jinými slovy to znamená, že pokud je zavolána metoda StartNewType(…) a následně StartProperty(…), tak po uzavření typu musí být nejdříve zavoláno uzavření vlastnosti metodou EndProperty(), a až poté může být zavoláno EndType().

Velmi důležitou metodou je **ReadModel()**, která ukončí veškeré generování, sestaví požadovaný JavaScriptový kód a vrátí ho jako svůj návratový typ.

6.3.2 Použití

Výše popsaný generátor JavaScriptového modelu je v aplikaci Scms využit pro vytvoření ViewModelu architektury MVVM. V souboru Startup.cs (viz. kapitola 3.3.3) je v metodě ConfigureServices(…) na konci následující část kódu:

```
// Build KnockoutModel
KnockoutModelFactory factory = new KnockoutModelFactory();
var model = factory.Builder.ReadModel();
var dataModel = System.IO.File.Create(env.WebRootPath +
"/js/admin/global/dataModel.js");
var logWriter = new System.IO.StreamWriter(dataModel);
logWriter.WriteLine(model);
logWriter.Dispose();
```

Ukázka kódu 19 - Volání generátoru JavaScriptového modelu

Na druhém řádku je vytvoření instance třídy KnockoutModelFactory, která má ve svém konstruktoru přípravu všech nastavených tříd objektů k vygenerování. Následuje řádek, kde se volá již třída KnockoutModelBuilder a nad ní ihned metoda ReadModel(), popsaná výše. Zbytek úryvku kódu už se stará jen o přepis souboru dataModel.js, který obsahuje právě generovaný model. Vzhledem k tomu, že tato metoda je volána vždy při startu webové aplikace, **JavaScriptový model je vždy aktuální**.

6.4 Využití Gulp pro balíčkování

Nástroj Gulp, popsaný v kapitole 3.3.5 je v projektu Scms využit pro složení několika zdrojových JavaSctiptových souborů do jednoho. Je však nastaveno, aby nalezl všechny .js soubory v adresáři /js/admin a jeho podadresářích. Byla využita také možnost vyloučení některých souborů. Konkrétně jsou vyloučeny všechny .js soubory, které končí na „.init.js“.

Toto řešení autor připravil proto, aby výsledný sjednocený soubor obsahoval všechny připravené JavaScript soubory, ale aby vynechal ty, které slouží k inicializaci, a je nutné je načíst až později při vykreslování stránky.

Obsah souboru gulpfile.js je k dispozici v příloze, ve zdrojových souborech webové aplikace.

6.5 Testování

Pro dosažení potřebné funkcionality bylo nutné aplikaci Scms při vývoji průběžně testovat. Testování aplikace probíhalo během vývoje. Pro testování byly mimo jiné využity nástroje vývojového prostředí Visual Studio 2015.

6.5.1 Testování funkcionality

Testování funkcionality probíhalo následující metodikou:

1. Nasazení aplikace (viz. 6.6)
2. Výběr aplikačního/implementačního požadavku ke kontrole
3. Testování, zda daná funkcionalita v systému existuje
4. Testování, zda daná funkcionalita pracuje správně
5. Zapsán výsledek testu
6. Pokud nebyly otestovány všechny požadavky, návrat ke kroku 2
7. Ukončení testu a zápis zjištění

Tato metodika byla postupně aplikována na všechny požadavky a aplikace **ve všech testech uspěla**, neboť implementace požadavku byla vždy obsažena a byla zároveň plně funkční.

6.5.2 Unit testování

V době psaní této práce nebyl framework .NET Core testovatelný pomocí běžných unit testů, nabízených vývojovým nástrojem Visual Studio. Z toho důvodu byly tyto testy zcela vynechány.

Avšak díky tomu, že je aplikace Scms navržena s ohledem na Dependency Injection, bylo možné využít testovací Framework xUnit [28], ve kterém autor připravil celou řadu

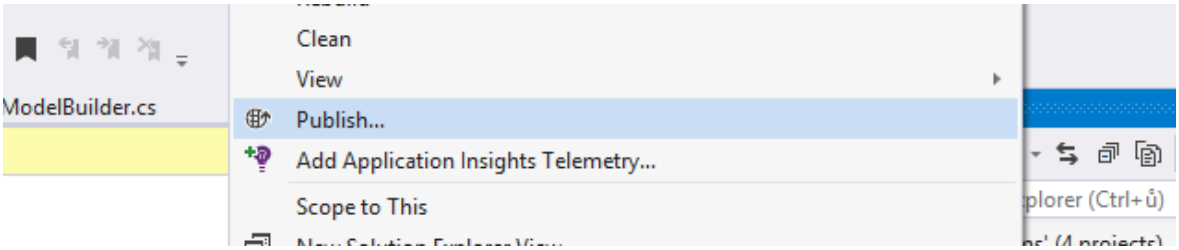
automatizovaných unit testů. Vzhledem k tomu, že se ale nejedená o oficiální a konečné řešení, nebyly tyto testy zahrnuty v komplexním řešení Scms, přiloženém v příloze 10.1.

6.6 Nasazení

Aby byla webová aplikace Scms dostupná veřejně, je nutné ji nejdříve zpřístupnit na nějaké webové adrese. Toho lze docílit buď spuštěním webové aplikace prostřednictvím běhového prostředí DNX s parametrem web, nebo nasazením aplikace do nejnovější služby IIS. V příloze se zdrojovými soubory (viz. 10.1) je k dispozici již hotový balíček celé aplikace, připravený ke spuštění/nasazení.

6.6.1 Vytvoření balíčku

Výše uvedený balíček byl vytvořen pomocí nástroje Publish, který je součástí Visual Studio 2015. Tento nástroj slouží k vytvoření spustitelného sestavení projektu. V rámci takzvaného „publish procesu“ dojde ke znovusestavení celé aplikace, kompilaci, provedení podpůrných úloh a složený všech souborů do jednoho adresáře s pevně danou strukturou.



Obrázek 10 - Nástroj Publish ve Visual Studio 2015

Dle zvoleného typu „Publish“ je dále s uvedeným adresářem nakládáno různými způsoby. Mimo kopírování tohoto adresáře do nastaveného adresáře je možné nechat webovou aplikaci nasadit rovnou na vzdálený webový server pomocí služby WebDeploy. V této práci byl však zvolen první přístup. Výsledný balíček, připravený pomocí služby „Publish“ je tak možné nalézt v příloze 10.1 s adresáři /App.

6.6.2 Spuštění

Obecně je pro spuštění .NET Core webové aplikace potřeba pouze běhové prostředí DNX a zadání jednoduchého příkazu „**dnx web**“ v příkazové řádce, přičemž je nutné se nacházet v adresáři s balíčkem projektu. Alternativně lze cestu k projektu zadat pomocí

argumentu „--project“. Jak již bylo uvedeno, kromě běhového prostředí DNX je možné aplikační balíček také předat hostovacímu serveru IIS, což ovšem není náplní této práce.

Postup spuštění konkrétní aplikace Scms je detailněji popsán v příloze 10.2.

7 Výsledky a diskuse

7.1 Hodnocení splnění požadavků

V této kapitole bude provedeno zhodnocení naplnění zadaných obecných a implementačních požadavků na webovou aplikace Scms.

Hlavní požadavek, kterým byla implementace jednoduchého, uživatelsky přívětivého CMS s integrovanou administrační částí, byl splněn. Aplikace byla vytvořena pomocí nejnovějších technologií a nástrojů v oblasti vývoje webových stránek. Nejobtížnější byla patrně práce s hlavním aspektem této webové aplikace, kterým byl .NET Core framework. Ten byl ještě v době psaní této práce v beta-verzích, což mělo za následek nestabilitu, časté pády a větší či menší odchylky od dokumentace. Finálně použitá verze naštěstí většinu zmíněných neduhů odstranila, či výrazně zlepšila, čímž byla implementace značně urychlena.

Aplikace Scms nabízí administrační část, ve které se nachází správa jednotlivých stránek, správa položek v menu a základní nastavení vzhledu. Všechny tři základní funkční požadavky tak byly splněny. Mezi hlavní pomocné nástroje, které velmi výrazně zefektivnily práci a zkrátily implementační čas, patří především framework KnockoutJS v kombinaci s autorem navrženým generátorem JavaScritpového modelu. Ve spojení s autorem navrženou architekturou datové základny webové aplikace tak vytvořili kombinaci, jejíž implementační doba byla zkrácena na minimum, avšak i přes to byla zachována přehlednost a relativní jednoduchost kódu.

Díky tomu, že aplikace využívá MongoDB, je připravena na splnění požadavku na spustitelnost na různých operačních systémech. Bohužel, ovladač databáze pro MongoDB, kompatibilní s .NET Core frameworkem, je ještě ve vývoji. Z toho důvodu v současné době aplikaci není možné spustit na jiném operačním systému než Windows. Všechny ostatní nástroje a frameworky, které byly použity, jsou již multiplatformní a mají takové licence, které nebrání využití Scms bez jakýchkoliv poplatků. Tímto byl splněn i požadavek na finanční nenáročnost.

7.2 Využitelné výstupy práce

7.2.1 Nasazení jednoduchého webu

Autorem vytvořená aplikace Scms je základní, jednoduchý CMS systém, který je možné kdykoliv nasadit tak, jak je, a začít používat. Nalezne uplatnění tam, kde bude zadavatel vyžadovat příjemný vzhled připravený i pro mobilní telefony, nekomplexní nastavení, moderní uživatelské rozhraní, a mimo jiné také schopnost běhu aplikace na jednom ze tří nejrozšířenějších desktopových operačních systémů.

7.2.2 Využití architektury CRUDQ

Architektura datové základny, navržená autorem této práce, je obecná a lze ji aplikovat na libovolné projekty vyžadující přístup, zápis a čtení z databáze. Mezi hlavní výhody této architektury patří její připravenost na použití s IoC kontejnerem, jasně daná struktura kódu i souborového sytému a v neposlední řadě také intuitivní použití. Jako reálnou ukázku implementace lze využít zdrojové kódy webové aplikace Scms.

7.2.3 Využití propojení MVC a MVVM

V rámci této práce autor navrhl mimo jiné vlastní řešení pro spojení dvou dobře známých architektur – MVC a MVVM. Každá z nich nalezne uplatnění na jiných místech, avšak lze je velmi efektně a efektivně propojit. Tímto je myšlen především autorem navržený a implementovaný generátor JavaScriptového modelu, který zásadním způsobem ulehčuje práci s ViewModelem na klientské straně.

7.2.4 Šablona při vývoji webových aplikací

Teoretická východiska, poznatky a výstupy této práce lze použít jako určitý druh manuálu, demonstrující propojení celé řady moderních webových technologií a nástrojů, které ve výsledku přinášejí významný synergický efekt.

7.3 Možná vylepšení

V oblasti vývoje kterékoliv aplikace či programu téměř nikdy nenastane situace, ve které by bylo možné označit dílo za kompletně hotové, do kterého není nutné již nijak

zasahovat. U webových aplikací toto pravidlo platí dvojnásobně. Webová aplikace Scms v době psaní této práce obsahuje implementaci předem daných požadavků. V průběhu krátkého používání finální verze však autor narazil na pár nedostatků, které by v případné další verzi měly být odstraněny.

Mezi hlavní současné nedostatky patří nekompatibilita s jiným operačním systémem než Windows. To je však dáno aktuální neexistencí určité verze ovladače zvolené dokumentové databáze. V brzké době však takovýto ovladač existovat bude, a bylo by vhodné jej vyměnit za ten aktuální. Aplikace tak bude zcela multiplatformní.

Další možná vylepšení se týkají spíše menších záležitostí v oblasti uživatelského rozhraní. To by vyžadovalo drobné změny týkající se rozložení určitých ovládacích prvků a doplnění některých chybějících, které by výrazně usnadnili práci s Scms. Mezi další kroky by jistě mělo patřit také doimplementování plnohodnotného uživatelského rozhraní pro vkládání a editaci vlastních obrázků a implementace dalších druhů ContentBlock objektů, jako například blok pro přehled obrázků či blok s kontaktním formulářem.

8 Závěr

Hlavní cíl práce se podařilo splnit návrhem a implementací plně funkčního CMS systému Scms, skládajícího se z webové prezentační části a administrační části. Jak se uvádí v kapitole 4.1, administrační část je složena ze správy stránek, správy navigace a z nastavení vzhledu, čímž splňuje požadovanou funkcionalitu. Dle zadání je tento systém vytvořen v prostředí ASP.NET Core 1.0 (dříve ASP.NET 5, viz. 3.1.1), s využitím frameworku MVC 6. Požadavek na zvolení vhodného, multiplatformního databázového systému byl splněn v kapitole 5.3 výběrem dokumentové databáze MongoDB.

Navržením a implementací vlastní architektury CRUDQ v kapitole 5.5 byl splněn dílčí cíl navržení architektury datového zdroje. Autorem zpracovaná architektura rozděluje veškeré datové operace na pět základních typů, přičemž každý typ operace lze aplikovat na libovolnou entitu z datového modelu. Všechny operace mají pevně dané místo v objektové struktuře, i ve struktuře souborového systému. Hlavním prvkem architektury je typ Interface (rozhraní), ze kterých je sestaven celý strom operací. Díky použití typu Interface, jako základního stavebního prvku, je navržená architektura nejen velmi dobře testovatelná a vhodná pro použití v kombinaci s Dependenci Injection, ale především umožňuje snadnou záměnu databázového systému, čímž splňuje jeden z hlavních požadavků na tuto architekturu.

Naplnění druhého dílčího cíle, a sice implementaci vlastního řešení pro efektivní propojení serverové a klientské strany webové aplikace, bylo docíleno v kapitole 6.3 vytvořením vlastního generátoru JavaScriptového datového modelu, obsahujícího funkcionalitu pro použitý MVVM framework KnockoutJS. Jak je uvedeno v této práci, díky tomuto generátoru je možné na serverové i klientské straně používat naprosto stejný datový model entit, přičemž ten na klientské straně je obohacen ještě o takzvaný Dependency Tracking, zajišťující velmi snadné provázání s uživatelským rozhraním.

9 Seznam použitých zdrojů

[1] S. Hanselman, „ASP.NET 5 is dead,“ 19 1 2016. [Online]. Dostupné: http://www.hanselman.com/blog/ASPNET5IsDeadIntroducingASPNETCore10AndNETCore10.aspx.

[2] E. Lipton, „Renaming to ASP.NET Core 1 and Entity Framework Core 1,“ GitHub, 1 22 2016. [Online]. Dostupné: https://github.com/aspnet/Announcements/issues/144.

[3] W3Techs, „Historical trends in the usage of server-side programming languages for websites,“ W3Techs, 1 2 2016. [Online]. Dostupné: http://w3techs.com/technologies/history_overview/programming_language. [Přístup získán 3 2016].

[4] J. Sharp, Microsoft Visual C# 2010 : krok za krokem, Praha 4: Albatros Media a. s., 2012.

[5] S. Smith, „Introducing .NET Core,“ 2015. [Online]. Dostupné: https://docs.asp.net/en/latest/conceptual-overview/dotnetcore.html.

[6] Community, „GitHub - CoreFX,“ Microsoft, 2016. [Online]. Dostupné: https://github.com/dotnet/corefx. [Přístup získán 2016].

[7] B. Massi, „Understanding .NET 2015,“ MSDN Blog, 25 2 2015. [Online]. Dostupné: https://blogs.msdn.microsoft.com/bethmassi/2015/02/25/understanding-net-2015/. [Přístup získán 18 2 2016].

[8] MSDN, „What's New in Visual Studio 2015,“ Microsoft, 2015. [Online]. Dostupné: https://msdn.microsoft.com/en-us/library/bb386063.aspx. [Přístup získán 2016].

[9] R. Andreson, „Instalation of ASP.NET 5,“ Microsoft, 2 2016. [Online].

[10] S. Smith, „Understanding ASP.NET 5,“ Microsoft, 2015. [Online]. Dostupné: http://docs.asp.net/en/latest/conceptual-overview/understanding-aspnet5-apps.html. [Přístup získán 2 2016].

[11] M. Jones, „Using Gulp.js and the Task Runner Explorer in Visual Studio 2015,“ 18 1 2016. [Online]. Dostupné: http://www.exceptionnotfound.net/using-gulp-js-and-the-task-runner-explorer-in-asp-net-5/. [Přístup získán 27 2 2016].

[12] K. A. Gaurav a H. Sam, The SOLID Principles - Approach to Write Robust Programs, 2015.

[13] S. Smith, „Dependency Injection,“ Microsoft, 2016. [Online]. Dostupné: http://docs.asp.net/en/latest/fundamentals/dependency-injection.html. [Přístup získán 2016].

[14] S. Smith, „Application Startup,“ Microsoft, 12 2015. [Online]. Dostupné: http://docs.asp.net/en/latest/fundamentals/startup.html.

[15] S. Smith, „Logging,“ Microsoft, 2016. [Online]. Dostupné: http://docs.asp.net/en/latest/fundamentals/logging.html. [Přístup získán 4 3 2016].

[16] E. Reitan, S. Addie a D. Roth, „Using Gulp,“ Microsoft, 2016. [Online]. [Přístup získán 15 2 2016].

[17] .NET Foundation, „Documentation,“ .NET Foundation, 2016. [Online]. Dostupné: http://docs.nuget.org/. [Přístup získán 17 2 2016].

[18] Microsoft, „Building your first MVC 6 application,“ Microsoft, 2016. [Online]. Dostupné: http://docs.asp.net/en/latest/tutorials/first-mvc-app/index.html. [Přístup získán 6 2 2016].

[19] R. Anderson, „Introduction to Tag Helpers,“ Microsoft, 2016. [Online]. Dostupné: http://docs.asp.net/en/latest/mvc/views/tag-helpers/intro.html. [Přístup získán 19 2 2016].

[20] O. Joe a M. Jason, Pablo’s SOLID Software Development, 2009.

[21] A. A. Zeeshan, Object Oriented Programming - Principle and Theory, C# Corner, 2015.

[22] D. Čápka, „MVC architektura,“ ITnetwork.cz, [Online]. Dostupné: http://www.itnetwork.cz/navrhove-vzory/mvc-architektura-navrhovy-vzor/. [Přístup získán 4 2 2016].

[23] R. Anderson, „Adding a controller,“ Microsoft, 2016. [Online]. Dostupné: http://docs.asp.net/en/latest/tutorials/first-mvc-app/adding-controller.html. [Přístup získán 18 2 2016].

[24] knockoutjs.com, „Documentation,“ knockoutjs.com, [Online]. Dostupné: http://knockoutjs.com/documentation/introduction.html. [Přístup získán 3 2 2016].

[25] DB-Engines, „DB-Engines Ranking,“ DB-Engines, [Online]. Dostupné: http://db-engines.com/en/ranking. [Přístup získán 5 3 2016].

[26] M. Delaney, „Support for .NET Core,“ MongoDB, 1 29 2015. [Online]. Dostupné: https://jira.mongodb.org/browse/CSHARP-1177. [Přístup získán 14 3 2016].

[27] W3Techs, „Historical trends in the usage of JavaScript libraries for websites,“ W3Techs, 1 3 2016. [Online]. Dostupné:

http://w3techs.com/technologies/history_overview/javascript_library/all. [Přístup získán 6 2 2016].

[28] xUnit.net, „About xUnit.net,“ xUnit.net, [Online]. Dostupné: https://xunit.github.io/. [Přístup získán 28 1 2016].

[29] MSDN, „.NET Core and Open-Source,“ Microsfot, 2016. [Online]. Dostupné: https://msdn.microsoft.com/en-us/library/dn878908(v=vs.110).aspx#Anchor_0. [Přístup získán 1 2016].

[30] D. Roth, „DNX Overview,“ Microsoft, [Online]. Dostupné: http://docs.asp.net/en/latest/dnx/overview.html. [Přístup získán 23 2 2016].

[31] DevIQ, „Explicit Dependencies Principle,“ DevIQ, 2014. [Online]. Dostupné: http://deviq.com/explicit-dependencies-principle/.

[32] P. Csaba, „SOLID: Part 1 - The Single Responsibility Principle,“ EnvatoTuts, 13 12 2013. [Online]. Dostupné: http://code.tutsplus.com/tutorials/solid-part-1-the-single-responsibility-principle--net-36074. [Přístup získán 4 2 2016].

[33] P. Csaba, „SOLID: Part 3 - Liskov Substitution & Interface Segregation Principles,“ EnvatoTuts, 24 1 2014. [Online]. Dostupné: http://code.tutsplus.com/tutorials/solid-part-3-liskov-substitution-interface-segregation-principles--net-36710. [Přístup získán 10 3 2016].

[34] P. Csaba, „SOLID: Part 4 - The Dependency Inversion Principle,“ EnvatoTuts, 13 2 2014. [Online]. Dostupné: http://code.tutsplus.com/tutorials/solid-part-4-the-dependency-inversion-principle--net-36872. [Přístup získán 10 3 2016].

[35] T. Kirda, „Angular vs. Knockout: Similarities and Fundamental Differences,“ Devbridge Group, 1 12 2014. [Online]. Dostupné:

https://www.devbridge.com/articles/angular-vs-knockout-similarities-and-fundamental-differences/. [Přístup získán 5 2 2016].

[36] BSONSpecs, „BSON,“ Creative Commons, [Online]. Dostupné: http://bsonspec.org/. [Přístup získán 3 2 2016].

10 Přílohy

10.1 Zdrojové soubory Scms

Zdrojové soubory aplikace Scms jsou na přiloženém DVD.

10.2 Instalace Scms

Na přiloženém DVD je také soubor Instalace.docx, obsahující návod na spuštění aplikace Scms na operačním systému Windows.